Event-Driven Message Passing and Parallel Simulation of Global Illumination

Tomáš Plachetka

2003

Event-Driven Message Passing and Parallel Simulation of Global Illumination

a dissertation
submitted to the faculty
of computer science, electrical engineering
and mathematics
of the university of paderborn
in partial fulfillment of the requirements
for the degree
doctor rerum naturalium
(Dr. rer. nat.)

by
Tomáš Plachetka
2003

Event-Driven Message Passing and Parallel Simulation of Global Illumination

dissertationsschrift
vorgelegt in der fakultät
für elektrotechnik, informatik und mathematik
der universität paderborn
zur erlangung des akademischen grades
doctor rerum naturalium
(Dr. rer. nat.)

von
Tomáš Plachetka
2003

Acknowledgements

There are many people who supported this work. I can mention only a few of them here but I am thankful to them all. Firstly I would like to express my sincerest thanks to my advisor, Prof. Dr. Burkhard Monien from the University of Paderborn, who was always very understanding when I was floating midway between science and engineering and who always gave me enough freedom in making decisions about what to do next.

I would like to thank Prof. Dr. Branislav Rovan and Prof. Dr. Peter Ružička from Comenius University in Bratislava, who turned my attention to parallel and distributed computing a long time ago. I would also like to thank Dr. Andrej Ferko, Dr. Ľudovít Niepel and Dr. Eugen Ružický from Comenius University, who showed me that computer graphics is an interesting area of research where parallel processing can be very helpful.

I was lucky to meet many other people who would never say no when I needed to discuss problems or ideas even though they did not directly match their own research interests. In the last few years I consulted mostly with Dr. Ulf-Peter Schroeder, Axel Keller, Prof. Dr. Friedhelm Meyer auf der Heide, Dr. Olaf Schmidt, Dr. Jürgen Schulze, Dr. Thorsten Falle, Dr. Rainer Feldmann, Dr. Ulf Lorenz and Dr. Adrian Slowik. Thank you.

My further thanks go to all my colleagues at the University of Paderborn, Comenius University in Bratislava, Sheffield Hallam University, to the people I worked with on various projects, and to my students and technical staff. They all contributed to making our working days a joy. Special thanks to Geraldine Brehony for the proofreading of a great part of this text.

I am obliged to all my friends who did not forget me when I was spending more of my spare time with books and computers than with them. Finally, my deepest gratitude goes to my whole family, especially to my parents and to my little sister, for their love and patience.

Contents

Chapter 1

Introduction

One of the goals of computer graphics is a *photorealistic image synthesis*. There are many applications which require photorealistic visualisation of three-dimensional (3D) environments which often do not actually exist. Film production, computer game production and architectural design are industrial areas in which photorealism is very important and it is thanks to these areas that computer graphics is flourishing today.

The requirements of the quality and speed of visualisation of 3D environments strongly depend on the application. An engineer observing a flow of air around a wing is more interested in the visualisation of air speed and air pressure than in a photorealistic image of the wing. A mechanical engineer designing a screw is the majority of the time interested in a simple flat-shaded projection of the screw on the computer screen rather than in a visualisation with reflections and shadows. Even when photorealism is desirable (e.g. in movies or in computer games), there are several levels of it. The choice of an appropriate rendering engine is always a matter of compromise. There is no "best" method, which is suitable for everyone. This thesis concentrates on photorealistic image synthesis. Throughout this thesis, *rendering* denotes the process of creating images and *rendering system* (or *rendering engine*) denotes a system which implements this process. Images rendered by a rendering system are visualisations of a 3D environment. The process of rendering solves the *global illumination problem*.

Although all rendering algorithms used in computer graphics only compute an approximation of real illumination, they are all computationally expensive. The simplest ones (which compute either no illumination or only its very crude approximation) are implemented in the hardware of graphics cards. More sophisticated rendering algorithms are implemented in the software and only use the hardware of graphics cards in their final visualisation phases (if ever). This thesis deals with parallelisations of photorealistic rendering algorithms which is one way of speeding them up.

The following section sketches the problems of the efficient programming of parallel applications using available standards. The next section informally ex-

plains how photorealism is measured and how photorealistic images are computed. Formal definitions are given later. An outline of this thesis can be found in the last section of this chapter.

1.1 Current parallel programming standards

Two standards for the support of programming of parallel applications are available today: Parallel Virtual Machine (PVM) [GBD+94] and Message Passing Interface (MPI) [MPI94], [MPI97], [Gro02]. PVM and MPI provide an application programmer with an abstract layer of message passing, which hides differences between various parallel system architectures. Both PVM and MPI assume that a parallel application consists of (sequential) processes that communicate using message passing. The processes do not share any memory.

The performance of the PVM and MPI implementations is measured using a set of benchmarks, by reporting statistics on the measured throughput and latency of message passing, speed of message assembly and decoding etc. The performance of PVM and MPI can be compared to the performance of lower-level communication libraries on some benchmarks. The loss of performance due to the use of PVM or MPI on the top of a lower-level communication library is usually not very high. Unlike the lower-level libraries, PVM and MPI can be (and have been) ported to practically all architectures without changes in their application programming interfaces. The porting of applications that build on PVM or MPI is therefore much easier (only compiling and linking is usually needed). The portability on the source code level pays out the slight performance loss. However, the set of benchmarks is not complete. A benchmark is missing which would reflect the needs of non-trivial parallel applications.

1.1.1 Polling in non-trivial parallel applications

Contemporary implementations of PVM and MPI do not allow for the efficient implementation of many important parallel applications: shared-memory simulation libraries, parallel databases, media servers, parallel simulations of global illumination, ... Although the applications look very different, two independent activities can always be observed when focusing on one process:

- T_1: a computation with an occasional communication
- T_2: servicing of requests coming from other processes

We call applications with this pattern of behaviour *non-trivial*.[1] When explaining to non-experts what non-trivial applications are, we use the following analogy:

[1] All non-trivial parallel applications belong to the class of *irregular parallel applications*. Communication patterns in irregular applications cannot be predicted.

Imagine a large building with many windows. All the windows must be cleaned. There are, say, four workers to complete the task. If cleaning the windows was their only responsibility, each worker would simply choose a dirty window, clean it and then look for another dirty window.

In addition to cleaning the windows, the four workers must build a brick wall behind the corner. The wall must be four layers high and each worker is responsible for building one whole horizontal layer—worker one lays bricks on the ground (layer one), worker two lays bricks on top of the bricks in layer one etc. Notice the dependencies between the workers. For instance, worker two cannot begin until worker one has first laid at least two bricks on the ground.

Both tasks, cleaning the windows *and* building the brick wall must be accomplished as soon as possible. The laying of the bricks clearly has a higher priority than cleaning the windows because a worker laying bricks creates work for the workers who are responsible for the higher layers.

There are several ways how cleaning the windows can be interleaved with building the brick wall. Here is an example of an inefficient one. All workers begin with cleaning the windows. From time to time, each worker interrupts the cleaning of the windows and walks to the wall to see whether he can add a brick to his layer. If so, he adds a brick and returns to cleaning the windows. If not, he immediately returns to cleaning the windows. This is called *active polling* (or *busy waiting*). Obviously, if "from time to time" means often (say, every one minute), then all the workers waste time and energy by walking to the wall (also when there is no work for them on the wall) and back to the windows. If "from time to time" means seldom (say, once an hour), then a worker who has just returned to cleaning the windows will continue cleaning them for the next hour—even though in the meantime there may be a more important work to be done on the brick wall.

A more efficient work organisation would keep the workers busy with laying the bricks whenever possible. When a worker who is laying bricks creates work for some other worker, he should shout at the other worker to stop cleaning the windows and come lay bricks instead. When a worker who is laying bricks cannot continue laying them, he should return to cleaning the windows and should keep cleaning the windows until he is called to the wall again. Such a work organisation is called *demand-driven*.

PVM and MPI are highly optimised for the performance of a single task (cleaning windows or building a brick wall) by several workers. However, active polling must be used when two (or more) tasks are combined. Our goal was to understand where the active polling originates from and how it can be avoided. The problem is related to a *lack of thread-safety in existing PVM and MPI implementations* and to an *imprecise semantics of asynchronous communication in the MPI Standard*.

1.2 Photorealistic image synthesis

This section gives a brief introduction to the computer synthesis of photorealistic images. Before arriving at a design of photorealistic rendering systems and algorithms implemented in the systems, several philosophical questions must be answered.

1.2.1 Measures of photorealism

Why do we say that some images look realistic whereas others do not? An important aspect of measuring the degree of realism involves human perception. A healthy human eye together with the nature around it make a perfect rendering system. The images produced by this system are *real*. If we replace the human eye with a good photographic camera, we get another rendering system which is similar to the original one in the sense that the images produced by this system are indistinguishable (by a human eye) from the real images. We might replace both nature and the human eye with a piece of software and compute the images on a computer—and the images might still be undistinguishable from the real ones! *A photorealistic rendering system, no matter how it works, should produce images that a human eye cannot distinguish from real (or photographic) images of a real environment.* The harder it is for a human eye to distinguish an image from a real one, the more photorealistic the image is.

This intuitive measure of realism has a flaw. If we compute an image of a non-existing environment on a computer, we have nothing to compare the picture to. The computed image usually corresponds to a photograph of some environment which might exist and which we can recognise. However, also in this case we must use our imagination and experience instead of a direct visual comparison to judge the quality of the computed picture.

Another flaw of the above definition of realism is that a human eye (or rather the human visual perception system) is easy to fool. Nice unreal pictures sometimes appear "realistic". Synthetic beasts in movies can appear real at first glance even though a closer study of the images reveals that they are in fact not. An untrained eye is very tolerant even of very obvious mistakes, especially in movies. For instance, the movie *Indecent proposal* (with Robert Redford and

Demi Moore) [Lyn93] contains at least two relatively long image sequences which contain a forgotten microphone hanging from above. (None of the people we have asked noticed the hanging microphone.) It is interesting to note that it also works the other way around—ugly "non-realistic-looking" pictures may be real! There is a strange shadow above the staircase in *Café Central* in the german city of Paderborn. People who work with computer graphics must be saying to themselves: "This shadow is too sharp to be realistic." (None of the people we have asked noticed the surprising appearance of the shadow.) Further examples of where our visual perception is likely to fail are the wonderful images of M. C. Escher's work [EBe92], stereograms [Ent93] and other optical illusions [Per85].

The previous discussion suggests that the measuring of photorealism based on the visual impression is unreliable and should be replaced by a more formal model. Such mathematical models do exist but their practical use implies the simplification of assumptions on the behavior of light. In other words, the quality of images can be evaluated with respect to a chosen mathematical model but not against the reality which they represent. Also, a comparison of software implementations of different rendering systems is usually impossible due to different simplifying assumptions made during the implementation of the systems. This is why mathematical quality measures are sometimes combined with a subjective perception of images, even if it means a retreat to the imprecise "psychological" definition of realism [McN00].

1.2.2 Photorealistic rendering systems

The final result of the rendering of a 3D environment is an image. The image can be created using a number of methods. It can be taken by a camera, painted by an artist or computed by a program. These different rendering systems may produce the same or similar images—however, they differ internally and their use depends on the purpose of the images.

When we see a nice scenery, we might want to take a picture of it because we are either not likely to be there again or the scenery is perhaps likely to change. A camera or a painter can put the image into a form which can be stored practically forever (a photographic film, a painting).

A painter is more flexible than a photographic camera. A painter does not need to see the scenery while painting if he or she remembers what the scenery looks like. The painter's imagination can also produce images of a scenery as seen from different perspectives or viewed under different lighting conditions. Moreover, a painter can paint images of environments which do not exist. The paintings can still be very realistic—comparable to photographic pictures. (However, realism is usually not what a painter tries to achieve because an artistic perfection is different from the measures discussed in the previous section and cannot be very well formalised.)

It is even more flexible to create and store a computer model of a 3D environment and to postpone the rendering of images for later. The model consists of surfaces, surface materials, atmosphere and light sources. Later on, (virtual) cameras can be added into the model and a rendering program can be used to render images of what the cameras "see".

The task of a realistic 3D artist is to create a realistic *3D model*. When we have a perfect model of a 3D environment, we would also like the images taken by the virtual cameras to be perfect. We do not want the artist to retouch the computed images, to manually paint shadows on the floor or mirroring on a glass table or to brush-up a brick wall to make it look like a brick wall. A 3D artist should concentrate on the modeling, not on making images. *A photorealistic rendering system should correctly simulate all light phenomena (or at least phenomena which are relevant to the given model).*

All of the above examples of rendering systems (a passive observer, a camera, a realistic painter, a rendering program) solve instances of the *global illumination problem*. The task of a 3D artist is to set up an instance of the global illumination problem.

1.2.3 Light phenomena and their simulation

The nature of light has been studied for many centuries. The most comprehensive theory on the behaviour of light is quantum electrodynamics [Fey88]. The basic assumption of this theory is that light is carried by particles called *photons*. Photons are emitted from light sources and transport energy through a medium (air, for instance) until they hit a surface. The interaction of a photon with a surface results in the absorption of the photon and in its scattering (reflection, refraction) which produces a new photon or photons. Photons do not travel along straight lines from their origin to their destination—they travel along arbitrary "crooked" paths in space. The intensity of light as measured at a point in space is an integral of contributions of many photons traveling along all possible paths between the light source and the destination.

Even though quantum electrodynamics is the best existing theory which explains the behaviour of light, it cannot be directly used in *macro-scale computer graphics*—it is too expensive to simulate the transport of light on a subatomic level. The modeling paradigms and algorithms used in computer graphics are based on *geometric optics* (developed by Sir Isaac Newton [New52]) which makes several simplifying assumptions.

The basic assumption of geometric optics is that photons only travel along straight lines. *Scattering is only allowed on surfaces.* In other words, the interaction of photons with participating media is either ignored or only certain types are allowed (in most algorithms it is easy to include media that only absorb light).

Another assumption is that *light is monochromatic*. This means, that all emitted photons have a single frequency. This assumption is made in order

to simplify the representation of "colour" (computation with vectors is more convenient than computation with continuous spectra).

Many light phenomena are completely ignored in computer graphics or are handled in a special way (when they are important to the application): *diffraction* ("bending" of light around obstacles), *interference* (an effect that can be observed on thin surfaces such as oil films or soap bubbles), *polarisation* (the scattering of light on surfaces such as water or glass depends on the orientation of the electric vector of the incident light beam), *fluorescence* (molecules of some materials absorb photons and then emit new photons at a different frequency, which makes clothes "glow in the dark" under certain lighting conditions), etc.

Some computer graphics algorithms (e.g. the basic radiosity algorithm) only assume an ideal diffuse reflection and no transmission, whereas others (e.g. the basic ray tracing algorithm) only assume an ideal (indirect) specular reflection and an ideal specular transmission.

Nevertheless, most of the light phenomena which are ignored are not relevant to applications of computer graphics (the relevant phenomena can usually be incorporated into the chosen rendering algorithm). Two important rendering algorithms are *ray tracing* and *radiosity*. Ray tracing is a view-dependent algorithm which traces rays (photons) from the observer's eye to the light sources. Radiosity is a view-independent algorithm which solves a linear equation system in order to compute the distribution of light on a finite number of patches which approximate surfaces of the 3D model. Even though ray tracing and radiosity are incomparable in almost all respects, they both compute solutions of the global illumination problem. The global illumination problem was formally defined by James T. Kajiya [Kaj86] as a Fredholm integral equation of the second kind. Ray tracing and radiosity algorithms solve special cases of *Kajiya's rendering equation*, using approximations which simplify the equation. Modern rendering algorithms generally try to avoid approximations and directly solve the Kajiya's equation, using probabilistic methods.

1.3 Outline of this thesis

This thesis is organised as follows.

Chapter 2 deals with the efficient parallel programming using message passing. The current standard communication libraries, PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) do not allow an efficient implementation of large parallel algorithms. The problem is not specific to parallel global illumination—practically all larger parallel applications are forced to use active polling (also known as busy waiting). Active polling not only diminishes performance and destroys the natural structure of parallel programs but it also leads to a non-deterministic message passing latency. The reason why active polling cannot be avoided is that PVM and MPI implementations which are currently

available are either thread-unsafe or (even worse) they are thread-safe but active polling is hidden inside the libraries. We extended both PVM and MPI libraries by using a new interrupt mechanism which allows for the writing of parallel programs without active polling. The interrupt mechanism does not make the PVM and MPI libraries completely thread-safe—nevertheless, it makes it possible to write multi-threaded applications without active polling (we call this property quasi-thread-safety). Moreover, we developed a communication library called TPL (Thread Parallel Library) which builds upon the extended PVM and MPI standards and which is thread-safe (on the set of its communication functions). Thread-safety alone is not enough. We define a formal framework for message passing which builds on the well-accepted parallel processing models and which helps to explain the semantical drawbacks of the existing message passing models such as PVM, MPI or CORBA. In particular, we explain how asynchronous communication can be formally looked at and why the specification of asynchronous communication in the above systems does not match the semantics defined in fundamental abstract message-passing models. TPL is not just another communication library, it is a straightforward implementation of our framework. TPL offers (unlike PVM, MPI or CORBA) asynchronous communication on parallel systems using standard hardware and software components.

Chapter 3 presents and explains a formal definition of the global illumination problem. Approaches to the global illumination problem are presented and their advantages and limitations are discussed.

Chapter 4 is devoted to the ray tracing method and its parallelisation. A novel perfect demand-driven load-balancing algorithm is presented, its optimality is proved and its exact message complexity is given. One disadvantage of a straightforward demand-driven parallelisation of ray tracing is that the 3D model must be replicated in the memories of all processors. We solve this problem by using a distributed object database. Each processor "owns" a subset of objects of the 3D model and besides that it maintains a memory for the storing of other objects. If a processor needs an object which is currently not stored in its memory, it interrupts the computation, makes place for the object in its cache and then sends an object-request to the object's owner. The caching policy tries to minimise the number of object-requests in order to reduce the communication between processors. We compare the efficiency of several caching policies in order to choose the most appropriate one. Most importantly, we experimentally show that the performance of parallel ray tracing is *strongly* influenced by the choice of the communication library. This result is not specific to parallel ray tracing. *An empirical comparison of the performance of parallel algorithms may be very biased if polling is used in the implementation of the algorithms or the communication library.*

Chapter 5 deals with the radiosity method. We concentrate on an integration of a two-pass algorithm which consists of a radiosity pass (which computes a view-independent diffuse illumination) and a ray tracing pass (the second pass

adds some of the view-dependent illumination effects to the precomputed radiosity solution). For the radiosity pass we use the shooting algorithm (Southwell relaxation) with a Monte Carlo form factor computation. The main goals behind this choice are a correct handling of materials during the radiosity pass and a seamless integration with the ray tracing pass. These ideas are not entirely new. Our novel contribution is a combination of the energy transfer with a state of the art Monte-Carlo form factor computation in one step. This combined step is always performed on the top level of the subdivision hierarchy without a loss of accuracy of the radiosity solution. An important positive aspect is an automatisation of the algorithm (minimisation of the number of parameters which control the algorithm).

Chapter 6 gives a summary of the results presented.

Chapter 2

Event-driven message passing

Parallel processing is a very common practice nowadays. Much of it is hidden inside chips (general-purpose processors and graphics cards), inside schedulers of operating systems which run processes on shared memory multiprocessor machines etc. This chapter focuses on efficient message passing parallel programming on a larger scale. Most parallel applications on this scale use either PVM (Parallel Virtual Machine) or MPI (Message Passing Interface).

One problem of the current implementations of the PVM and MPI standards is that many of them are not thread-safe (there is no thread-safe implementation of PVM and there is no freely available implementation of MPI). This forces application programmers to use unnatural and inefficient *active polling* (also known as *polling* or *busy waiting*) in many important applications which we call non-trivial. (More precisely, there are several implementations of PVM and MPI which *are* thread-safe, but the active polling is hidden *inside* the libraries—which is even worse than the active polling inside applications![1]) We explain where the active polling comes from and present the previous approaches to the problem. Then we present an interrupt mechanism which makes both PVM and MPI implementations quasi-thread-safe. Quasi-thread-safety allows the programming of multi-threaded parallel non-trivial applications without active polling. We also sketch how PVM and MPI implementations can be made completely thread-safe, without active polling.

The second (perhaps less apparent) problem of the above standards is a lack of asynchronous communication (PVM) or its imprecise semantics (MPI), respectively. This also leads to polling in the applications which require asynchronous communication, although the form of this polling is slightly different. We explain and solve this problem as well.

[1] MPI/Pro by MPI Software Technology, Inc. seems to be an exception. [DS02]

2.1 Non-trivial parallel applications

There are surprisingly many parallel applications (including simulation of global illumination) which cannot be implemented in an efficient way using current PVM or MPI implementations. These applications can be clearly characterised and form a class of *non-trivial parallel applications*.

Definition. *Non-trivial parallel application* is a parallel application which consists of parallel processes which do not share any memory and communicate via message passing. Each process performs two independent activities, as depicted in Fig. 2.1. T_1 and T_2 operate on the same data (shared memory). •

T_1	CPU intensive computation, communicating occasionally with other processes.
T_2	Fast servicing of requests coming from other processes.

Figure 2.1: Two independent activities in one process of a non-trivial application

Practically all the larger parallel applications belong to this class. (There is no more exaggeration in the previous sentence than in the sentence "Every larger sequential application needs a dynamical memory management.") Examples of non-trivial parallel applications are distributed databases (or applications which use a distributed database), media servers, shared memory simulation libraries, parallel scientific computations which use application-independent load balancing libraries etc.

2.2 Development of parallel programming

In order to explain what is missing in the modern parallel programming standards (PVM, MPI) as regards active polling in non-trivial applications, we will shortly look at the roots of parallel programming. Out of a large number of parallel computers, programming paradigms and programming languages, we chose Inmos Transputer and Occam as a representative sample. Even though Transputers have not survived the technological progress of the last twenty years, many ideas which were originally implemented in Transputers are still valid. Particularly, the problem of non-trivial applications is solved in Transputers.

Many parallel computer architectures, parallel programming paradigms, parallel computer languages, parallel algorithms etc. have been developed over the past 20 years. The demand came from the need to solve problems which required more computing power than single-processor computers could offer (e.g.

weather prediction, fluid dynamics simulation, simulation of global illumination). These problems can usually be divided into subproblems which can be solved independently of one another and their results are then combined together. The subproblems can be mapped onto independent processing elements (which can exchange data). All processing elements simultaneously compute their subproblems, thus reducing the total computational time required.

Long discussions on the choice of parallel programming model resulted in questions such as:

- Should the parallelism be expressed explicitly (by a programmer) or should a machine (a compiler or an operating system or a processor) automatically generate a parallel program from a sequential one?
- Should the underlying hardware be a specialised hardware or just a collection of standard computers connected in a network?
- Should message passing or shared memory be used for communication?
- How fine grained should the parallelism be?
- Should the processing be synchronous or asynchronous?
- Should the communication be synchronous or asynchronous?
- What should the interconnection network look like?
- Should there be a special programming language for expressing parallel algorithms or should existing ones be used?
- ...

Some of these questions are important to developers of parallel computers, operating systems and compilers, some to parallel application programmers and some to theoreticians working on parallel algorithms. Almost any combination of answers to the questions above is correct. The resulting models are equivalent in the sense that they can be mapped onto each other. But who should do the mapping?

The diversity of ways how parallel processing can be looked at may have been one of the reasons why parallel applications are still so rare. Clearly, a standard which would allow the writing of parallel programs which last longer than the hardware and system software used was missing.

Transputers [GK90] and the Occam programming language [Gal96] were one of the first historical attempts to establish a standard, which seamlessly connects hardware, programming paradigm, programming language and parallel algorithms. Transputers have not survived the technological progress, but in our opinion they significantly influenced the development of parallel computing.

2.2.1 Occam programming language

Sir William of Occam (1284–1347) was an English philosopher. He advocated a principle known as *Occam's Razor*: "Pluralitas non est ponenda sine necessitate". ("Entities must not be multiplied beyond what is necessary.") In other words, "If there are several solutions or approaches to a problem then the simplest one should be used." The Occam programming language (named after William of Occam) is indeed minimalistic. It is not our intention to formally or completely describe the Occam language or the Transputer. We will only define a small subset of Occam in order to demonstrate in an example how simple and at the same time powerful the language is. We will also show how non-trivial applications are supported by Occam and the Transputer.

Occam requires the programmer to explicitly express the parallelism. An Occam program consists of a set of processes which can run either sequentially or in parallel. Each of the processes can consists further of processes which can again run either sequentially or in parallel. This nesting is potentially unlimited but must end up with atomic processes. Processes running in parallel are not allowed to share variables—they are only allowed to share communication channels and explicitly exchange data using message passing.

The idea of the process nesting was introduced by Hoare. [Hoa85]. Occam is a "materialisation" of Hoare's CSP model (Communicating Sequential Processes). It is easy to schematically visualise every Occam program, thanks to the process nesting and channel declarations. The visualisation can only include the first level of the program structure (or a few first levels) which is usually sufficient to understand the idea of the parallelisation without going into implementation details. (INMOS developer's toolsets included an integrated editor which allowed the programmer to browse the structure of the program and edit its code.)

The basic Occam `concepts` are:

- **Data and data types.** Occam provides the programmer with the usual basic data types BOOL, BYTE, INT, REAL. Data can be organised in arrays (this corresponds to *replication* in the Occam's terminology). Array indices are integers and begin with 0. A variable is local to the process in which it was declared (and in its subprocesses). Processes running in parallel are not allowed to share any variables—an exception to this is a variable which is not altered by any of the parallel processes. Variables must be declared and strongly typed.

- **Processes.** Processes are organised in a hierarchical manner. An Occam program is a single process. This single process consists of either one atomic process (assignment, input, output, `SKIP`, `STOP`) or a constructor (`SEQ`, `PAR`, `IF`, `CASE`, `WHILE`, `ALT`) which combines processes themselves are either atomic processes or constructors. The structure is thus always a tree. Indentation denotes process nesting in an Occam program. There is

no dynamic memory management in Occam (or in Transputer). Recursion cannot be expressed in Occam (it must be simulated).

- **Channels.** Channel is the only means of communication between two parallel processes (each channel *must* be shared by exactly two parallel processes). Occam channels are uni-directional (a process is allowed to either read from or write to a channel but not both) and synchronous (a process which tries to read from or write to a channel becomes blocked until the process on the other end of the channel is ready to communicate). Channels, as with variables, must be declared and strongly typed. "Type" of a channel is the protocol used on that channel (a description of the sequence of data types passed through the channel). **Timers** are treated as special channels with only one end which can be read. Timers are used to either read the clock value or to block the reading process for a specified time.

Occam defines 5 **atomic processes**:

Assignment. Assigns a value to a variable and terminates.
`variable := value`

Input (receive). Reads a value from a channel, stores it into a variable and terminates. This process blocks until the process at the other side of the channel is ready to write to the channel.
`channel ? variable`

Output (send). Writes a value to a channel and terminates. This process blocks until the process at the other side of the channel is ready to read from the channel.
`channel ! value`

No action. Does nothing and terminates. This process is used in some constructors to explicitly express that no action should be taken.
`SKIP`

Stop. Does nothing and never terminates (blocks forever). This process is usually used to indicate an error condition.
`STOP`

The atomic processes can be combined to more complex processes using **constructors**. Two of the most important constructors are:

`SEQ, a chain of sequential processes.` A process of this constructor begins after the previous has terminated. The `SEQ` constructor terminates when its last process has terminated.

```
SEQ
  process_1
  process_2
  ...
  process_N
```

`PAR, a collection of parallel processes.` All processes run in parallel. The `PAR` constructor terminates when all its processes have terminated.

```
PAR
  process_1
  process_2
  ...
  process_N
```

The remaining constructors are `IF`, `CASE`, `WHILE`, and `ALT`, whereby the first three correspond to the constructors which are known from other programming languages. `ALT` is a special constructor which acts as an input multiplexor. It allows a process to wait for a number of events and execute an action when an event happens. When data can be received from several channels, then one of the events is triggered (an arbitrary one). The following example involves a process which blocks until it receives data either from $\texttt{channel}_1$ (executing $\texttt{action}_1$ in this case) or from $\texttt{channel}_2$ (executing $\texttt{action}_2$ in this case):

```
ALT
  channel_1 ? variable_1
    action_1
  channel_2 ? variable_2
    action_2
```

Furthermore, each channel input can be accompanied by a boolean condition (a guard). An event is triggered when a channel becomes readable and at the same time the boolean condition guarding the channel is satisfied. Guarded inputs can be combined with non-guarded ones. An `ALT` may also contain one special event, `TRUE`, which is triggered when none of the other events is triggered.

All constructors can be **replicated** in Occam. A replicated `SEQ` corresponds to a sequential loop. For instance, the program in Fig. 2.2 computes $j = 2^{10}$ (the inner `SEQ` is replicated).

A replicated `PAR` starts a collection of parallel processes. The processes run the same code. The program in Fig. 2.3 starts 10 parallel processes which are connected to a pipeline. The value 1 is passed to the pipeline by a special `source` process which is connected to the input end of the pipeline. The pipeline computes 2^{10} which is passed as a result to another special process, `sink`, which is connected to the output end of the pipeline (the two special processes run in parallel with the 10 pipeline processes).

```
INT j:
SEQ
  j := 1
  SEQ i = 1 FOR 10
    j := 2 * j
```

Figure 2.2: An example of a replicated `SEQ`. The program computes $j = 2^{10}$

```
[11]CHAN OF INT chpipe:
PAR
  chpipe[0] ! 1
  PAR i = 0 FOR 10
    INT value:
    SEQ
      chpipe[i] ? value
      chpipe[i+1] ! 2 * value
  INT j:
  chpipe[10] ? j
```

Figure 2.3: An example of a replicated `PAR`. The program computes $j = 2^{10}$

A replicated `ALT` is usually used in the multiplexing of input from an array of channels. Fig. 2.4 shows a very artificial example of a replicated `ALT`.

```
[10]CHAN OF INT ch:
PAR
  PAR i = 0 FOR 10
    ch[i] ! 2
  INT j:
  SEQ
    j := 1
    SEQ i = 0 FOR 10
      ALT k = 0 FOR 10
        ch[k] ? value
          j := j * value
```

Figure 2.4: An example of a replicated `ALT`. The program computes $j = 2^{10}$

Processes can be assigned names in Occam. This improves the readability of a program. Parameters can be passed to named processes. The pipeline which computes 2^{10} (Fig. 2.3) can be written as shown in Fig. 2.5.

An Occam program is independent of the underlying hardware. The hardware is usually a Transputer or a network of Transputers. There are special language extensions for the specification of the *mapping of processes onto processors and*

```
PROC pipe(CHAN OF INT in, CHAN OF INT out)
  INT value:
  SEQ
    in ? value
    out ! 2 * value
:

PROC source(CHAN OF INT out)
  out ! 1
:

PROC sink(CHAN OF INT in)
  INT j:
  in ? j
:

[11]CHAN OF INT chpipe:
PAR
  source(chpipe[0])
  sink(chpipe[10])
  PAR i = 0 FOR 10
    pipe(chpipe[i], chpipe[i+1])
```

Figure 2.5: An example of named processes in Occam. The program computes $j = 2^{10}$ in `PROC sink`

mapping of channels onto physical links, whereby parallel processes of the topmost `PAR` constructor are typically mapped onto different processors. Mapping a program onto several processors only influences the efficiency of the program, not its semantics.

Example of an Occam program: Dining philosophers

The problem of dining philosophers [Dij71] is used in many textbooks as a motivating example of resource sharing. It helps in the understanding of the synchronisation of parallel processes and solution strategies to the deadlock problem. We chose this problem to show the power which is hidden in simplistic Occam constructors:

Problem of dining philosophers

> Five philosophers are sitting by a round table. They spend time alternately philosophing and eating spaghetti. Each philosopher has a plate in front of him. There is one fork between each pair of plates.

> Each fork is shared by two neighbouring philosophers. In order to eat, a philosopher needs both neighbouring forks, the left one and the right one. A philosopher is not allowed to take both forks simultaneously, he must decide which fork to acquire first.
>
> "Philosophing" should be interpreted (in the context of this problem) as doing nothing for a random length of time, while not holding any fork in one's hand. "Eating" should be interpreted as doing nothing for a random length of time, while holding both forks.
>
> The task is to write a program which simulates the life of the philosophers. The program must guarantee a *fairness*—a hungry philosopher must eventually get to eat. A problem which must be solved is the prevention of deadlock and the consequent starvation of the philosophers. A deadlock occurs for instance when all the philosophers acquire their left forks, in which case no fork remains on the table and all the philosophers will starve (unless at least one of them voluntarily returns the fork he is holding).

One possible solution to the deadlock problem involves the breaking of the symmetry among philosophers. For instance, if the philosophers are assigned numbers from 1 to 5 and if odd philosophers acquire forks in a reverse order to even ones, then deadlock will never occur. (There are also other solutions as to how deadlock can be prevented but we shall continue with this particular solution.)

It is very simple under these assumptions to write an Occam program which simulates the situation at the table. Philosophers, as well as forks, are processes which are connected to a ring (each philosopher communicates with the two forks next to him and each fork communicates with the two philosophers next to the fork). The program is shown in Fig. 2.6.

Note how the `fork` process is implemented. At the beginning a fork waits for a signal from any of its two neighbouring philosophers. Once a signal arrives, the fork is assigned to the philosopher who sent this signal and the fork begins listening only to that philosopher. Another signal from the same philosopher means that the philosopher returns the fork, and the fork begins listening to both philosophers again.

The `philosopher` process which is trying to acquire a fork simply sends a signal to the fork (in the `IF` constructor). The send will block when the fork is not listening to the philosopher (all communication in Occam is synchronous). A blocked send will unblock after the message has been received by the fork. Note that the sends after `eat()` (used for returning forks) never block.

The main program starts five instances of the `philosopher` process and five instances of the `fork` process in parallel, and connects the processes using channels to a circle. Fig. 2.7 shows a schematic visualisation (a process diagram) of

```
PROC fork(CHAN OF INT left, right)
  INT signal:
  WHILE TRUE
    ALT
      right ? signal
        right ? signal
      left ? signal
        left ? signal
:

PROC philosopher(INT id, CHAN OF INT left, right)
  WHILE TRUE
    SEQ
      think()
      IF
        (id REM 2) = 0
          left ! 0
          right ! 0
        (id REM 2) = 1
          right ! 0
          left ! 0
      eat()
      right ! 0
      left ! 0
:

[10]CHAN OF INT ph2fork:
PAR
  PAR i = 0 FOR 5
    philosopher(i, ph2fork[(2 * i + 9) REM 10], ph2fork[2 * i])
  PAR i = 0 FOR 5
    fork(ph2fork[2 * i], ph2fork[(2 * i) + 1])
```

Figure 2.6: Simulation of dining philosophers in Occam

the program in Fig. 2.6. Only the outermost `PAR` and the outermost channel declarations must be followed in order to draw a process diagram of any Occam program.

The Occam program is *extremely short*. (We did not go into any details concerning an output to a terminal or the generation of a random delay in `PROC think` as this is not important.) An equivalent program written in a conventional (sequential) programming language such as C or Pascal would be much bigger. Why? The reason being that the simulation of dining philosophers is

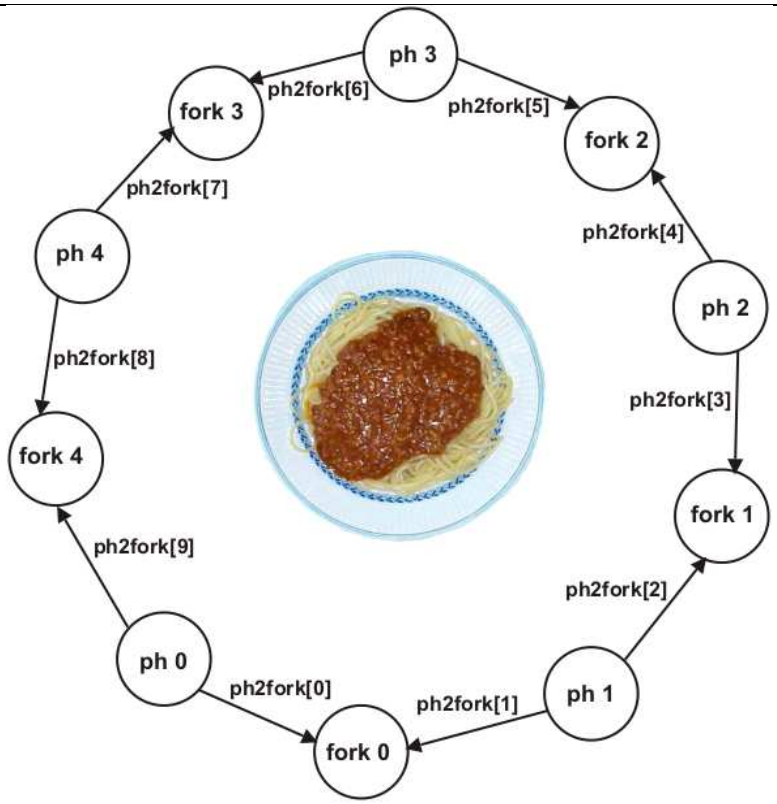

Figure 2.7: Simulation of dining philosophers, a process diagram

an inherently parallel problem. A sequential program written in a conventional programming language must implement the *scheduling* hidden behind the Occam `PAR` construction. A table with all process' states must be maintained and the independent executions of the processes must be simulated.

2.2.2 Transputer

Transputer was a processor (more precisely, a family of processors) introduced by a British company INMOS Ltd (now SGS-Thomson Microelectronics Ltd) in 1986. The processor (T414) was optimised to support parallel programming, particularly programs written in the Occam language (all Occam's constructors and atomic processes have their counterparts in Transputer's instructions). In the three years which followed there were 10 processors in the Transputer family, out of which T805 and T9000 were the most successful ones.

Fig. 2.8 shows a block diagram of the T805 Transputer. The features that makes this architecture special are the four links used to connect several Transputers to a network (pins `LinkIn0`...`LinkIn3` and `LinkOut0`...`LinkOut3`) and the 4 kB on-chip RAM. Transputers were often used in embedded systems—all they required was power (pins `GND` and `VCC`) and an external clock (`ClockIn`).

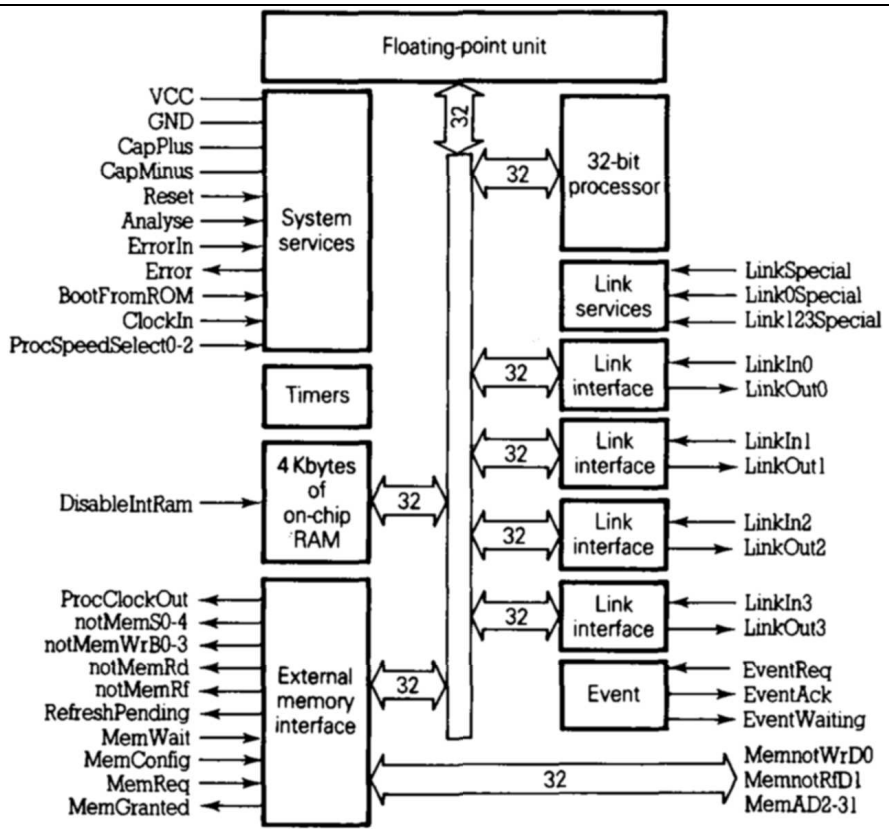

Figure 2.8: Hardware block diagram of the T805 Transputer

The full technical description of Transputer processors can be found in [GK90]. In the following section we will only explain the process scheduling in the Transputer, which is related to our work.

The basic concept of the Transputer low-level programming is a `process`. Processes in Transputer correspond to Occam's parallel processes which are created in the outermost `PAR` constructor. Each process is assigned a priority (high or low) and a fixed amount of memory (*workspace*) while it is being created. A process can be either running or blocked (waiting on a channel or timer).

Only one process can run at a time in one Transputer. Transputer maintains two priority queues, a high priority queue (HPQ) and a low priority queue (LPQ). Each process which is neither running nor blocked is stored in one of these queues, depending on the priority of the process. The process scheduling is micro-coded in hardware and the process switching is extremely fast. The scheduling works as follows:

- A running high priority process is never preempted. It runs until it terminates or blocks. When a high priority process blocks, it is placed at the end of the HPQ.

- When a blocked high priority process unblocks (becomes ready to run), three cases can arise:

  1. *No process is running.* In this case the unblocked high priority process is scheduled to run.
  2. *Another high priority process is running.* In this case the unblocked high priority process is placed at the end of the HPQ.
  3. *A low priority process is running.* In this case the running low priority process is preempted and placed at the end of the LPQ. The unblocked high priority process is scheduled to run. (This is the only case where the context of the running process must be saved.)

- A running low priority process runs until it is either preempted by a high priority process or until it runs longer than 50 ms. The former case has already been discussed above. In the latter case, the scheduler waits until the running process executes an instruction which leaves the process' context undefined (in doing so no context must be saved). In this situation there are two possibilities:

  1. *The LPQ is empty.* In this case the running process keeps running.
  2. *The LPQ is not empty.* In this case the running process is placed at the end of the LPQ and the first process of the LPQ is scheduled to run.

The following section shows that this scheduling model corresponds to the one needed for an efficient implementation of non-trivial applications which were introduced in Section 2.1.

2.2.3 Occam, Transputer and non-trivial parallel applications

Each of the parallel processes of a non-trivial application (see Section 2.1) needs to perform a local computation and at the same time quickly service requests from all other processes. The design of the Transputer allows for an efficient implementation of this scenario if each top-most process of the non-trivial application is mapped onto one Transputer.

A non-trivial application can be implemented in Occam as follows. Each top-most process is a `PAR` which contains two subprocesses, T_1 and T_2. T_1 does the computation, while T_2 carries out the servicing of incoming requests (Fig. 2.9).

A subtle problem must be solved when T_1 and T_2 share variables and when T_1 or T_2 alters a shared variable (it must be recalled here that Occam forbids the altering of variables shared by parallel processes). In such a case a third process,

```
PROC NONTRIVIAL([NR.IN] CHAN OF INT in, [NR.OUT] CHAN OF INT out)

  PROC T1()
    SEQ
      ...compute and occasionally communicate...
  :

  PROC T2()
    INT request:
    ALT i = 0 FOR NR.IN
      in[i] ? request
        ...service request...
  :

  PAR
    T1(in, out)
    T2(in, out)
:
```

Figure 2.9: Implementation of one process of a non-trivial application in Occam

DM (Data Manager) must run in parallel with T_1 and T_2. DM acts as a passive server which listens to T_1 and T_2. Only DM manipulates the "shared" data. T_1 or T_2 are only allowed to indirectly access the "shared" data (read or write them), via the passing of messages to DM—otherwise they work with local copies.

Presumably the incoming requests are sent by the processes which are blocked until the requests are answered. In order to increase the efficiency of the whole parallel program, the processes which are waiting for a response should be serviced as fast as possible. T_2 should therefore be prioritised over T_1. If there are no incoming requests, T_2 sleeps (it is blocked in `ALT`) and thus it does not prevent T_1 from running. If there is a request, we want T_2 to run *immediately*, possibly preempting T_1. T_1 should never preempt T_2. **Note that all these requirements are guaranteed by the Transputer hardware scheduling semantics if T_2 runs with high priority and T_1 with low priority** (the priority of DM is not important).

2.3 Current message passing standards: PVM and MPI

There is a subtle difference between *parallel* and *distributed* computing. The first term is used for computing on dedicated multi-processor machines which fit into a single box (e.g. Transputer-based systems). Distributed computing

usually denotes computing on a network of "ordinary" computers connected via an "ordinary" network.

The current trend involves writing *portable* parallel applications. The difference between parallel and distributed computing disappears—at least from the point of view of an application's programmer. It is more important to minimise the effort needed for the porting of a parallel application onto another architecture than to maximise the application's performance on a specialised hardware platform. This goal is achieved by developing standards which provide applications with abstract message passing functions and which can be efficiently mapped onto practically all the available parallel systems. PVM (Parallel Virtual Machine) [GBD+94] and MPI (Message Passing Interface) [MPI94], [MPI98], [MPI97] are parallel programming libraries (more precisely, specifications of application programming interfaces) which have been established as standards for parallel programming. The vast majority of parallel applications use either PVM or MPI.

The porting of a PVM or MPI application onto a new system usually simply means the compilation of the application on the new system. Most vendors of parallel machines and operating systems have a tailored implementation of PVM or MPI, optimised for their systems. The supported systems range from proprietary parallel hardware through shared memory multiprocessors or (possibly heterogeneous) workstation clusters to virtual computers which consist of loosely coupled computers in wide-area networks (Internet).

Even though PVM and MPI significantly differ (in their implementations as well as in their interfaces), they use common programming paradigms:

- The parallel machine which runs the program, regardless of what its architecture looks like, is viewed as a virtual parallel machine with message passing capabilities.
- A parallel application consists of parallel processes that do not share any memory. Explicit communication must be used to exchange data between processes.
- The actual implementation of the inter-process communication is transparent to the programmer. PVM and MPI provide the programmer with a set of communication functions, most important of which are point-to-point `send` and `recv` functions.
- PVM and MPI are libraries. A process of a parallel application is a sequential program written in a host language (e.g. C, C++, Fortran) and linked to PVM or MPI.
- Neither PVM nor MPI try to be minimalistic (as opposed to Occam). They provide the programmer with many functions which are not necessary in the

sense that they can be assembled from other functions. The additional functions make (arguably) programming applications more comfortable. Also, some of the functions can be implemented in a more efficient way inside the library than in the application.

2.4 Point-to-point message passing in PVM and MPI

PVM and MPI differ in their specifications of point-to-point communication. Some differences are obvious, e.g. a function that sends a message is called `pvm_send` in PVM whereas in MPI it is called `MPI_Send`. The functions also differ in the numbers and types of arguments, in their return values, ... This section does not deal with similar differences, instead it focuses on differences concerning the *semantics* of point-to-point communication.

Point-to-point message passing involves a message delivery between two parallel processes, *sender* and *receiver*. Each process is assigned a *rank* (called *task identifier* in PVM terminology) which is an integer distinct to the process. Each message is accompanied by a message *tag*, also an integer. The sender's and the recipient's ranks together with the message tag form a message *header*.[2] A message can contain data. The data is stored in a message *body*. The body may be empty (messages with an empty body are called zero-length messages).

The delivery of a message is regarded independently from the point of view of the sender and the receiver processes. The sender initiates a *send operation* (initiation of a send operation is sometimes called "posting a send"), which states the receiver to which the message is to be delivered and the message tag which will be used. The receiver initiates a *receive operation* (initiation of a receive operation is sometimes called "posting a receive"), which states the sender (or set of senders) from which a message is to be received and the message tag (or tags) which the message must have. A message can be delivered when an initiated (and not yet completed) send operation exists and when an initiated (and not yet completed) receive operation exists that match. The delivery of a message completes both the send and receive operations associated with the message.

Note that the existence of matching receive and send operations R and S does not guarantee that a message will be passed from the process which initiated S to the process which initiated R. There may be a third process in the system which initiated a send operation S' which also matches R. Any of the send operations S and S' may complete.

The communication library provides a somewhat weaker guarantee on the message delivery. This guarantee is formulated as the *progress rule* in the Message

[2]Implementation of a communication library may store additional information in the header, for example the size of the message in bytes.

Passing Standard. [MPI94], [MPI98], [MPI97]:

> **Progress rule.** If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same destination process.

Remark. The notions *operation*, *initiation of an operation*, *completion of an operation* are not precisely defined in the MPI standard (e.g. a send operation is defined as `MPI_Send`, a receive operation is defined as `MPI_Recv`). This leads to misunderstandings among MPI implementors and MPI users. Particularly the progress rule is often incorrectly interpreted. The progress rule not only applies to `MPI_Send` and `MPI_Recv` as the text of the standard might suggest—it also applies to all communication functions (to blocking as well as to nonblocking ones). We give a more formal definition of these notions in Section 2.5. •

The reference book on MPI [SOHL+95] uses a slightly different formulation of the progress rule:

> **Progress rule: blocking communication.** If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system. The send operation will complete, unless the receive is satisfied by another message. The receive operation will complete, unless the message sent is consumed by another matching receive posted at the same destination process.
>
> **Progress rule: nonblocking communication.** A communication is *enabled* once a send and a matching receive have been enabled communication posted by two processes. The progress rule requires that once a communication is enabled, then either the send or the receive will proceed to completion (they might not both complete as the send might be matched by another receive or the receive might be matched by another send). Thus, a call to `MPI_Wait` that completes a receive will eventually return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is nonblocking, then the receive completes even if no complete-send call is made on the sender side.
>
> Similarly, a call to `MPI_Wait` that completes a send eventually returns if a matching receive has been started, unless the receive is

satisfied by another send, and even if no complete-receive call is made on the receiving side.

If a call to `MPI_Test` that completes a receive is repeatedly made with the same arguments, and a matching send has been started, then the call will eventually return `flag=true`,[3] unless the send is satisfied by another receive. If a call to `MPI_Test` that completes a send is repeatedly made with the same arguments, and a matching receive has been started, then the call will eventually return `flag=true`, unless the receive is satisfied by another send.

2.4.1 Message assembling and sending

Prior to the initiation of a send operation, the sending process must tell the communication library which data should be sent in the message body (unless the message body is empty). This phase—which we call message assembling—can consist of several function calls when using PVM or MPI. We will only explain the so-called *packing* method here, the result of which is a contiguous memory buffer which contains the data to be sent (the message body).[4]

Remark. The packing functions do slightly more work than simply copying data from the user's memory space into the send buffer when the communication takes place in a heterogeneous system because the representations of data (e.g. the byte order or the length of basic types) in different processes may be different. There is a canonical encoding of all basic types defined in the POSIX standard [ISO90], called *XDR encoding*. The data in the send buffer is stored in the XDR encoding which guarantees their unique interpretation in the sender and receiver processes. •

After a message has been assembled, it can be sent to a recipient. The initiation of the send operation is implemented in `send` functions in the communication library. If a call to a `send` function does not return until the send operation has been completed, the function is called a *blocking send* (or a *synchronous send*). If the function is allowed to return before the completion of the send operation, it is called a *nonblocking send* (or an *asynchronous send*).

The crucial questions here are:

- Is the send buffer allocated and freed by the application or by the communication library?

[3] The argument `flag` is called `completed` in the next section.

[4] The packing method requires additional data copying. There are other methods of message assembly provided by both PVM and MPI that do not require the storing of the data in a contiguous buffer. However, our performance tests show that copying the data is not very costly compared to other actions as regards sending a message on commodity systems such as the hpcLine by Fujitsu-Siemens.

- How large must the send buffer be in order to store the packed (XDR) data?
- When is it safe to reuse or free the send buffer?

PVM

The PVM library internally provides the send buffers. PVM can maintain several send buffers at a time. There is always one *active send buffer* and all packing functions pack (append) the data into the active send buffer. The application can switch between buffers (by saving the current buffer and making another buffer the active buffer).

The process which wants to pack a message either calls the function

```
int pvm_initsend(int encoding)
```

which destroys the current active buffer and creates a new one, or

```
int pvm_mkbuf(int encoding)
```

which creates a new buffer without destroying the active one.

The application does not need to specify the size of the buffer—the PVM library is responsible for the provision of enough space to store the packed data.

After a buffer has been created, the message is assembled by calling one of the packing functions (there is one packing function for each basic type). For instance, this is a function that packs an integer (or an integer array) into the active send buffer:

```
int pvm_pkint(int *integer, int count, int stride)
```

After the message has been assembled in the active send buffer, it can be sent using the function

```
int pvm_send(int recipient, int tag)
```

This function sends the contents of the active message buffer to the recipient—more exactly, it initiates a send operation. The PVM library does not specify whether this function is a blocking or nonblocking send (it can safely be regarded as a nonblocking send). The call may return before the send operation has been completed. However, PVM guarantees that the active send buffer may be reused or freed after the return from a `pvm_send()` call, and it also guarantees progress formulated in the progress rule.

MPI

The MPI library requires the application to allocate and free send buffers. No buffer is necessary in the case of zero-length messages but if the message body is not empty, the application must decide on the size of the buffer to be allocated before packing data into it. The application does not know how large the packed representation of the data is. Therefore MPI provides a function

```
int MPI_Pack_size(int n, MPI_Datatype t, MPI_Comm com, int *size)
```

which returns in `size` the number of bytes necessary to store `n` items of type `t` sent via the communicator `com` (a communicator is a "channel" between the sender and receiver processes).

There is only one packing function in MPI:

```
int MPI_Pack(void *data, int n, MPI_Datatype t, void *buf,
   int size, int *offset, MPI_Comm com)
```

which packs `n` items of type `t` from the memory location `data` to the buffer `buf` of size `size` at offset `offset`. The buffer is intended to be sent via the communicator `com`. Note that the value of `offset` is updated by `MPI_Pack`. The new value of `offset` is used by the next call to `MPI_Pack` (`offset` must be set to 0 by the application when packing data into an empty buffer).

There is a variety of send functions in MPI and any of them can be used to send the data which is packed in a buffer—more precisely, to initiate a send operation. The functions differ only in the conditions guaranteed when they return.

```
int MPI_Send(void*buf, int offset, MPI_PACKED, int recipient,
   int tag, MPI_Comm com)
```

is a default send. Its semantics corresponds to the semantics of pvm_send in PVM. It is not specified whether the send operation is blocking or nonblocking, and so the call to `MPI_Send` may return before the send operation has been completed. However, MPI guarantees that the send buffer can be freed or reused by the application after the call has returned, and it also guarantees progress formulated in the progress rule.

```
int MPI_Ssend(void*buf, int offset, MPI_PACKED, int recipient,
   int tag, MPI_Comm com)
```

is a synchronous send. This function does not return until the initiated send operation has been completed. The completion of the send operation implies that the send buffer can be freed or reused.

```
int MPI_Isend(void*buf, int offset, MPI_PACKED, int recipient,
   int tag, MPI_Comm com, MPI_Request req)
```

is a nonblocking send which initiates the send operation and returns immediately, without waiting for its completion. The application must not free or reuse the send buffer until the send operation has been completed. In order to let the application detect the completion, `MPI_Isend` returns a handle to the send operation (a request), `req` and provides functions that test whether the operation has been completed:

```
int MPI_Wait(MPI_Request *req, MPI_Status *status)
```

blocks until the operation pointed to by `req` has been completed. Hence a return from the call implies that it is safe to free or reuse the buffer.

```
int MPI_Test(MPI_Request *req, int *completed, MPI_Status *status)
```

returns immediately, indicating in `completed` whether the operation pointed to by `req` has been completed or not.

```
int MPI_Bsend(void*buf, int offset, MPI_PACKED, int recipient,
   int tag, MPI_Comm com)
```

is a buffered send. Note that its interface is the same as the interface of the default `MPI_Send`. The same applies to its semantics, with one difference: `MPI_Bsend` first copies the message data from `buf` to an extra buffer space and then immediately returns. Note that it is safe for the application to reuse or free the buffer `buf` after the return from a `MPI_Bsend()` call. The extra buffer space is provided by the application which (prior to a MPI_Bsend() call) calls the function

```
int MPI_Buffer_attach(void *extra_buf, int size)
```

where `extra_buf` points to a block of memory which belongs to the application, of `size` bytes. A pairwise function

```
int MPI_Buffer_dettach(void *extra_buf, int size)
```

is used to reclaim the extra buffer space from MPI (so that the application can free or reuse it). The extra buffer space must be large enough to store message data of all buffered send operations that have not been completed at any one time. The function `MPI_Buffer_dettach()` blocks until all buffered sends that use the buffer complete.

Remark. There are other send functions in MPI which we have omitted here—we just presented the most important and representative ones. Note that MPI is much richer than PVM as regards the choice of send functions. However, as we show in Section 2.6.3, the seemingly richer set of send functions does not mean a richer functionality. The use of MPI functions which are related to asynchronous or buffered communication (that means the use of all send functions except the default and synchronous send) is in fact very limited. •

2.4.2 Message receiving and disassembling

Receiving a message differs from sending a message, even though there are certain similarities. One difference is that a receive operation can, but does not have to specify the sender from which it wants to receive a message. Similarly, a receive operation can, but does not have to, specify the tag a message must have in order to match the receive operation. This *wildcard* matching only applies to the receive operations, not to the send ones.

Prior to initiating a receive operation, a buffer must be allocated to the receiving process. However, how does the receiving process know the size of the buffer which must be allocated when it decides to receive any message? This is another difference between sending and receiving—the sender can compute the size of the buffer, whereas the receiver can not. It is also unclear at first glance whether the application or the communication library should allocate the buffer.

After a message has been received, the receiver usually wants to disassemble

the data stored in the message body. This is one more asymmetry between sending and receiving—the sender knows what the message body contains, whereas the receiver must rely on the fact that the sender assembled the message using an agreed method. In other words, the receiver relies on the fact that the sender obeys the agreed *protocol*.

Similarly to sending, a question arises when the buffer can be freed or reused. The answer is easy: the application (not the communication library) knows when it has disassembled all the data which it needed from the receive buffer.

PVM

The PVM library internally provides the receive buffer and it automatically adjusts its size to the size of the incoming message. There are several receive functions provided by PVM, the application can basically decide whether it wants to block until a matching message arrives (this corresponds to an initiation of a receive operation and waiting for its completion) or only probe whether there is a matching message (this corresponds to a temporary initiation of a receive operation, checking whether it can be matched against a send operation and canceling the receive operation after the check).

The blocking receive function

```
int pvm_recv(int from, int tag)
```

initiates a receive operation and blocks until a message which matches `from` and `tag` arrives. If the parameter `from` is set to -1, then a message coming from any process is matched (ranks of all processes are positive, the value of -1 serves as a wildcard). Similarly, `tag` set to -1 matches any message tag. The arrival of a matching message completes the receive operation and the function call returns to the application.

The nonblocking receive function

```
int pvm_nrecv(int from, int tag)
```

initiates a receive operation and checks for a matching message. If there is one, the message is delivered and the receive operation is completed. If there is none, the receive operation is canceled. In both cases, the function call returns without being blocked.

The probing function

```
int pvm_probe(int from, int tag)
```

does not create a receive operation, it only checks whether there is a message which matches `from` and `tag`. The function call returns without blocking. Note that if `pvm_probe` detects a matching message then a subsequent call to `pvm_recv` or `pvm_nrecv` (with the same parameters) completes the receive operation initiated by the call.

The functions `pvm_recv` or `pvm_nrecv` return an integer which identifies the buffer where the delivered message is stored on the completion of the receive

operation. Note that if a wildcard was used in a call to these functions, the application still does not know which process sent the message and with which tag. This information (the message header) can be obtained using the function

```
pvm_bufinfo(int bufid, int *bytes, int *tag, int *from)
```

This function obtains the buffer identifier `bufid` which is returned by `pvm_recv` or `pvm_nrecv` and returns information on the message header: the length of the message (in `bytes`), the message tag and the rank of the sender (in `from`).

After a message which matches `pvm_recv` or `pvm_nrecv` has been delivered, the buffer storing the message becomes the active message buffer. The data stored in the active message buffer can be disassembled using unpacking functions (there is one packing function for each basic type). For instance, this is a function that unpacks an integer (or an integer array) from the active receive buffer:

```
int pvm_upkint(int *integer, int count, int stride)
```

The application has the option to save the active message buffer and free it later. If it does not, the contents of the active message buffer is simply overwritten by a new message when a subsequent receive operation completes.

MPI

The MPI library does not automatically provide receive buffers. The application is responsible for the allocation of a sufficiently large buffer to the incoming message when initiating a receive operation. It is silently assumed that the application already knows the contents of the message (types of the data stored in the message) that can arrive before initiating a receive operation. To simplify matters, we assume that the arriving messages have been assembled using the packing method. The receiver can determine the size of the buffer using the `MPI_Pack_size` function which we described when talking about message packing.

The default receive function is synchronous:[5]

```
int MPI_Recv(void *buf, int count, MPI_Datatype t, int from,
   int tag, MPI_Comm com, MPI_Status *status)
```

initiates a receive operation and is then blocked until the receive operation is completed. `buf` is the receiving buffer where the message data are stored, `count` is the maximum number of data elements of type `t` that may be received and stored in `buf`, `t` is the type of the data elements. The `from` and `tag` parameters specify the messages which are allowed to be received. Wildcards can be used in both `from` and `tag`: `MPI_ANY_SOURCE` in `from` means the accepting of messages from all processes, `MPI_ANY_TAG` in `tag` means the accepting of all message tags. A return from an `MPI_Recv` call guarantees that a matching message has been delivered and the buffer `buf` can be unpacked or freed or reused by the application. The header of the message received is stored in `status`.

[5]MPI has no receiving counterpart to the synchronous send function `MPI_Ssend`. There is no receiving counterpart to the buffered send function `MPI_Bsend`, either.

The nonblocking receive function

```
int MPI_Irecv(void *buf, int count, MPI_Datatype t, int from,
   int tag, MPI_Comm com, MPI_Status *status, MPI_Request *req)
```

initiates a receive operation and immediately returns, without blocking and without canceling the receive operation. A return from the `MPI_Irecv()` call does not guarantee the completion of the receive operation. The meaning of all parameters is the same as with `MPI_Recv`. The additional `req` parameter is a handle of the receive operation. The handle can be passed to `MPI_Wait` or `MPI_Test` functions, which respectively wait for the completion of the receive operation (blocking) or test whether the receive operation is completed (nonblocking).

The blocking probe function

```
int MPI_Probe(int from, int tag, MPI_Comm com, MPI_Status *status)
```

blocks until a send operation that matches the parameters `from` and `tag` is found. The return from this function guarantees the completion of a subsequent receive operation with the same parameters. The MPI probe functions do not receive the message.

The nonblocking probe function

```
int MPI_Iprobe(int from, int tag, MPI_Comm com, int *flag,
   MPI_Status *status)
```

does not create a receive operation, it only checks whether there is a message which matches `from` and `tag`. The function call returns without being blocked and the information as to whether a matching message has been received is stored in `flag`. Note that if `MPI_Iprobe` detects a matching message then the completion of a subsequent receive operation with the same parameters is guaranteed.

2.5 Unifying framework for message passing

In this section we introduce a new framework for message passing. Our main intention is not to introduce another formal model but to unify the existing ones. Section 2.3 discussed the similarities and differences between the PVM and MPI standards but it is perhaps not yet obvious as to whether a program which uses PVM functions can also be written using MPI functions or vice versa.

Well-known models of parallel processing include PRAM (Parallel Random Access Machine) [FW78], ATM (Alternating Turing Machine) [MS87], Cellular Automata [vNe66] etc. Widely accepted models of message passing are CSP (Communicating Sequential Processes) by Hoare [Hoa85] and the "channel model" by Andrews [And91]. The formal message passing models as well as the real-life message passing standards are apparently different even though they all deal with communication between parallel processes. However, the models are equivalent in the sense that a program in one model can be simulated in any other model. We believe that the framework we propose covers all the existing

message passing models, formal ones as well as the real-life needs.

Another reason for the introduction of a unifying framework is that there is no formal model that we know of which defines message passing, which can be directly mapped onto contemporary networks and which reflects notions used in the definitions of real-life message passing systems such as PVM or MPI. For instance, the Hoare's original model lacks asynchronous communication which is used in all contemporary real-life message passing systems. The "channel model" by Andrews defines the semantics of synchronous and asynchronous communication but the underlying mechanism used for communication between two processes is a channel shared between the two processes. A channel is an abstract FIFO structure similar to a pipe in the UNIX operating system. A channel has a *capacity*—it stores messages sent by the sender process. The receiver process removes the messages from the channel. However, the wires in real-life computer networks do not have a capacity—either the receiver or the sender processes store messages, not the wire itself. Therefore the channel abstraction cannot be directly applied to real-life networks. The insertion of a third process between the sender and the receiver does not directly help because the question remains as to how the sender and the receiver can communicate with the third process.

Our framework covers fundamental message passing concepts: synchronous and asynchronous communication, buffering, flow control. The major novelty of the framework is a strict separation of the interface between a parallel application and the message passing system from the implementation of the message passing system. This allows us to define minimal semantical requirements for the implementation of a message passing system which are independent of the hardware, operating system, means of communication, programming language and other similar factors used in the implementation of the communication system.

From a software engineering point of view, our message passing framework defines an interface between application programmers and implementors of communication systems in terms of basic message passing operations. This is what existing message passing models also do but at the same time they bind the semantics of the operations to mechanisms used in the implementations of the message passing systems. This binding reduces the set of architectures onto which the models can be directly mapped. Our framework avoids such a binding.

The binding of operations to mechanisms also makes a comparison between message passing models which are based on different mechanisms difficult. For instance, a reasoning in a model bound to a shared memory communication is much different to a reasoning in a model bound to a channel communication. These two apparently different models describe the same concept even though the expression of, say, the proof of correctness of a program written in one model in terms of the other model is not obvious. This is where our model helps. An abstraction from a particular mechanism makes the reasoning valid for a whole class of models which adhere to the semantics defined in our framework.

Our framework for message passing systems is in many ways similar to the

well-known framework for database systems used by academic researchers as well as by the implementors of database systems. [BHG87], [BL93], [Bac98] It defines an interface between the database application and a database system. The interface consists of basic operations which work on database records: `read` and `write`, `insert` and `delete`. (The last two operations are often omitted in database textbooks that silently assume that the database is non-empty and its cardinality does not change.) The semantics of these basic operations is defined independently of the actual binding of the primitives to a database programming language and independently of the implementation of the operations in the database system. On the one hand, this allows the writing of database applications without any knowledge as to how these operations are implemented—this means, independently of other applications running in the system, independently of whether the database system is a centralised or a distributed one and independently of the hardware or the operating system. On the other hand, the clean interface definition gives rise to development of important abstract theories such as serialisability and recovery which help the implementors of database systems to optimise their systems while adhering to the semantics of the basic operations. This all holds for the message passing framework proposed in this paper, only the set of operations and their semantics are different to those defined in database systems. The database operations work on database records, whereas the message passing operations work on messages.

A small step towards a similar separation of message passing operations from their implementation can be observed in the definition of the Message Passing Interface, MPI. However, the semantics of MPI is described quite informally on more than 300 pages of text and notions used throughout the text are not consistently used. This often causes confusion between both application programmers and implementors of MPI. Particularly, we show that the MPI language binding does not include asynchronous communication even though the MPI standard claims to support it. The same holds for another standard, CORBA, Common Object Request Broker Architecture. [Gro98], [HV99] In Section 2.6.3, we present code fragments crucial to irregular applications which need asynchronous communication and which cannot be expressed in MPI or CORBA. The "asynchronous communication" defined in MPI and CORBA is not equivalent to the asynchronous communication defined in fundamental abstract models.

2.5.1 Components of the message passing framework

This section gives a formal definition of the message passing framework. Firstly we introduce the components used in the framework and their roles. The components and their relationships are depicted in Fig. 2.10.

- *Application process* is a component that needs to communicate with other similar components. The application process is typically a process in the

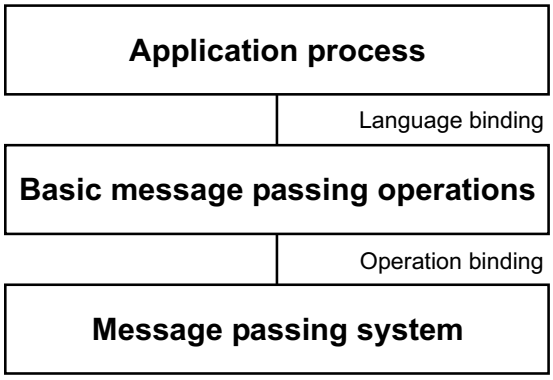

Figure 2.10: Components of the message passing framework

POSIX sense but this framework does not require that. It is not important whether the process is single- or multi-threaded, and in which programming language the process is written or the communication primitives which the process uses is also not important. What is important from the point of view of this framework is that the communication primitives of the application process can be expressed using the four basic message passing operations.

- *Basic message passing operations* are four abstract operations which are provided by the message passing system: `create`, `destroy`, `recv` and `send`. These operations are the interface between the application process and the message passing system.

- *Message passing system* is a system that implements the semantics of the basic message passing operations. The implementation does not need to be hardware or software. An abstract theory defining protocols which implement the basic operations on shared memory architectures can also be seen as a message passing system. A theory defining protocols which implement the basic operations on distributed memory system can also be seen as a message passing system.

In the database analogy, an application process corresponds to a transaction which passes basic database operations to a database system. The process can be written in an arbitrary programming language, for instance C or OCCAM (similarly, a database transaction can be written for instance in SQL or embedded C).

The language binding defines how constructions of the programming language which is used for the implementation of the application process translate into sequences of basic passing operations. The language binding can be implemented as a precompiler which generates function calls that generate the basic operations and pass them to the message passing system. The choice between *synchronous and asynchronous communication* falls into the competence of the lan-

guage binding—this choice does not influence the semantics of the basic message passing operations or the message passing system.

The set of the basic message passing operations is relatively small. We suggest the set that consists of only four operations which cover the fundamental needs of the exchange of information between processes. The four basic message passing operations have their counterparts in database systems as shown in Table 2.1.

Message passing	Databases
`recv`	`read`
`send`	`write`
`create`	`insert`
`destroy`	`delete`

Table 2.1: Basic message passing operations and their counterparts in database systems

The operation binding specifies how the basic message passing operations are mapped onto a specific architecture of the message passing system. A somewhat artificial example of a message passing system architecture is a single process which reads the basic operations from a file and interprets them. Another example of an architecture is a ring which directly connects application processes. In this case the message passing system must use an internal protocol which implements the routing of messages in the ring. Such a protocol is called a *mechanism* in our framework.

The message passing system processes the operations which arrive from application processes. The processing of the operations in the message passing system implicitly defines the semantics of the operations.

2.5.2 Application process

The application process passes basic message passing operations to the message passing system. Each application process has a unique identifier. The set of application processes can be either static or dynamic. We will consider the dynamic model in which the message passing system assigns the unique identifiers to processes on-the-fly as the processes sign on and off. A static model is only a special case of the dynamic model.

An application process does not need to be a single process in the POSIX sense. It can for example be a collection of POSIX processes or a single thread of control or even a group of people. However, from the message passing system's point of view one application process is regarded as one entity. The message passing system does not need to know what an application process does or what

it looks like. The system only obtains basic message passing operations generated by the application processes and performs them.

An application process can directly pass the basic message passing operations to the system. However, it can also use higher communication primitives (e.g. a barrier synchronisation instruction) from which the basic operations are generated. The message passing system does not see the higher communication primitives. We call the mapping from higher communication primitives to basic operations a language binding. Language binding does not affect the semantics of the basic message passing operations.

2.5.3 Basic message passing operations

The basic message passing operations are the interface between the application and a message passing system. The semantics of these operations is independent of the higher communication primitives or of the implementation of the system. Firstly we introduce some notions needed to informally explain the semantics of the basic operations. An important notion is the *scope* of an application process. Any object can only be manipulated by the application process in whose scope the object exists.

Application processes use *messages* as the only means of communication. The `create` operation creates a new message in the application process which issues the `create` operation. An application process can only access a message (read or write the contents of the message) that exists in its scope. Our framework does not specify how messages are represented or what they contain. The language primitives for reading or writing a message are not part of the interface between the application process and the message passing system. The `destroy` operation removes a message from the scope of the application process which issues the `destroy` operation.

There are two basic operations in relation to point-to-point message passing between two processes. The `send` operation creates a new *send request* in the scope of the application process which issues the `send` operation. A send request is a tuple $< S, x, y, M >$, where x is the identifier of the process which issues the `send` operation, y is the identifier of the destination process (y can be equal to x) and M is the identifier of a message which exists in the scope of the process which issues the `send` operation. The completion of the send request means that the message M has been removed from the scope of the process x and an exact copy of M has been created in the scope of the process y.

Similarly, the `recv` operation creates a new *receive request* in the scope of the application process which issues the `recv` operation. A receive request is a tuple $< R, x, y, \_ >$, where x is the identifier of the process which issues the `recv` operation, y is either an identifier of the source process (y can be equal to x) or $*$ ($*$ denotes *any* source process) and $\_$ means that no message is associated with

the receive request.[6] The completion of the receive request means that a new message has been created in the scope of the process x. The message identifier is returned upon the completion of the receive request. Right before the completion, the exact copy of the message exists in the scope of the source process y if $y \not\equiv *$, or in some source process if $y \equiv *$.

A request is an abstract entity which is used in the formal definition of `send` and `recv` operations. Some systems do not even have to explicitly represent requests—provided that the formal semantics of `recv` and `send` operations are correctly implemented.

Note that there are no operations removing requests from the system. They are not needed. The message passing system takes care of the removal of requests after their completion (if requests are explicitly represented in the implementation of the system).

2.5.4 Message passing system

Basic message passing operations are input of the message passing system. The message passing system processes the operations and looks for matching request pairs. The matching algorithm implicitly defines the semantics of the basic message passing operations.

Requests $R_1 \equiv< S, x_1, y_1, M >$ and $R_2 \equiv< R, x_2, y_2, \_ >$ are a matching pair iff $y_1 \equiv x_2$, and either $y_2 \equiv x_1$ or $y_2 \equiv *$.

When the system finds a matching request pair

$$< S, x_1, y_1, M >, < R, x_2, y_2, \_ >$$

it *completes the request pair*, by performing the following sequence of actions (the message M is passed from the process x_1 to the process y_1):

- New message M' is created in the scope of the process x_2. The contents of M' is identical to the contents of the message M.
- Message M is removed from the scope of the process x_1.
- Both the send and the receive requests are completed (removed from the system).

A request that has been created remains in the system until it is matched with some other request. The system guarantees that matching request pairs are not ignored forever (*weak progress*): *If there is a matching request pair at any one time then the system eventually finds and completes some matching pair.* This

[6] The semantics of the `send` and `recv` operation can be easily extended so that any set of destination and source process identifiers is stored in send and receive requests. We only present the one-destination and one-or-all-source semantics in order to simplify the notation.

guarantee can be strengthened (*strong progress*): *If there is a matching request pair at any one time then the system eventually completes at least one of the requests of this matching pair.*

The message passing system can work sequentially in discrete time steps. In each time step it either reads a new message passing operation or completes a matching request pair (if there is one). In this case the completion of request pairs must be prioritised over the reading of new operations in order to guarantee the strong progress. If the rate of incoming operations is higher than the rate of completed request pairs, some request pairs can remain forever in the system.[7]

The system can complete several request pairs at the same time (if the system works in discrete time steps, several request pairs can be completed in one time step). However, the effect of the parallel completion must be equivalent to the effect of some sequential completion.

2.5.5 Language binding

The application process does not need to know how to pass basic message passing operations to the message passing system. It does not even need to explicitly use the basic operations, it can use collective message passing operations, broadcasting and similar operations that are not included among the basic ones. The process can be even written in a non-imperative programming language, e.g. LISP or PROLOG. The language binding defines how sequences of the basic operations are generated from the higher-level programming constructs used by the application process and how the operations are passed to the message passing system.

Synchronous and asynchronous message passing constructs

A particularly interesting issue as regards the definition of higher-level languages for message passing is the support of synchronous and asynchronous message passing constructs. Note that the semantics of the basic passing operations is asynchronous in the sense that a completion of a request which is associated with a basic operation (see the Section 2.5.3 and Section 2.5.4) is independent of the intervention of the application process. Below we sketch what the message passing constructs might look like in an imperative programming language such as C.

An asynchronous send, `Isend` (the "I" stands for "immediate"), can be implemented as a single function call which in turn produces a basic `send` operation

[7] Note that the pending requests and messages associated with them consume memory. In order to keep the concepts clean, our framework assumes a potentially infinite amount of memory available in processes. An incorporation of a flow control mechanism (which bounds the amount of memory used by the pending requests) is possible in real-world implementations of the framework.

that is passed to the system. The application process does not need to know whether or when the request is completed. Note that this implies that the message passing system is responsible for freeing the message buffer after the completion of the request. The MPI standard [MPI94], [MPI98], [MPI97] specifies that the application is responsible for freeing the buffer, see Section 2.4. The consequence of this is polling in the MPI applications, as we show Section 2.6.3.

A synchronous send, `Ssend` (the "S" stands for "synchronous"), can be implemented as a single function call which produces a basic `send` operation that is passed to the system. This function call blocks until the request which is associated with the operation is completed. The application is not responsible for freeing the message buffer after the completion of the request. (The MPI standard states otherwise, see Section 2.4.)

An asynchronous receive, `Irecv`, can be implemented as a single function call which in turn produces a basic `recv` operation that is passed to the system. An asynchronous receive means that the application process is willing to receive a message in the future but does not want to wait for it now. It is the responsibility of the application to wait for the requested message when it needs it (the application is not able ask the message passing system whether or not the message has already arrived but the system can notify the application when the request has been completed). The message passing system is responsible for the allocation of memory for the incoming message, the application is responsible for freeing the memory when the contents of the message is no longer needed. (The MPI standard states that the application is responsible for the allocation of memory for the incoming message, see Section 2.4.)

A synchronous receive, `Recv`, can be implemented as a single function call which produces a basic `recv` operation that is passed to the system. The function call blocks until the request which is associated with the operation is completed. The message passing system is responsible for the allocation of memory for the incoming message, the application is responsible for freeing the memory when the contents of the message is no longer needed. (The MPI standard states that the application is responsible for the allocation of memory for the incoming message, see Section 2.4.)

Remark. The set of the four point-to-point functions can be reduced to two. The relevant functions are the blocking `Recv` and the nonblocking `Isend`. This choice is natural. A process (or a thread) wants to receive a message when it cannot proceed without the message. A process (or a thread) which sends a message does not usually need to know when or whether the recipient decided to receive the message. •

Remark. The message passing primitives defined in MPI cannot be mapped to our message passing framework. More precisely, it is impossible to extract the

basic message passing operations from the MPI functions so that their semantics would correspond to the semantics defined in our message passing framework. The reason is the wrong buffer allocation and deallocation policy defined in the MPI standard. Roughly expressed, *the MPI standard does not define message passing.*

The automatic buffering policy and the language primitives of PVM match our framework. However, the lack of thread-safety in the current PVM implementations imposes restrictions to the use of the language primitives (i.e. to the generation of sequences of message requests in a process). •

2.5.6 Operation binding

Operation binding specifies how the semantics of the basic message passing operations is implemented on a specific architecture of a message passing system. In other words, operation binding states how the system performs its steps, how it finds the matching request pairs and how it completes them. An architecture does not necessarily mean hardware. Examples of abstract architectures include processors connected to a ring or a torus with channels, processors which share memory, processors connected to an Ethernet network etc. A *mechanism* (*a protocol*) must be found for every architecture that at least simulates a sequential message passing system with the weak progress guarantee.

Examples of protocols which may be useful for the definition of operation binding for distributed memory architectures can be found in [Ray88]. The protocols include election algorithms for various topologies, decentralised deadlock detection, termination detection, distributed data management, fault tolerance algorithms etc.

Equivalence of our framework to Andrews' message passing model

The channel model by Andrews is described in [And91] in Chapter 7, "Asynchronous Message Passing". Synchronous message passing is defined in Andrews' book as asynchronous message passing with some additional constraints. We will not repeat the whole formal semantics of Andrews' model here (otherwise we would have to repeat the definition of the programming logic which is used throughout the book), we will only focus on the key concepts of the model.

Andrews' model is based on the concept of channels. A channel is basically a FIFO queue in which messages are stored. The queue can be extended by appending a message to the tail of the queue and shortened by removing a message from the head of the queue. The queue has a potentially unlimited capacity—that means, a message can be appended to it at any one time.

Andrews defines the semantics of `send` and `receive`:
"The effect of executing `send` $ch(expr_1, \ldots, expr_n)$ is to evaluate the expressions

$expr_1, \ldots, expr_n$, then append a message containing these values to the end of the queue associated with channel ch."
"The effect of executing `receive` $ch(var_1, \ldots, var_n)$ is to delay the receiver until there is at least one message on the channel's queue. Then the message at the front of the queue is removed, and its fields are assigned to the var_i. Thus, in contrast to `send`, `receive` is a *blocking* primitive since it might cause delay. The `receive` primitive has blocking semantics so that the receiving process does not have to busy-wait polling the channel if it has nothing else to do until a message arrives."

Andrews further defines a property which ensures that if a receiver is blocked on a channel and a message arrives on the channel, then the receiver will eventually consume the message.

Claim. If Andrews' model is restricted so that each channel is read by exactly one process (but several processes may write into the same channel), then the semantics of an abstract message passing system is equivalent to the semantics of the channel model defined by Andrews.

Proof. The only technical difficulty in the comparison of the two models is that Andrews' model silently assumes a dynamic allocation and deallocation in `send` and `receive` (this is the memory which stores the elements of the channel FIFO queue). In our model we assume an explicit dynamic allocation and deallocation of memory which stores the messages—this allocation and deallocation is carried out in `create` and `destroy` operations, separately from the `recv` and `store` operations.

We will first show that Andrews' channel model can simulate our model. Consider a program which contains message passing operations `create`, `destroy`, `recv`, `send`. Replace each `recv` in the program with `receive` $ch(vars)$ where ch is the only channel which is read by the `receive` ($vars$ denotes now the contents of the message which has been allocated by the corresponding `create` operation. The semantics of the `create` operation remains unchanged—this operation performs a dynamic allocation of memory for the message. Similarly, the semantics of the `destroy` operation remains unchanged—this operation performs a dynamic deallocation of the memory block which has been created by some `create` operation.) Replace each `send` in the program with `send` $ch(expr)$ where ch is the channel which is read by the recipient. It is easy to observe that the replacements above do not change the semantics of the program in Andrews' model.

We will show that our message passing model can simulate the (slightly restricted) Andrews' channel model. Consider a program which contains `send` and `receive` statements with the channel semantics. Replace each channel `send` $ch(exprs)$ statement with the sequence of two message passing operations `create`; `send`. The former creates a message which is large enough to store the values of $exprs$. The latter creates a send request which addresses this message to the

process which reads the channel *ch*. The program which issues these operations is further extended with the filling of the newly created message with *exprs* in the time period after it has issued the `create` operation and before it has issued the `send` operation. Replace each channel `receive` $ch(vars)$ statement with the sequence of two message passing operations `recv`; `destroy`. The first creates a receive request which matches all messages (a wildcard is used in this request). The second frees the memory used by the incoming message. The program which issues these operations is extended with the filling of its local variables with *vars* before it has issued the `destroy` operation. It is easy to observe that the eventual consumption of the message by the receiver is guaranteed by our model (by the progress rule). •

Remark. Note that the restriction in the Andrews' model (each channel is read by exactly one process) is only made in order to simplify the operation binding for distributed memory architectures. If we allow wildcards in `send` requests in our model then this restriction is not necessary. (Only the definition of the request matching must be extended in our model in this case. The rest of the model remains unchanged.) •

2.6 Threaded non-trivial PVM and MPI applications

2.6.1 Threads and thread-safety

The process model implemented in the Transputer is—as expressed in contemporary terms—a restricted concurrent thread model. An Occam program which is executed in a Transputer can be seen as a process that consists of several independent threads of control.

All modern operating systems support the concept of threads. A process is a program which can consist of several threads. [ISO90] The threads share the same memory space (except for the stack—each thread has its own stack) but each has its own flow of control. An "ordinary" sequential program runs as a process with one thread.

Definition. *Thread* is an encapsulation of the flow control in a program. •

Particular care must be taken when writing multi-threaded programs. There is only one running thread at any one time (on a single-processor system), but the running thread can be descheduled and replaced with some other thread anytime (unless the thread scheduling policy states otherwise). Any thread which is not

blocked at any one time can be scheduled to run. The interleaving of threads is equivalent to a quasi-parallel execution of the threads.

An explicit synchronisation is needed when threads share resources (e.g. variables).[8] There are many examples of so-called *racing conditions* (an unwanted phenomenon caused by an unforeseen order of the execution of running threads) in concurrent programming textbooks. [And91] A popular example is a linked list which is manipulated by two threads, whereby one thread adds an item to the list and the other thread removes an item. The linked list becomes corrupted when the two threads are interleaved in a certain way. Processors and operating systems provide mechanisms for the synchronisation of threads, such as mutexes and semaphores.

Several threads often need to call the same function. I/O functions such as `open`, `read`, `write` are very common examples. A problem may occur when a function needs to maintain a global state. For instance, the implementation of the `open` function may add the newly opened file to a global linked list. In such a case a racing condition can occur when a thread calls `open()` while another thread is manipulating the linked list in its `open()` call. A function is regarded as thread-safe when its implementation avoids racing conditions.

Definition. *Thread-safe function* (*reentrant function*) is a function that can be concurrently called from several threads, providing the same semantics to each individual thread. •

Thread-safe function can thus be called from several threads without *corrupting memory* (without destroying internal memory structures used by that function) and without *thread-interference* (the semantics of the function does not change from the point of view of the calling thread if at the same time the function is concurrently called from another thread).

Functions are usually collected in libraries. Even if each of the functions is thread-safe, a racing condition may occur when different functions of a library are concurrently called from different threads. A library is regarded as thread-safe if such a racing condition cannot occur.

Definition. *Thread-safe library* is a library in which all functions can be concurrently called from several threads without memory corruption and without thread-interference. •

Practice shows that the programming of thread-safe libraries is difficult. However, thread-safety is so important for many applications that vendors of operating systems invest the effort into making at least the low-level system libraries [ISO90] thread-safe, and carefully document those which remain thread-unsafe.

[8] Note that no synchronisation is needed when multiple threads only *read* a shared variable.

System calls which may eventually block deserve special attention—a blocked system call must only block the calling thread, not the entire process.

A natural scenario of the implementation of a non-trivial application (Section 2.1) involves running two threads in each process as shown in Fig. 2.11. The heavy computation is hidden in the function `compute()` of the thread T_1. We restrict the occasional communication of the thread T_1 to only sending messages to other processes (later it becomes clear why this restriction is not important). The thread T_2 services the incoming requests. The `recv` function is a blocking receive (`pvm_recv` in PVM, `MPI_Recv` in MPI). This means that if there are no requests to be serviced, the thread T_2 is blocked in the `recv()` call, leaving the full CPU power for the computation in T_1.

```
thread T1()                    thread T2()
{                              {
  while (not_done)               while (not_done)
  {                              {
    compute();                     recv();
    send();                        service_request();
  }                              }
}                              }
```

Figure 2.11: Natural threaded implementation of one process of a non-trivial application: it only works if the communication library is thread-safe

The natural scenario of Fig. 2.11 only works if the communication library is thread-safe (because of the concurrent `send` and `recv()` calls). We are aware of no thread-safe implementation of PVM. There are some MPI implementations that are thread-safe but internally use active polling, see Section 2.6.3—the only two exceptions we know about is the IBM implementation for IBM RS/6000 SP [Tre97] and MPI/Pro implementation by Software Technologies, Inc. [DS02]

Observation. *Most existing implementations of PVM and MPI are not thread-safe.* This means that (most) PVM/MPI functions can not be concurrently called from several threads. The natural scenario of Fig. 2.11 does not work for all these implementations. •

2.6.2 Polling in threaded non-trivial PVM and MPI applications

When the communication library is not thread-safe, thread T_1 in Fig. 2.11 must not call `send()` when the thread T_2 is inside the `recv()` call. The easiest way of

guaranteeing the mutual exclusion of calls to the communication library involves the protection of each call to the library with a mutex (this protection can also be hidden in the library). This approach is suggested in [HR03b], [HR03a] in order to overcome thread-unsafety of PVM or MPI implementations. It avoids racing conditions but it does not solve the problem of non-trivial (irregular) applications (although the cited works aim to become a framework for development of irregular applications). If the `recv()` and `send()` calls are mutually excluded and the thread T_2 is blocked in the `recv()` call then the `send()` call in the thread T_1 will block until a message from some other process arrives, which would unblock T_2. However, if no message arrives, then the whole process will remain blocked forever.

A general approach which guarantees the mutual exclusion of calls to a thread-unsafe communication library and at the same time a progress in a non-trivial application is *active polling* (also known as *busy waiting* or for short *polling*).

Fig. 2.12 shows a polling implementation of one process of a non-trivial application. The shared mutex `comm` is used for the protection of the communication library calls in T_1 and T_2. This mutex can only be locked (acquired) by one thread at any one time. A thread which attempts to lock an already locked mutex becomes blocked until the thread which holds the mutex unlocks it. The thread T_2, after having locked the mutex `comm`, calls a nonblocking `probe()` to check whether there is an incoming message. If there is a message, it is received in the `recv()` call and processed. If there is no message, T_2 unlocks the mutex and falls asleep for some time. The `sleep()` call before entering the loop is necessary in order to give some CPU time to the computation in T_1. Before falling asleep, T_2 unlocks the mutex in order to allow T_1 to call `send()`.

The pseudo-code of Fig. 2.12 can be expressed in different ways when using a (thread-unsafe) PVM or MPI library. Some of the alternatives include:

1. The *nonblocking* `recv` (`pvm_nrecv` in PVM, `MPI_Irecv` followed by `MPI_Test` in MPI) can be used in the thread T_2 instead of the nonblocking `probe` (`pvm_probe` in PVM, `MPI_Iprobe` in MPI) and the blocking `recv` (`pvm_recv` in PVM, `MPI_Recv` in MPI).

2. All actual communication with other processes can be moved to the thread T_2. The `send()` in the thread T_1 can be replaced with an insertion of the message to a send queue (together with a message header which stores the recipient's address, message tag, ...). Inside the polling loop, the thread T_2 checks whether the send queue is empty. If the queue is not empty, T_2 sends the messages in the send queue to the recipients.

3. If the structure of the function `compute` in the thread T_1 is simple and if the application itself does not require multiple threads, then the polling from the thread T_2 can be moved to the thread T_1. An example of a simple structure of the `compute` function is a loop which is repeated many times.

```
thread T1()
{
  while (not_done)
  {
    compute();
    lock(comm);
    send();
    unlock(comm);
  }
}
```

```
thread T2()
{
  while (not_done)
  {
    lock(comm);
    arrived=probe();
    if (arrived)
    {
      recv();
      service_request();
    }
    unlock(comm);
    sleep(time);
  }
}
```

Figure 2.12: Polling implementation of one process of a non-trivial application: a thread-safe communication library is not required for the application to work correctly—on the other hand, polling makes the application inefficient and non-portable

In this case the thread T_2 can be eliminated, which reduces the program to the single thread T_1 running the sequential loop (this loop contains a computation which is mixed with nonblocking `send()` calls). In addition, after every few executions of the loop a nonblocking `probe` is called to check for incoming requests. If there are requests, they are serviced—otherwise the computation continues. The problem with this solution is that it assumes a regular application structure. Moreover, the tuning of the number of loop executions after which the `probe` is executed is strongly machine-dependent.

The choice of the right alternative can improve the efficiency of a certain application on a certain system. However, when the application is ported onto a new system, the tuning procedure must be repeated in order to find the optimal polling parameters. Even worse than that, the optimal setting of the polling parameters can also depend on the inputs to the application, in which case an empirical tuning does not help.

Claim. *Every non-trivial application which builds on a thread-unsafe communication library is forced to use active polling (busy waiting).* •

2.6.3 Polling in communication libraries

When avoiding polling in an application, one must ensure that none of the libraries used by the application uses polling. If a polling library function is called from a thread, then all other application threads are slowed down by the polling thread.

Section 2.6 shows that the problem with non-trivial applications is a lack of thread-safety of libraries which are used for inter-process communication. To the best of our knowledge, there is no thread-safe implementation of the PVM library. The MPI-2 standard [MPI97] defines four levels of thread-safety (whereby the highest one, `MPI_THREAD_MULTIPLE` is needed for efficient non-trivial applications). Some MPI implementations are thread-safe (`MPI_THREAD_MULTIPLE`). However, the MPI developers did not keep in mind the reason *why* the thread-safety is important—they "solve" the problem using *active polling inside the library*! The original problem with non-trivial applications is thus only dug one level deeper. Such an approach can be found in MPICH2 [Gro02].

Claim. *If active polling is used to solve thread-safety problems inside a communication library, it is impossible for an application using the library to avoid active polling.* Even worse than this, the application does not even have the freedom of choice between the alternative polling implementations as described in Section 2.6.2—it is forced to use the one implemented in the library. •

A thread-safe communication library that does not internally use threads but uses polling instead must keep a table of pending `send` and `receive` requests. The polling thread-safe scenario may work as follows (similarly to the program in Fig. 2.12):

- A mutex is used inside the library to ensure a mutual exclusion of `send()` and `recv()` calls.
- Blocking `send()` and `recv()` calls do not internally block. The library polls on behalf of all pending requests in the implementation of each "blocking" `send` or `receive`. The global mutex is locked during the time needed to detect progress and it is unlocked during the `sleep()` call used in the polling loop.

Some research papers on communication libraries [Fer98], [KHS96] do not mention their implicit use of active polling, thus hiding the source of a significant performance loss.

Polling related to asynchronous communication

Another potential source of active polling in communication libraries involves using optimisation techniques for single-threaded communicating processes. We

will focus on the implementation of the asynchronous (nonblocking) `MPI_Isend` in MPICH [GL96]. As we will show, the implementation is *wrong* because it violates the progress rule defined in the MPI standard [MPI94], [MPI98], [MPI97]. An MPI program which demonstrates this weakness is given in Appendix A. We will also show that *even if the implementation of* `MPI_Isend` *was correct, the application using the nonblocking send would suffer from polling.*

An `MPI_Isend()` call returns immediately to the caller. The MPI tutorial book [GLS95] (Chapter 4, Section "Using Nonblocking Communication", page 81) says: "The buffer containing the message to be sent using `MPI_Isend` must not be modified until the message has been delivered (more precisely, until the operation is complete, as indicated by one of the `MPI_Wait` or `MPI_Test` routines)." (`MPI_Wait` is a blocking MPI function which blocks until the corresponding `MPI_Send` has been completed. `MPI_Test` is a nonblocking MPI function which returns immediately with the information on the completion of the corresponding `MPI_Send`.)

A question arises: Under what circumstances can an `MPI_Isend` complete *before* `MPI_Wait` or `MPI_Test` has been called? (If no such circumstances exist, then `MPI_Isend` loses its reason for existence.)

In order to complete an `MPI_Isend`, the sender's side must *push* the message to the network (this pushing corresponds to writing to a socket). The pushing itself can either be blocking (blocks until the recipient's side is ready to receive) or nonblocking (tests whether the recipient's side is ready to receive). The pushing must be hidden in the MPI library. However, `MPI_Isend` returns the control to the application after some initial nonblocking pushing at the latest.

If the message can not be delivered to the network during the initial pushing, the next opportunity to retry the pushing is when the MPI library regains control again. This happens when the application calls an MPI function. The pushing of pending `MPI_Isend` messages can be (and usually is) implemented as a side effect of most MPI functions. An eventual delivery of pending `MPI_Isend` messages is only guaranteed if the application either regularly calls MPI functions or the control in the application gets to the `MPI_Wait()` call paired with the `MPI_Isend`.

The hidden pushing of pending `MPI_Isend` messages which is implemented as a side-effect of any MPI function works well with single-threaded applications but causes problems when either the application is multi-threaded (not necessarily non-trivial in the sense of Section 2.1) or when the application's processes run on multi-user operating systems. Consider the following sequence of MPI calls in one process: `MPI_Isend()`; `MPI_Recv()`, and assume that the initial pushing during the `MPI_Isend()` was not successful (the receiver's side was not ready to receive the message at that time). The `MPI_Recv()` call blocks until a matching message arrives. But the arrival of the matching message may depend on the delivery of the pending `MPI_Isend()` message. For this reason the MPI library must not passively block in the `MPI_Recv()` call. Instead of this the implementation of `MPI_Recv()` must either block while waiting for *both* events (the pushing of the

pending outgoing message which has been sent using `MPI_Isend()` and the arrival of a message in `MPI_Recv()`) or the polling for the matching message must be interleaved with the pushing of the pending `MPI_Isend()` message. The second option—interleaved polling—can be found e.g. in MPICH [GL96]. The use of the interleaved polling may be advantageous for a single-threaded application or an application which runs on a single-user operating system. However, a multi-threaded application which uses the above mechanism in one of its threads will spin until a matching message arrives and the control returns from the `MPI_Recv` call. This spinning means that the thread which is inside the interleaved polling unnecessarily consumes up to 100% of the CPU time, blocking the other threads or other user's processes (if the process is running on a multi-user operating system) which may make better use of the wasted CPU time.

Also, the interleaved polling only guarantees progress when the control in the application process regularly reaches an MPI call. In a general case, no progress is guaranteed.

Claim. MPICH violates the progress rule defined in the MPI standard. In other words, MPICH does *not* comply with the MPI standard.

Proof. Consider a program that consists of two parallel processes, P_1 and P_2. The program performs the following actions (in this order):

1. P_1 calls `MPI_Isend()`, sending a message to P_2.
2. P_1 runs an infinite loop (followed by `MPI_Wait()`).
3. P_2 calls `MPI_Recv()` which matches the message sent by P_1.

The progress rule states that the message sent by P_1 will eventually be delivered once it is matched by the `MPI_Recv()` in P_2 (the MPI standard states that this must occur independently of the other actions in P_1). The message sent by P_1 will never be delivered to P_2 in MPICH. •

A common workaround involves the insertion of an `MPI_Test()` call (or some other MPI call) into the infinite loop in the process P_1. However, sometimes this workaround cannot be directly applied. For instance, the text "P_1 runs an infinite loop" in step 2 can be replaced with "P_1 makes an I/O call which can block". This causes application programmers to look for other workarounds, which generally destroy the natural organisation of the code.

Remark. The default `MPI_Send` can be safely implemented as a synchronous `MPI_Ssend`. However, MPICH does not implement `MPI_Send` as `MPI_Ssend`—`MPI_Send` in MPICH can return before the message has been delivered to the receiver. This means that the progress rule is also violated for the default `MPI_Send` in MPICH. •

The MPI standard requires the application to free the send buffer used in `MPI_Isend`. The application can free the buffer after it made sure that an `MPI_Isend()` call has completed. The functions `MPI_Wait` or `MPI_Test` can (or rather, must) be used to detect the completion of the `MPI_Isend`. This is another source of polling—this time in the application, not inside the MPI library:

Claim. The semantics of asynchronous (nonblocking) communication in MPI forces the application to use polling in order to avoid using an unbounded amount of memory.

Proof. Consider a program with an unpredictable flow of control. The program calls `MPI_Isend()` unpredictably many times. Each of these calls must be paired with an `MPI_Wait()` at some point in order to free the buffer used in `MPI_Isend()`. However, the unpredictability of the flow of control makes it impossible to place the `MPI_Wait()` anywhere but at the end of the program (right before `MPI_Finalize()`). This is undesirable because the buffers of *all* `MPI_Isend()` calls are not freed until the end of the program (even though they are no longer needed). •

A general workaround assumes running a separate thread which polls for completion on behalf all pending `MPI_Isend()` calls. Another workaround involves preceding each `MPI_Isend()` call with an `MPI_Wait()` call. However, this reduces the maximum number of pending `MPI_Isend()` calls to one, which can lead to unexpected deadlocks in applications which relies on a higher number of concurrent `MPI_Isend`s. A more general solution of this kind involves preceding each `MPI_Isend()` call with an `MPI_Testany()` call with the intention to control the amount of pending `MPI_Isend`s—but this is equivalent to polling.

The consequence is that an efficient use of the nonblocking communication defined in the MPI standard is restricted to the applications in which communication and computation phases alternate and are strictly separated. The communication in one phase can then overlap with the computation in the next phase. Such a model is studied in [LAB93].

The CORBA standard [Gro98] explicitly prescribes polling in applications that use asynchronous communication. This is a quotation from the tutorial book [OPR96] on using CORBA (Section 8.2.2, Deferred Synchronous Communication):

> "What CORBA refers to as the deferred synchronous communication style is a form what the non-CORBA world calls asynchronous communication. When an application invokes a deferred synchronous request, the application does not wait for the request to complete before it continues with other work. However, the application must

periodically check to see if the request has completed by polling using the `CORBA_Request_get_response` operation on the `CORBA_get_next_response` routine.

The deferred synchronous style of communication is most appropriate when you do not want your application to have to wait for the current request to complete before sending the next request. For example, if you do not know how long the operation may take, you may not want your application to wait for the request to complete. This will almost always be the case when you write a CORBA client to connect to someone else's framework or server."

2.6.4 Limits of active polling

This section explains why active polling diminishes the performance of non-trivial parallel applications and to what extent. Consider the polling implementation of a non-trivial application (Fig. 2.12, Section 2.6.2). The only parameter which allows the tuning of the implementation is the `time` argument of the `sleep(time)` call in the thread T_2. The setting of `time` determines the tradeoff between the latency of the servicing of incoming requests and the wasted CPU cycles:

- If `time` is a short time period, say, 1 millisecond, then the latency of request servicing is 0.5 millisecond on average. On the other hand, CPU cycles are unnecessarily wasted every 1 millisecond if there are no requests to be serviced. The computation in the thread T_1 is slowed down.
- If `time` is a long time period, say, 1 second, then the CPU cycles are wasted only once a second. However, the average latency of request servicing is 0.5 seconds.

The setting of the `time` constant is dependent on many factors: application's computation/communication ratio, input data, operating system, network speed, network latency, buffering scheme in the communication library, ... The `time` constant must therefore be experimentally tuned. The need for tuning makes the polling approach non-portable, but a closer look at the implementation of `sleep` in operating systems reveals an even a more serious polling deficiency.

The POSIX standard [ISO90] defines a high resolution version of the `sleep` system call, `nanosleep(time)`, which is supported by all contemporary operating systems. `nanosleep(time)` puts the caller asleep for the specified `time` whereby `time` is given in nanoseconds. A typical implementation of `nanosleep(time)` in the kernel of an operating system is shown in Fig. 2.13.

Sleeping for a `time` shorter than the threshold of the kernel does not cause the descheduling of the calling thread—the calling thread is "actively" sleeping. In other words, other threads or processes do not get scheduled while the current thread is sleeping (in fact, the current thread is burning CPU cycles). For

```
nanosleep(time)
{
  if (time < VERY_SHORT_TIME_PERIOD)
    ...run idle CPU cycles for the given time...
  else
    ...set an alarm and deschedule the current process/thread...
}
```

Figure 2.13: Implementation of `nanosleep` in the kernel of an operating system

this reason `time` which is shorter than the threshold must be avoided in polling implementations of non-trivial applications (Fig. 2.12).

Sleeping for a time longer than the threshold causes the calling thread to be descheduled and rescheduled again after `time` has elapsed. However, it turns out that without a special tuning of the kernel (which leads to many unwanted consequences) *the minimal duration of* `nanosleep(time)` *is 0.02 seconds* in all systems available to us (Solaris, Linux, Ultrix). We measured this value by calling `nanosleep(time)` many times in a loop. (Alternative system calls such as `usleep` or `select` behave similarly. This behaviour is caused by the clock granularity in the operating systems which is set to 10 ms. As the clock ticks are discrete, this value is often increased with additional 10 ms.[9]) In other words, `nanosleep(time)` can be called at most 50 times per second. This has a worrying implication for all non-trivial applications using the polling scheme of Fig. 2.12:

Claim. The upper bound on the number of serviced requests in a polling process of a non-trivial application of Fig. 2.12 is 50 per second. This number does not depend on the speed of the processors or the network connecting them. •

Remark. The loss of performance is not the only negative consequence of polling. Another negative consequence (sometimes even worse than performance loss) is a *non-determinism*. A request that needs attention can be created at any time during the `sleep()` call of the thread T_2 (see Fig. 2.12). The process which sent the request usually waits for a reply—in other words, it is idle. The length of its idling interval depends on the length of time the recipient's thread T_2 will sleep from the moment at which the request was created. As the need for a request is randomly created, the request servicing times are random. The expected servicing time is a half of the polling time interval (0.01 seconds or more, as we have shown above). If a process of a communication-intensive non-trivial application sends

[9]We experimentally found that the `select` system call adds the additional 10 ms penalty least frequently on Linux 2.4.20-xfs i686.

1000000 requests, the expected polling overhead is 10000 seconds (which add to the parallel application time). However, the *actually measured* polling overhead of one run can theoretically be anything between 0 seconds and 20000 seconds (ca. 5 hours). •

2.6.5 Previous work related to thread-safety of PVM and MPI

Thread-safety of PVM and MPI has been studied for many years but the problem remains unsolved. We know of none implementation of thread-safe PVM and only about two successful implementations of thread-safe MPI which do not use polling (an inofficial IBM implementation and the implementation by MPI Software Technology, Inc.). This section presents a few key research works in the field.

PVM

The *LPVM* (Lightweight-process PVM) system was introduced in [ZG98]. LPVM is designed for shared-memory machines. PVM tasks are implemented as threads in LPVM. The authors recognise two main issues that have to be dealt with in order to make PVM multi-thread-safe: *global state and reentrancy*. LPVM removes global states from the PVM library by assigning receive and send buffers (and other resources) to each task. The user interface of PVM 3.3 is only slightly modified but a major redesign of `libpvm` is needed. Our implementation is simple and does not require the removal of global states.

TPVM [FS98] takes a different approach which assumes a thread to be the basic unit of parallelism in a distributed system. There is a thread server which registers all threads running in the system. This fine-grained model is mapped onto the coarse-grained process model of PVM for the purpose of message passing. Rather than going into technical details, we will explain the problem of TPVM from the point of view of a non-trivial application. There is a global message queue accessible to all threads in each process. A thread which wants to receive a message follows the following protocol. Firstly it looks for the message in the global message queue. If the message is there, the thread continues; if not, the thread attempts to receive a message from another TPVM task using a nonblocking receive. If a message is there, but it is addressed to another thread, the thread stores the message in the global message queue and retries the nonblocking receive (and later wakes up threads which are waiting for those messages); otherwise it falls asleep. The following is the weakness of the protocol: When a thread falls asleep, then **another** thread must attempt to receive a message in order to wake up the sleeping thread. If no such thread exists, *the sleeping thread*

will sleep forever—even though there may be a message addressed to the sleeping thread which was sent by some other TPVM process. This situation can be resolved by running a special thread that regularly polls for incoming messages and wakes other threads up. Our mechanism does not require the running of a polling thread.

A misleading attempt by PVM developers to support non-trivial applications was the introduction of message handlers in the version PVM 3.4. A function
`pvm_addmhf(int src, int tag, int ctx, int (*func)(int mid))`
was added to the PVM library. This function registers a user's message handler `func`. The handler is fired (the function `func` is called) when a matching message arrives (the message header must match the parameters `src`, `tag` and `ctx`). An elegant implementation of a non-trivial application would involve registering a set of message handlers instead of running the thread T_2 of Fig. 2.11. The program would remain single-threaded, executing only the thread T_1. However, the manual to `pvm_addmhf` says (note the marked text): "... `pvm_addmhf` specifies a function that will be called *whenever libpvm copies in a message* whose header fields of `src`, `tag` and `ctx` match those provided to `pvm_addmhf`." In other words, *the message handlers are only called when the application requests it*—a thread which regularly calls e.g. `pvm_probe()` must be running, which forces *pvmlib* to copy in the arriving message. This is equivalent to active polling and contrasts to what is stated in [GKPS97]: "...these message handlers are invoked internally without any user intervention."

PVM allows the delivery of signals between tasks. Signals, in combination with message handlers (see previous paragraph) may be used to get rid of the polling thread in a non-trivial application. A non-trivial application would be implemented as a single-threaded application which executes the code of T_1 of Fig. 2.11. The thread T_2 of Fig. 2.11 would be implemented as a set of message handlers. The message passing protocol would be extended as follows (the actual implementation can be very complex):

1. Process A sends a message to process B
2. Meanwhile, process B runs T_1, without taking the incoming message into account.
3. Process A sends a signal to process B, saying "Get up, you have a message!"
4. Process B receives the signal and fires a signal handler. The signal handler calls `pvm_probe` which fires a message handler which receives the message and performs an appropriate user's action `handle_message`.

A similar idea was used in [SKH96] for the implementation of active messaging in PVM. No polling occurs in the above scenario. However, technical issues make this approach *inefficient and non-portable* and restrict its use to *homogeneous parallel machines*.

MPI

There are only two existing implementations of MPI which are thread-safe and do not use polling:

- An experimental (inofficial) implementation by IBM [Tre97] was developed for IBM RS/6000 SP machines and has never been ported on other systems. The official IBM implementation of MPI uses polling to solve the problem of thread-safety.

- MPI/Pro, a commercial MPI implementation by MPI Software Technology, Inc. (MSTI), which claims to be fully thread-safe. [DS02] It also claims to implement asynchronous communication without polling. However, the product description states that MPI/Pro implements the MPI 2.0 standard [MPI97], therefore it is unclear how the problem of completion described in Section 2.6.3 is tackled in MPI/Pro.

MiMPI [GCC99] claims to be fully thread-safe but the authors do not explain how they solve this problem in their paper. According to our brief personal communication with the authors, MiMPI uses a socket pair between all pairs of processes and avoids sharing any other resources through threads. This socket replication only makes this approach scalable up to a certain number of processes because of the memory and other system limitations of processors.

ScaMPI [HOB+99] is a commercial implementation of MPI by Scali. ScaMPI uses polling to implement thread-safety.

A multi-threaded implementation of MPI is proposed in [PS98]. This proposal has never been implemented. The authors address efficiency issues in this paper (note the marked text): "... It should be mentioned that switching between threads and using synchronisation primitives also incurs a finite overhead. *Hence, communication benchmark results obtained with a multi-threaded communication software will probably be not as good as the results of a single-threaded implementation that burns CPU cycles in busy waiting for incoming data and delivers low communication latency.* However, this latency can be hidden with the overlap of communication and computation, so in a real-life situation, well-designed applications that use multi-threaded communication software will reveal better overall performance than their single-threaded counterparts even though the communication benchmarks show the opposite." The reasoning behind the marked text is wrong. In fact just the opposite is true for many important applications (all non-trivial applications)—*a very high communication latency is the main flaw of the polling approach*!

Some works assume a so-called *active device* support such as hardware interrupts triggered by a message arrival. [BMR02], [GRS97], [LRBB96] However, this assumption makes the approach non-portable to other architectures which do not support active devices. Moreover, even when active devices are available, the

coupling of the devices, operating system and the communication library remains unclear.

Another concept of thread-safe message passing using P4 and MPI is proposed in [CSD94]. The authors identify non-reentrant functions in P4 and sketch a threaded implementation of MPI which may lead to the solving of the problem of thread-safety of MPI. These ideas have never been implemented.

A possible impact of threads on new generation MPI implementations is studied in [HSS+98]. The MPI implementations based on polling (such as MPICH) are referred to as "first generation" implementations, the implementations based on threads are referred to as "second generation" implementations. Arguments are given which support both the polling and threaded approaches. An important issue throughout the paper is the interpretation of the progress rule, see Section 2.4. The authors recognise two interpretations, a "liberal" one (progress depends on whether the application periodically calls certain MPI functions) and a "strict" one (progress does not depend of actions in the system). In our opinion, the "liberal" interpretation is simply wrong. Also statements such as "Perhaps the strongest argument for polling is that it minimizes latency." are very disputable because they may only apply to a certain class of MPI programs (the trivial ones).

2.6.6 Quasi-thread-safe PVM and MPI

It is not particularly frustrating that PVM and MPI implementations are not thread-safe. What is frustrating is that non-trivial applications cannot be efficiently implemented using these libraries. This section describes a mechanism of an *interruptable blocking* `recv` which is missing in the libraries [Pla02b]. This mechanism does not make the libraries thread-safe (see Section 2.6.1) but it allows an efficient and portable implementation of non-trivial applications. We concentrate on a socket-based inter-process communication in the sequel but the mechanism can also be applied to the shared memory implementation of message passing (or even to a mix of shared memory and socket communicators).

The reasons why the natural threaded implementation of a non-trivial application of Fig. 2.11 cannot be implemented without active polling are:

1. Tasks T_1 and T_2 must be implemented as threads.

2. The thread T_2 must run a *blocking* `recv`.

3. The thread T_1 cannot `send` any messages while the thread T_2 is blocked in the blocking `recv` because the communication library (PVM or MPI) is not thread-safe (the `recv` and `send` functions cannot be concurrently called from two threads).

4. PVM and MPI are not thread-safe because it seems to be very difficult from a software engineering point of view to implement all their interface functions in a reentrant and portable way without active polling.

The mechanism of the interruptable blocking `recv` addresses point 3. In terms of concurrent programming, the thread T_2 is blocked in the critical section of the blocking `recv` when at the same *time*T_1 needs to call the `send` function.

The reason why the concurrent `send()` call leads to problems strongly depends on the implementation of the communication library. For instance:

- PVM 3.4 uses a single buffer for incoming and outgoing messages. The buffer state is manipulated by both `send` and `recv` functions. (More precisely, the buffer state is manipulated by the `mxfer` function which is used for both sending and receiving messages. The `mxfer` function is called from both `send` and `recv` functions.) The `send()` call in the thread T_1 destroys the *buffer state* which was set up for the thread T_2 executing the `recv()` call.

- MPICH 1.2.4, driver ch_p4 uses distinct buffers for the `send()` and `recv()` calls. The memory corruption results from the implementation of the *internal communication protocols* used by the ch_p4 driver. For instance, the implementation of `send` involves a bidirectional communication for the so-called rendez-vous protocol. The rendez-vous protocol is used for transfer of large messages which are sent in chunks, not as a whole. In the rendez-vous protocol the recipient sends an acknowledging message to the sender after having received a chunk, telling the sender "I am ready to receive another chunk". These acknowledging messages either get mixed with the regular messages which are being awaited by the thread T_2 executing the blocking `recv` (which puts them into the unexpected queue and these never reach the thread T_1) or they are received (and processed) by *both* threads T_1 and T_2 which also leads to an inconsistent state of the library. Upon an arrival of a message, the MPI library is not able to decide which thread is supposed to process the message—the MPI library does not even know that there are several threads.

Interruptable blocking receive

Implementations of the PVM and MPI libraries are very different (the implementations of the MPI standard itself are very different). Nevertheless, a more detailed study of the low-level implementations of the `recv` function reveals a certain regularity. On the very lowest level of the implementation of a blocking `recv` there is a `select() call`.[10] This is a system call defined by the POSIX

[10] A polling loop can be used instead of the `select()` call. However, such an implementation would be poor for the reasons discussed in Section 2.6.3 and Section 2.6.4.

standard [ISO90] which waits on a number of file descriptors to change status.[11] It is important to note that `select` can be (and is) efficiently implemented in kernels of operating systems—the calling process or thread sleeps until an event occurs, therefore not consuming any CPU cycles. The file descriptors in the case of the blocking `recv` are reading file descriptors bound to the sockets on which a message may arrive. Fig. 2.14 shows the situation.

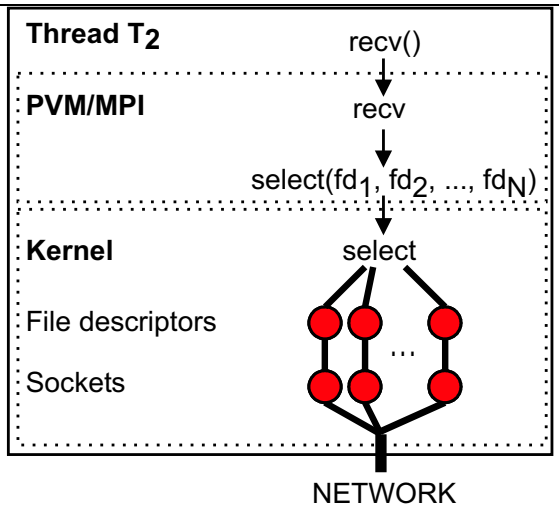

Figure 2.14: Implementation of the blocking `recv` in a socket-based communication library

The thread T_1 wants to send a message while the thread T_2 is blocked in the blocking `recv` (that means, inside a `select()` system call which is transparent to the user). T_2 remains blocked until a message from some other process arrives. T_1 cannot wait until the `recv()` call in T_2 unblocked (not for efficiency reasons—the arrival of a message at T_2 may depend on sending the message from T_1). *The basic idea of the interruptable blocking* `recv` *involves sending a "fake message" from the thread* T_1 *to the thread* T_2 *whenever* T_1 *needs to send a message to some other process.* The function of this "fake message" is to interrupt the `recv` in T_2 for the amount of time needed for T_1 to send the message to the other process. The interrupt mechanism must ensure that it is safe for T_1 to make the `send()` call during the interrupt.

However, we assumed that the communication library is not thread-safe—it does not allow a thread to send a message while another thread is blocked in the `recv`. How does T_1 send the "fake message" to T_2? In order to solve this problem, we exploit the fact that the communication between the threads T_1 and

[11] The `select()` call is similar to the Occam's `ALT` constructor (see Section 2.2.1). The functions `select()` and `poll()` are the only POSIX system calls which implement an efficient (non-polling) multiplexing on several file descriptors. (The name of the `poll()` function is a little misleading—`poll()` and `select()` are almost identical in their semantics and efficiency).

T_2 takes place on one processor, more precisely, in the scope of one operating system's kernel. The "fake message" bypasses the network and directly fires the `select` in the kernel, making the thread T_2 think that it has received a message from the network.

If the thread T_1 writes into a file descriptor which is being used for the purpose of communication with some other process, it must emulate the message passing protocol used by the communication library in order for the message to be correctly received by T_2. Furthermore, a message can arrive from the network on the file descriptor which is being written by T_1. In order to avoid these problems, *the set of reading file descriptors in the* `select` *in* T_2 *is extended with a special file descriptor,* `intr_fd`*. The* `intr_fd` *is the reading end of a POSIX pipe. The thread* T_1 *writes to the writing end of the pipe,* `intr_wfd`*, in order to interrupt the blocking receive in* T_2*.* There is no need to emulate the message passing protocol on the pipe because `intr_fd` is exclusively used for the purpose of interrupting the thread T_2. The only information T_2 needs is that it has been interrupted. *The code of the blocking* `recv` *is therefore extended so that it can detect which of the file descriptors has been fired—if* `intr_fd` *has been fired,* T_2 *knows that it is being interrupted by some other thread.* Fig. 2.15 depicts the scenario.

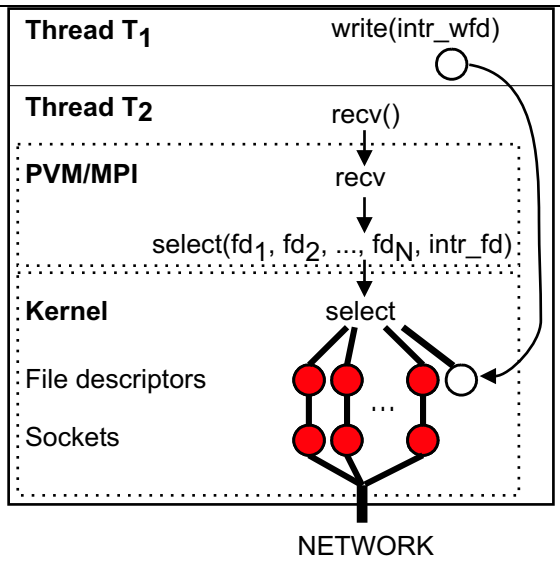

Figure 2.15: Scenario of the interruption of a blocked `recv`. The `intr_fd` file descriptor is the reading end of a POSIX pipe. The thread T_1 writes to the writing end of the pipe (`intr_wfd`), firing the blocked `select` in T_2

Encapsulation of the interruptable blocking receive mechanism

There are several ways as to how the interruptable blocking receive can be integrated in the implementations of PVM and MPI. The cleanest way would involve

hiding the mechanism in the communication library in order to make the library completely thread-safe. However, a performance-optimised implementation of complete thread-safety would probably require a complete rewrite of some library modules. This would probably be an extensive uncreative work. We chose another way. We extended the interfaces of the PVM and MPI libraries with two new functions. The following are the C interface and semantics of the new functions:

`void interrupt_recv(void)`
Blocks until the thread which had been blocked in the critical `recv`'s section has been blocked outside of the critical `recv`'s section. Following this the `interrupt_recv()` call returns. (If there is no thread blocked in the critical `recv`'s section, the `interrupt_recv()` call never returns.)

`void resume_recv(void)`
Makes the thread which was blocked outside of the critical `recv`'s section reenter the critical `recv`'s section and block there. After this `resume_recv()` returns. (If there is no thread blocked outside of the critical `recv`'s section, the `resume_recv()` call never returns.)

The sole addition of these new functions does not of course make the PVM or MPI library thread-safe—a random sequence of concurrent library calls from a multi-threaded application results in an undefined behaviour of the application. However, it is safe for a multi-threaded application to concurrently call some of the library functions in a certain defined order. Most importantly, a sequence of concurrent calls which are needed for a polling-less implementation of non-trivial applications is safe.

Definition. *A library is called thread-safe on a set of sequences of concurrent calls* when none of the sequences of the set leads to memory corruption or to thread interference. We call such sequences *safe*.

A communication library is called quasi-thread-safe when a safe sequence of concurrent calls which allows a thread to send a message exists while another thread is blocked in a blocking `recv()`. •

Claim. Both PVM and MPI libraries extended with the `interrupt_recv` and `resume_recv` functions are quasi-thread-safe.

Proof. The sequences of concurrent calls to a communication library that must be safe in order to allow a thread T_1 to send a message while a thread T_2 is blocked in a blocking `recv()` are:
{T_2: `recv()`; T_1: `interrupt_recv()`; T_1: `send()`; T_1: `resume_recv()`}

{T_1: `interrupt_recv()`; T_2: `recv()`; T_1: `send()`; T_1: `resume_recv()`}

A process of a non-trivial application which uses a quasi-thread-safe communication library (thread-safe on the above set of sequences) is depicted in Fig. 2.16. It is obvious that the program only produces safe sequences of calls to the communication library. •

```
thread T1()                          thread T2()
{                                    {
  while (not_done)                     while (not_done)
  {                                    {
    compute();                           recv();
    interrupt_recv();                    service_request();
    send();                            }
    resume_recv();                   }
  }
}
```

Figure 2.16: Threaded event-driven implementation of one process of a non-trivial application using a quasi-thread-safe communication library. This program does not contain any polling

Remark. A technical detail is the termination of the process in Fig. 2.16 (the setting of the `not_done` conditions in the `while` loops). In order for the process in Fig. 2.16 to work correctly, it is important that there is a matching `recv()` call in T_2 to each `send()` call in T_1 (otherwise the thread T_1 would block forever in `interrupt_recv()` preceding the `send()` call). On the other hand, a correct termination requires than the number of `recv()` calls in T_2 is *equal* to the number of `send()` calls in T_1 (in other words, the number of the executions of the `while` loop in T_1 is equal to the number of the executions of the `while` loop in T_2). One way of guaranteeing this involves sending a special finalising message from T_1 to T_2 (the process addresses the finalising message to itself using the usual sequence {`interrupt_recv(); send(); resume_recv()`} in the thread T_1) when T_1 is sure that it will not send any more messages. •

Implementation of the interruptable blocking receive mechanism in a (thread-unsafe) communication library

The implementation of the two newly introduced functions `interrupt_recv` and `resume_recv` requires changes to the original implementation of a (thread-unsafe) communication library. However, the extent of these changes is not very large.

Only the implementation of the blocking `recv` is affected (in the case of PVM only minor changes in *pvmlib* are needed—the *pvmd* remains unchanged). Fig. 2.17 shows the inner workings of the communication library functions. We use a *synchronous* POSIX pipe [`intr_wfd`, `intr_fd`] for the orchestration inside the communication library. This means that a write to the writing end of the pipe `intr_wfd` becomes blocked until the reading end `intr_fd` has been read; and vice versa, a read from `intr_fd` becomes blocked until some data has been written to `intr_wfd`. A synchronous pipe is not the only synchronisation mechanism that can be used in this scenario (it is also perhaps not the most efficient one)—but it is portable (pipes are defined in the POSIX standard [ISO90]) and it can later be replaced with any equivalent mechanism.

A look at Fig. 2.16 and Fig. 2.17 reveals the idea behind the implementation described in the previous paragraph. The thread T_2 eventually becomes blocked in the `recv()` call. The blocking is caused by the `select()` system call in `recv`. Now the thread T_1 needs to call a `send()`. It cannot do so immediately because the thread T_2 is blocked in a critical section of `recv`. T_1 therefore calls `interrupt_recv()` first which in turn writes to the pipe whose reading end is connected to one of the file descriptors in T_2's `select()`. The `select()` in T_2 is fired. (Note that T_1 is blocked at this moment in the `write()` and remains blocked until T_2 reads the `intr_fd`.) The thread T_2 takes over, bails out of the critical section of the `recv`[12] and reads `intr_fd` which unblocks the `write()` in T_1 (T_1 can now safely call `send()`). Then T_2 becomes blocked in the second `read()`. After T_1 has returned from its `send()` call, it calls `resume_recv()` which writes to `intr_wfd`. This unblocks the second `read()` in T_2 which reenters the `select()` in the critical `recv`'s section. T_1 returns from `resume_recv()` to the application code.

2.6.7 Towards a complete thread-safety of PVM and MPI

The mechanism of the interruptable blocking receive can make any communication library completely thread-safe without any loss of efficiency caused by active polling. The communication library should be structured as follows:

- Each process runs a thread (let us call it *main thread*) which is exclusively used for receiving all messages addressed to the process. When a message arrives, it is stored by the main thread in a global message queue. The main thread is automatically started during the initialisation of the library.
- Access to the message queue is protected by a mutex.

[12] The implementation of "bailing out" of the critical `recv`'s section may be tricky. The code shown in Fig. 2.17 is only an abstraction of actual solutions. At the time of writing we have solutions for socket-based PVM 3.4 and MPI 1.2.4 (driver ch_p4).

```
interrupt_recv()
{
  write(intr_wfd);
}

resume_recv()
{
  write(intr_wfd);
}

recv()
{
  do
  {
    /* BEGINNING OF CRITICAL SECTION */
    ...original recv code preceding select() is inserted here...
    select(original set of file descriptors, intr_fd);
    if (some of the original file descriptors was fired)
    {
      interrupted = FALSE;
      ...original recv code following select() is inserted here...
    }
    else
    {
      interrupted = TRUE;
    }
    /* END OF CRITICAL SECTION */
    if (interrupted)
    {
      read(intr_fd);
      read(intr_fd);
    }
  } while (interrupted);
}
```

Figure 2.17: Implementation of the interrupt mechanism inside a communication library. `intr_fd` is the reading end of a synchronous pipe, `intr_wfd` is the writing end of the pipe

- All `send()` calls are mutually excluded using a mutex. (The acquiring and releasing of the mutex are hidden in the implementation of `send`.) The `send` function calls `interrupt_recv()` at the beginning and `resume_recv()` at the end.
- The user's code runs as a thread (or several threads if the user's code is multi-threaded).
- The blocking `recv` function (as well as all other functions accessing the network) is implemented in such a way that it only accesses the global message queue, not the network. If there is no matching message in the queue, the recv function becomes blocked on a conditional variable. The main thread is responsible for waking up a blocked thread when a message for the blocked thread arrives.
- Thread addressing should not be part of the communication library's interface. This means that messages can only be addressed to processes, and not to threads inside a process. It is up to the user to develop a thread addressing scheme if it is needed in the application.

There are two main objections against implementing the above scenario inside communication libraries such as PVM or MPI. [HSS+98] The first objection is that developers of the communication libraries generally try to avoid using threads inside their libraries. This objection is not fully justified because most contemporary operating systems do support threads. The porting of communication libraries onto systems which are not POSIX compliant is more a political rather than a technical issue. (Moreover, there may be two versions of a communication library contained in a distribution—one for systems which support threads and another version for systems which do not.)

The second objection is related to the efficiency of existing single-threaded applications (which are not non-trivial in the context of Section 2.6). The latency of a `recv` in such applications may increase in the above scenario. The reception of a message in the application involves a thread switching between the main thread and the application thread. If the implementation of the thread switching is slow on certain systems then this additional overhead cannot be neglected. However, there is also an argument which supports the use of the proposed scenario even for single-threaded applications—the message matching the `recv()` call may already be available in the message queue *before* the call to `recv()` (if the other process has already issued the corresponding `send()` call and the message has been received by the main thread). This latency hiding can compensate for the overhead incurred by the thread switching.

Even if the second objection were justified, a communication library should also support non-trivial applications. If the use of threads inside the communication library is not feasible (the first objection), users of the library must be

given the possibility to implement non-trivial applications without polling. The extension of the libraries' interfaces (for instance as described in Section 2.6.6) is the least intrusive way of achieving quasi-thread-safety.

The next section shows how the above scenario can be implemented *on the top of (outside of) quasi-thread-safe PVM and MPI*—in a library which is between PVM/MPI and the user's application. We call this library TPL, Thread Parallel Library.

2.7 TPL: Event-Driven Thread Parallel Library

TPL is a communication library which provides an application programmer with functions which are needed for efficient programming of non-trivial parallel applications (as usual, distributed memory and message passing are assumed). *TPL is thread-safe* (more precisely, *thread-safe on the set of function sequences that perform message passing*). The ideas described in Section 2.6.6 and Section 2.6.7 are implemented in TPL in a way which is transparent to the user. An application which builds on TPL can be multi-threaded (a single-threaded application is just a special case of a multi-threaded one) and communication functions can be safely called from multiple threads. Threads can be dynamically created and destroyed in each process of the application.

TPL is not only another communication library, *TPL is a rigorous implementation of the message passing framework introduced in Section 2.5 on systems which are based on commodity hardware and system software.* TPL efficiently implements active messages and message handlers.

Unlike PVM or MPI, TPL tries to be minimalistic. There are two good reasons for keeping the design minimal (in this order of importance): *portability* and *efficiency*.

There are two degrees of portability. The first degree is an adherence to the POSIX standard [ISO90] in the implementation of the TPL library. (An essential part of the POSIX standard which the library depends upon is the concept of threads.) The second degree is the portability of TPL applications. PVM and MPI are two message passing standards available today. A mapping onto both PVM and MPI is implemented in the TPL library. A positive consequence is that *an application written in TPL can be linked and run with either PVM or MPI without changing a single line of the application's code.*

The minimalistic design of TPL simplifies the tuning of the library for a specific system. We did not invest much effort into the tuning. Our main goal was to prove that the event-driven mechanism outperforms the polling one.

Claim. To the best of our knowledge, *TPL is the first thread-safe communication library which is portable and at the same time allows the efficient implementation of non-trivial parallel applications (without active polling in the application or in*

the library itself). The system running the application can be heterogeneous.[13] •

Claim. To the best of our knowledge, *TPL is the first communication library which allows an application to link and run with an arbitrary (quasi-) thread-safe implementation of PVM or MPI without changing the application's code.* •

2.7.1 Concept

A TPL application consists of processes which communicate via explicit message passing. The processes do not share memory (the message passing can but it does not need to be implemented as shared memory communication on a lower level). Each process is assigned a unique *rank*. The code of every process is identical except for their ranks.[14] The rank is usually used at the very beginning of each process to assign the process its *role*. A role is the code executed by a process with a dependence on the rank of the process. (For example, in a process farm there are two kinds of processes: one master and several workers. *MASTER* and *WORKER* are roles.)

TPL uses an *underlying communication library* for message passing. This underlying communication library can be any implementation of PVM or MPI or any other communication library which is at least quasi-thread-safe (or completely thread-safe).[15] The choice of the underlying communication library does not influence the functionality of the application[16] and does not require any change in the application (the interface of TPL remains intact). The layered software architecture is shown in Fig. 2.18.

TPL implements the bindings of its interface to PVM as well as to MPI interfaces. We implemented two versions of TPL. The major differences between the two versions is the message queueing model and the thread management. The first version, TPL 1.0, contains a simple thread management and a queuing model based on so-called message subscription. The second version, TPL 2.0, fixes some problems of TPL 1.0 and directly implements the message passing framework of Section 2.5. There is no internal thread management in TPL 2.0 and the message queueing model is simplified.

[13] The commercial implementation MPI/Pro claims to have the desired properties—however, we could not confirm this. We asked the technical support team at MPI Software Technology, Inc. for more information but the reply we received did not answer our questions.

[14] The concept of all processes having an identical code is known for example from MPI or PARIX. [Par94] Most PVM implementations do not require all processes to run an identical code.

[15] A lack of thread-safety (or quasi-thread-safety) in the underlying communication library would result in a retreat to active polling in TPL which we decided not to support.

[16] The choice of the underlying communication library influences the efficiency of the application.

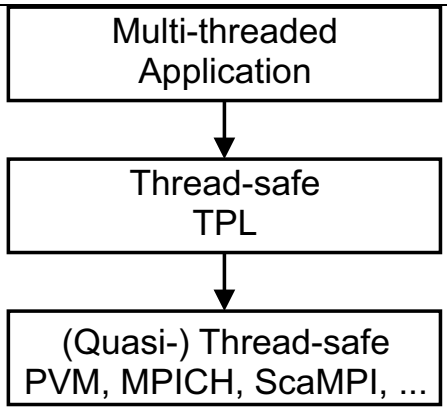

Figure 2.18: TPL layered software architecture

2.7.2 Process startup and termination

All processes are started once at the beginning and they cannot be dynamically created later.[17] Even if this simplification has been made, the hiding of different startup mechanisms in TPL and the way of making them transparent to a TPL application is not easy. Neither MPI nor PVM specify how the processes are initially created. For instance, MPICH uses a script `mpirun` to start the processes. The manual starting of processes from the console is allowed in some PVM implementations whereas in other PVM implementations only the first process is started manually (this first process is then responsible for starting the remaining processes using `pvm_spawn()`).[18] The process startup also remains system specific in TPL. However, from an application programmer's point of view the startup procedure always looks the same.

Each process' entry point is the function `main` in TPL. The command line arguments `argc` and `argv` are passed to the `main` function in all processes:

```
int main(int argc, char *argv[]);
```

As the command line arguments `argc` and `argv` can be used by the system specific startup mechanisms, each TPL process must call the function

```
void tpl_initialize(int *argc, char **argv[])
```

at the very beginning. This function initialises TPL's internal memory structures, makes initial calls which are required by the underlying communication library and restores the original contents of `argc` and `argv` in the first process (if all processes have been spawned at this point then all processes obtain the same `argc` and `argv`).

It is still unclear whether all processes are running at this point. In the case

[17] The static process spawning was a design decision which was made to conform to the available implementations of PVM and MPI.

[18] The startup mechanism can be "hard-wired" in higher-level environments like cluster management systems (e.g. CCS [KRR94], [KR98]).

of e.g. MPICH they are (the `mpirun` script spawns all processes). However, in some implementations of PVM there may be only one process running whose responsibility is the spawning of the remaining processes. TPL takes care of this difference in the implementation of the additional two functions that must follow `tpl_initialize()`:

```
void tpl_world_info(int *nr_procs, int *nr_tasks)
```

returns information on the parallel machine running the program. The number of nodes of the parallel machine is returned in `nr_procs` and the recommended number of processes which should started on the machine is returned in `nr_tasks`.[19] Note that if the processes have already been started, TPL returns the actual number of running processes.

The last function related to startup is `tpl_spawn` which must also be called when the processes have already been started by the underlying system:

```
void tpl_spawn(int *nr_tasks, int *rank, int *argc, char **argv[])
```

The function `tpl_spawn` has several responsibilities. Firstly, it spawns all processes if they have not yet been spawned and it initialises the underlying communication library. `nr_tasks` specifies the number of processes that should be started. `tpl_spawn` modifies `nr_tasks` so that it contains the actual number of tasks that have been successfully spawned. Secondly, it restores the original contents of `argc` and `argv` (if this has not yet been done) and it also sets the working path of all processes to the working path of the process which was spawned first (PVM implementations do not do this).[20] The rank of the calling process is returned in `rank`.[21]

After the sequence
`{tpl_initialize(); tpl_world_info(); tpl_spawn();}`
has been called as explained above, TPL guarantees the following:

- All `nr_procs` processes are running (one process per processor).
- All processes have identical `argc` and `argv` and working directories. (The environment of all processes is identical to the environment of the first process on systems where only the first process is started manually.)
- Each process is assigned a unique rank which is an integer from 0 to `nr_procs` − 1.

[19] The recommended number of processes is usually equal to the number of nodes. However, if TPL detects that the nodes are e.g. double-processors, it recommends starting two processes per node (that means one process per processor).

[20] The underlying communication library is already actively involved in this phase—the initial process of synchronisation requires message passing.

[21] The ranks used in TPL applications are integers from 0 to `nr_tasks` − 1. These ranks may differ from the actual process identifiers which are used by the underlying communication library. TPL takes care of the translation of the ranks to actual process identifiers and vice versa. This translation is transparent to the application.

Remark. It is not possible to control the mapping of processes onto processors in TPL. TPL relies on the default mappings of underlying libraries. If the number of processes is equal to the number of processors, the default mappings always map one process to each processor. It is very seldom that an application needs to map more than one process onto one processor or to place a certain process onto a certain processor. Therefore this is not regarded as a real limitation. •

Each process must call

```
void tpl_deinitialize()
```

before it terminates. `tpl_deinitialize` blocks until all running threads which are registered to TPL (`tpl_add_thread`, see Section 2.7.3) have terminated (more precisely, the call blocks until all threads which are registered to TPL have been unregistered using `tpl_del_thread`, see Section 2.7.3.).[22]

Remark. All the functions above must be called once, from the main thread. •

2.7.3 Thread management

A process of a TPL application consists of threads. One of these threads is special and is called the *main thread*. In a typical application the main thread spawns other threads at the beginning and then serves as a message dispatcher until the process terminates. The main thread is identical to the function `main` which is started by the operating system.

There are many implementations of the thread concept: Solaris threads, POSIX threads (pthreads) OpenMP, GNU threads, ... TPL internally uses the POSIX *pthreads* library because it is available on most systems. It is not really important which of the libraries is used as they all support the same mechanisms—only their interfaces are different.

Threads are not separately addressable entities in the TPL message passing model (a message can only be addressed to a process, using the process' rank). TPL provides a means of delivering a message to a specific thread in a process—however, the thread addressing scheme must be implemented in the application.

Remark. The rest of this section only applies to TPL 1.0. TPL 2.0 does not implement any bookkeeping for the running threads. •

TPL carries out a basic bookkeeping of the running threads. It does not keep a record of each running thread—instead it maintains an internal thread counter

[22] The blocking of `tpl_deinitialize` only applies to TPL 1.0.

in order to ensure the correct termination of the process (TPL must be sure that there are no running threads when it decides to clean up its memory structures). TPL also stores the ID of the main thread for internal purposes.

The application can freely use any functions provided by the thread library but it must assist TPL in its book-keeping task. TPL must be informed of events such as the creation or termination of a thread. In order to ensure the correct synchronisation of *pthreads* and TPL, a few rules must be obeyed in the application:[23]

- Each thread (with the exception of the main thread) must call
  `tpl_add_thread(pthread_self())`
  after it has been created (prior to any further calls to TPL). TPL creates a message queue for this thread at this point (see Section 2.7.4).

- Each thread (with the exception of the main thread) should call
  `tpl_signal_new_thread()`
  after it has been created. This unblocks the thread that created the current thread. `tpl_signal_new_thread()` does not need to immediately follow the call to `tpl_add_thread()`. In the meantime, the newly created thread usually subscribes messages that it wants to receive in the future (see Section 2.7.4).

- Each thread (with the exception of the main thread) must call
  `tpl_del_thread(pthread_self())`
  once, before it terminates (the thread must not make any other calls to TPL after this).[24] TPL destroys the thread's message queue and decreases the internal thread counter at this point.

- Each thread that creates another thread must call
  `tpl_prepare_new_thread()`
  before the call to `pthread_create()`. TPL increases the internal thread counter at this point.[25]

[23] All these rules can be hidden inside TPL. However, this may limit the application in the use of other functions of the *pthreads* library. The approach proposed here is more flexible.

[24] A thread can terminate another thread in which case the destroyed thread does not have an opportunity to call `tpl_del_thread()`. In such a case `tpl_signal_new_thread()` must be called from the other thread, with the ID of the destroyed thread (instead of `pthread_self()`).

[25] Increasing the internal thread counter in `tpl_add_thread` would lead to a racing condition as regards `tpl_deinitialize` which takes care of the correct process termination. (Recall that `tpl_deinitialize` should block until all started threads have called `tpl_del_thread`.) However, there may be a thread that has been created but has not yet called `tpl_add_thread`. In this case TPL does not know about that thread and a call to `tpl_deinitialize` returns although it should be blocked.

- Each thread that creates another thread should call
  `tpl_wait_new_thread()`
  after the call to `pthread_create()`. This call blocks until `tpl_signal_new_thread()` has been called by the created thread.

A generic structure of a multi-threaded TPL process is depicted in Fig. 2.19.

2.7.4 Message passing

There are two layers of message passing in TPL. The coarse layer is message passing between processes. This coarse layer is covered by PVM and MPI and TPL implements the mappings of abstract `send` and `recv` functions onto PVM and MPI functions. The fine layer is message passing involving threads in processes.

In TPL, each process (or rather, each thread of the process) can send a message to any other process or to a set of processes. Each process (or rather, each thread of the process) can receive a message from any other process (or a thread of that process). A message cannot be addressed to a specific thread of a process. However, any thread can receive an incoming message (even several threads can receive the same incoming message).

A message consists of a *header* and a *message body*. The header consists of the sender's rank, recipient's rank and a message tag (an integer defined by the user). The message body is a contiguous buffer containing data. (TPL provides functions for packing and unpacking the data, see Section 2.7.6. These functions take care of different representations of basic data types in different systems.) The message body can be empty. A process can address a message to itself.

Remark. This section focuses on sending and receiving messages in any thread except of the main thread. The main thread can also send and receive messages but it acts as a message dispatcher for all other threads. The mappings of TPL communication functions to PVM or MPI are different to the main thread. The role of the main thread is explained in Section 2.7.5. •

Sending

Sending a message to a process involves a sequence of calls
`{tpl_begin_send(); ...packing...; tpl_send(); tpl_end_send();}`
The `...packing...` part is used for assembling the message and is explained in Section 2.7.6.

The following sending sequence implements the mutual exclusion of the PVM's or MPI's `send` calls sketched in Section 2.6.7:
`void tpl_begin_send()`
locks a mutex which protects a PVM's or MPI's `send()` and then calls

```
void *thread_1(void *arg)
{
  tpl_add_thread(pthread_self());
  /* ...subscribe messages... */
  tpl_signal_new_thread();
  /* ...compute and communicate... */
  tpl_del_thread(pthread_self());
}

/* ...definition of other threads... */

int main(int argc, char *argv[])
{
  int nr_procs;
  int nr_tasks;
  int my_rank;
  pthread_t thread_id;

  /* PROCESS STARTUP */
  tpl_initialize(&argc, &argv);
  tpl_world_info(&nr_procs, &nr_tasks);
  tpl_spawn(&nr_tasks, &my_rank, &argc, &argv);

  /* THREAD STARTUP */
  /* Start thread_1 */
  tpl_prepare_new_thread();
  if (pthread_create(&thread_id, NULL, thread_1, NULL) == 0)
    tpl_wait_new_thread();
  else
    tpl_error("Could not start thread_1\n");
  /* ...start other threads... */

  /* THREAD AND PROCESS TERMINATION */
  tpl_deinitialize();
  return(0);
}
```

Figure 2.19: Generic structure of a multi-threaded TPL process (TPL 1.0)

`interrupt_recv()` which interrupts the blocking `recv` of the main thread. Note that it is safe for the current thread to perform a `send` after this call has returned. Other threads can continue in their computations but become blocked on the locked mutex when they attempt to send messages.[26]

```
void tpl_send(int *recipients, int nr_recipients, int tag,
   *void message, int offset)
```

performs the actual send (`pvm_send` or `MPI_Send`). When `nr_recipients` is equal to one, TPL uses `pvm_send` or `MPI_Send` in the implementation of `tpl_send`. Multiple recipients can also be specified in `recipients` and `nr_recipients` in which case TPL uses the multicast functions of PVM or MPI.

```
void tpl_end_send()
```

unlocks the mutex acquired in tpl_begin_send, allowing other threads to send messages.

Remark. A call to `tpl_send` must eventually return—otherwise the process would not be able to receive or send any further messages. Therefore the `send` function of the underlying communication library must be asynchronous (non-blocking). In the case of PVM, the semantics of `pvm_send` fulfills this requirement.

In the case of MPI, the semantics of `MPI_Isend` also fulfills this requirement but the buffer used by the message must unfortunately be freed after the completion of the `MPI_Isend` (in other words, asynchronous `send` is missing in MPI, see Section 2.6.3). In order to overcome this problem, we extended the MPICH library with a new function. This function, `MPI_Asend`, is almost identical to `MPI_Isend` but the MPICH library frees the message buffer automatically after the request has been completed. •

Receiving

TPL only supports a *blocking receive* even though it would be easy to implement a nonblocking receive as well. In a multi-threaded program any nonblocking receive can be replaced with an additional thread running a blocking receive.

Receiving a message involves of a sequence of calls

`{tpl_begin_recv(); tpl_recv(); ...unpacking...; tpl_end_recv();}`

The `...unpacking...` part is used for disassembling the message and it is explained in Section 2.7.6.

```
void tpl_begin_recv()
```

does nothing. It is reserved for underlying communication libraries other than

[26] An attempt to send a message blocks when another thread is sending at the same time. However, a thread is allowed to receive a message while another thread is performing a `send` because receiving a message in a thread is the same as accessing a local message queue in TPL which is not in conflict with the `send`. (This holds for all threads except for the main thread.)

PVM and MPI.

```
void tpl_recv(MATCHING_FUNC *match, int *sender, int *tag,
   void **message))
```

blocks until a message matching the specified criteria is available. `match` is a user-defined function which obtains the sender's rank and the message tag on input and decides whether there is a match. If there is (if the function `match` returns `TRUE`) `tpl_recv` unblocks and returns the sender's rank, message tag and the packed message body. The message body must be unpacked before a call to `tpl_end_recv()`.
The following is the interface of the function `match`:

```
int match(pthread_t thread_id, int sender, int tag,
   void *message)
```

The function `match` is called internally by TPL to determine whether the header of an incoming message matches the user-specified header. The function returns `TRUE` if there is a match and `FALSE` otherwise.

```
void tpl_end_recv(void *message))
```

frees the memory used by the packed `message`.

Remark. The use of a matching function is a generalisation of PVM's or MPI's matching. PVM and MPI allow either the testing of whether the rank and tag of a message is equal to the given sender's rank and tag, or the specification of a wildcard in the rank or the tag or both. TPL can additionally test for a range of senders' ranks (for instance). •

2.7.5 Message handling and message callbacks

TPL relies on the existence of a `main thread` in an application which serves as a message handler and dispatcher. The main thread is the function `main` which is started by the operating system.

After the main thread has initialised and started other threads, it calls (usually right before `tpl_deinitialize()`)

```
void tpl_handle_messages(HANDLING_FUNC *message_handler)
```

Under normal circumstances the termination of this function results in a termination of the application. This function serves several purposes:

- It receives *all* messages from other processes. The main thread is the only thread which physically receives the messages from the network using a blocking receive function of PVM or MPI that matches all incoming messages.
- Upon the arrival of a message, it unpacks the message, performs a user-defined action (a *callback*) and inserts the unpacked message into other

threads' (*message subscribers'*) message queues.

- It controls the termination of the whole process. Upon the arrival of a user-defined termination message, `tpl_handle_messages` stops listening to the network and returns.

Message queueing model, message subscription and message callbacks in TPL 1.0

Fig. 2.20 shows the message queueing model of TPL 1.0. The understanding of this model is crucial to the understanding of the purpose of the function `message_handler` which is a user-defined function passed as a parameter to `tpl_handle_messages`.

Each thread except for the main thread has its own message queue. None of the threads except for the main thread receives messages from the network. The messages are only looked for in the local message queue. All local message queues are protected by a mutex because they are also accessed by the main thread which inserts messages into them as they arrive. There is also a conditional variable associated with each queue. If a thread tries to receive a message which is not in its queue, it blocks on the conditional variable until it is woken up by the main thread (this happens when a thread is blocked on the conditional variable and a matching message has just been inserted into the queue).

Remark. Note the difference between the sending and receiving of messages in threads (not the main thread). `tpl_send` sends a message directly to the network, whereas `tpl_recv` only accesses a local message queue. •

The thread message queues only contain pointers to messages. One message can appear in several message queues, therefore several pointers can point to the same message data structure. Each message data structure contains a reference counter. After a thread (not the main thread) has read a message from its local queue (`tpl_recv`) and unpacked the message data, the corresponding message pointer is deleted from the thread's message queue and the reference counter in the message data structure is decreased (`tpl_end_recv` takes care of this). The message data structure is freed when the reference counter reaches zero.

Each thread must *subscribe* the messages that it wants to receive in the future. This usually happens immediately after the thread has been started (see Fig. 2.19), but messages can be subscribed and unsubscribed at any time.

```
void tpl_subscribe(pthread_t thread_id, MATCHING_FUNC *match)
```

adds a subscription of the thread `thread_id` to a list of subscriptions.[27] Whenever

[27] The functions `tpl_wait_new_thread` and `tpl_signal_new_thread` (see Section 2.7.3) are used to ensure that a newly created thread has made its subscriptions before the main thread continues. Without this synchronisation a message "addressed" to the newly created thread

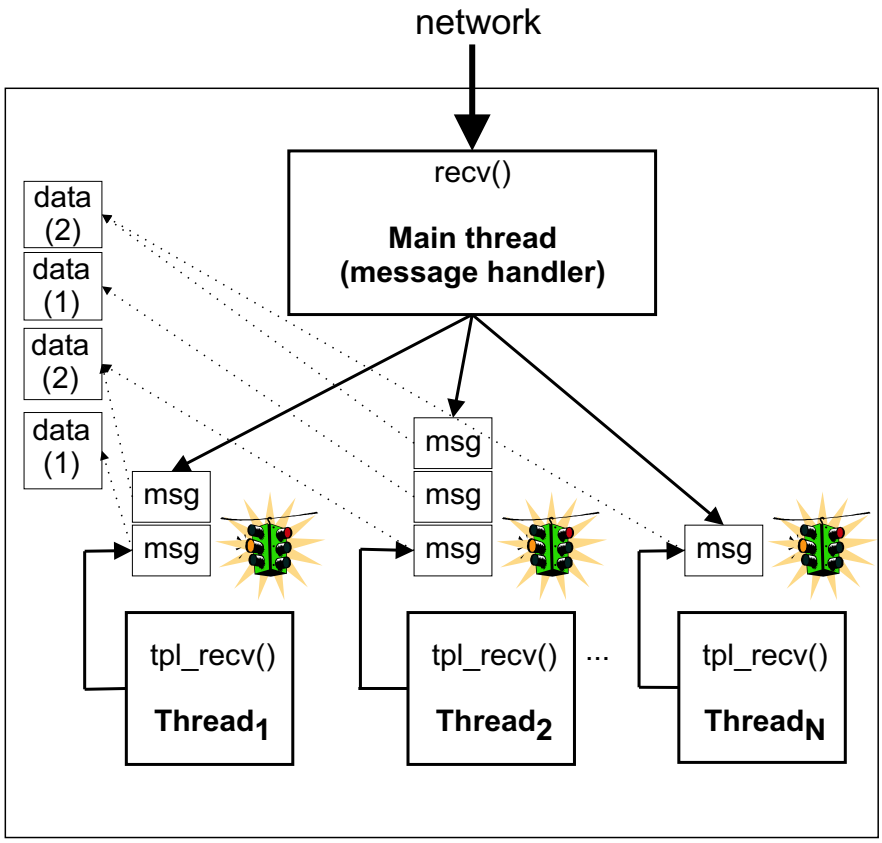

Figure 2.20: Message queueing model of TPL 1.0. Upon the arrival of a message, the main thread inserts messages into the queues of the threads which subscribed the message. In order to avoid a replication of the (possibly large) data stored in the message bodies, only the message headers are inserted into the message queues. The message data is stored only once and referenced by the message headers. The main thread also signals the semaphore associated with the message queue into which it is inserting a message (in order to wake up the thread which may already be waiting for the message)

a message arrives, the main thread matches the message against all subscriptions in the list and when it finds a match, it inserts the message into the message queue of the corresponding thread. If there is no match, the message is discarded.

`void tpl_unsubscribe(pthread_t thread_id, MATCHING_FUNC *match)`

removes a subscription from the list. (If `match` is a `NULL` pointer, all subscriptions of the thread `thread_id` are removed from the list.)

might get lost before the new thread can subscribe it.

Remark. The termination of a process is an example of a situation where several threads want to subscribe the same message (more precisely, a message with the same tag is reserved for termination). Upon the arrival of a termination message, all threads are notified. •

Message callbacks are user-defined actions which are triggered when a message from the network arrives. The idea of callbacks is not to disturb computing threads with events that can be serviced by the main thread.

A typical example of a callback is the servicing of a request from some other process for data stored in the recipient's memory. A recipient's thread can be running a computation in one of its threads while the request for data is received by its main thread. The request does not need to be inserted into the computation thread's message queue. Instead of this the main thread replies with the data and discards the message afterwards.

Callbacks are implemented in the function `message_handler` which is passed to the main thread as an argument of `tpl_handle_messages`. This function is called every time a message arrives. `tpl_handle_messages` terminates when the user-defined function `message_handler` returns `TRUE`. The implementation of the function `tpl_handle_messages` which is called from the main thread is sketched in Fig. 2.21.

Message queueing model, message subscription and message callbacks in TPL 2.0

The queueing model in TPL 2.0 is depicted in Fig. 2.22. There is one global queue of incoming messages (similar to the unexpected queue in the implementation of MPICH and in other MPI implementations). Threads do not need to subscribe and unsubscribe messages in TPL 2.0. Any thread can decide to receive or send a message at any one time. Unlike in TPL 1.0, in TPL 2.0 each message arriving from the network is delivered to only one thread (or consumed by the message handler).

Another difference between TPL 1.0 and TPL 2.0 is that there is no thread management in TPL 2.0. The TPL 2.0 library does not know which threads are running (the library only knows which thread is the *main thread*—the main thread acts as the message handler). The thread synchronisation is left to the application.

The function

```
tpl_handle_messages(HANDLING_FUNC *message_handler)
```

internally calls the user-supplied function `message_handler` upon the arrival of a message. The function `message_handler` processes the message and returns one of the following values:

- `TPL_ACTION_ENQUEUE` indicates that the message should be enqueued in the

```
void tpl_handle_messages(HANDLING_FUNC *message_handler)
{
  int sender;
  int tag;
  MESSAGE *msg;
  void *packed_data, *unpacked_data;
  int quit;

  quit = FALSE;
  while (! quit)
  {
    tpl_begin_recv();
    /*
    Note:
    tpl_recv receives from network when called from main thread
    */
    tpl_recv(pthread_self(), &sender, &tag, &packed_data);
    quit = msg_handler(sender, tag, packed_data, &unpacked_data);
    tpl_end_recv();
    /* ...initialise the message structure msg... */
    msg->sender = sender;
    msg->tag = tag;
    msg->data = unpacked_data;
    /*
    ...match msg against all subscriptions and
    insert msg into message queues of subscribers...
    */
    /* ...if no subscriber found then discard msg... */
  }
}
```

Figure 2.21: Implementation of `tpl_handle_messages` in TPL 1.0

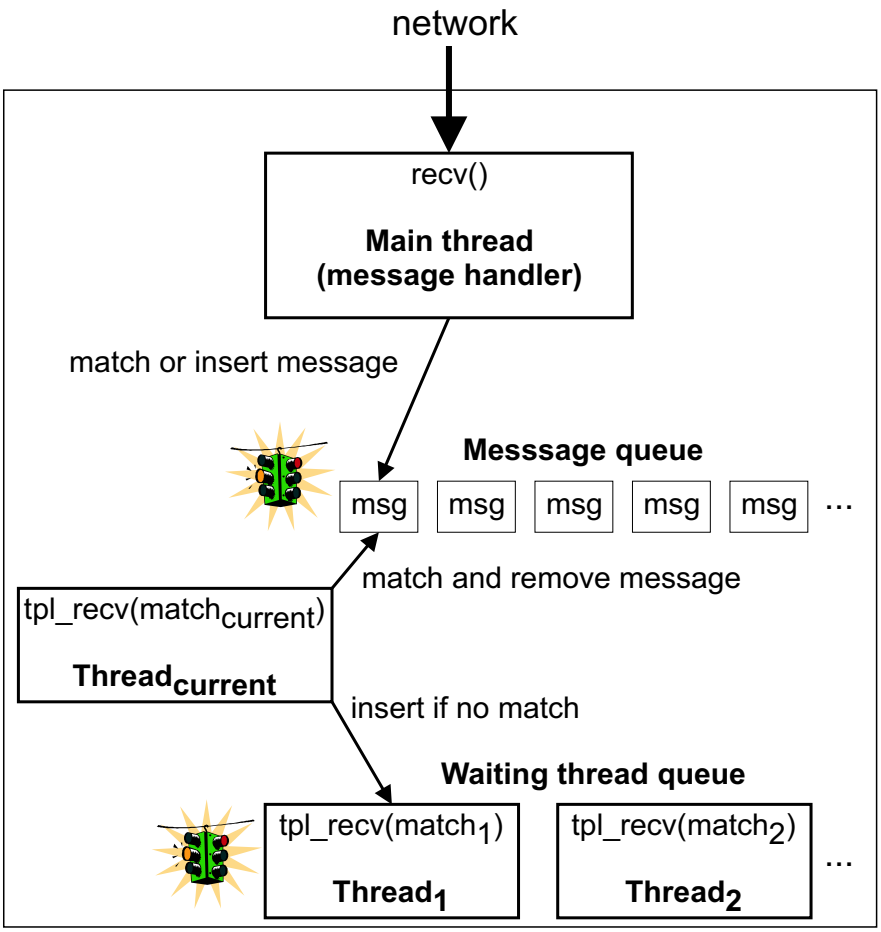

Figure 2.22: Message queueing model of TPL 2.0. Upon the arrival of a message, the main thread first looks for a match among the threads waiting in the thread queue. If there is a match, the message is passed to the waiting thread (and the thread is woken up). If there is no match, the message is inserted into the message queue. A thread (which is not the main thread) which is calling `tpl_recv()` first looks into the message queue. If it finds a matching message in the message queue, it removes it from the queue. If there is no matching message in the message queue, the thread inserts itself into the waiting thread queue

message queue.

- `TPL_ACTION_DROP` indicates that the message has already been processed by the `message_handler` and it should be forgotten without being enqueued.
- `TPL_ACTION_EXIT` indicates that the function `tpl_handle_messages` should terminate (without the insertion of the message into the message queue).

The return value of `TPL_ACTION_EXIT` is used to terminate the main thread. The application must take care about the correct termination of the remaining threads before the main thread terminates. Note that after the termination of the main thread no messages can be sent or received. A useful trick in the implementation of a termination protocol is to run a communication round which ensures that all processes are ready to terminate. After this round every process sends a termination message to itself.

In TPL 2.0, the user-supplied function `message_handler` is not responsible for unpacking the message body (unless the handler is willing to process the message itself and drop it afterwards).

2.7.6 Message packing and unpacking

TPL supports *heterogeneous systems* in the extent of the underlying communication library. Processors that run the processes may be different or run different operating systems. Data types (e.g. `int`, `float`) can have different lengths on two processors or their binary representation can differ.

This has some implications for message passing. It is desirable that if a process 0 sends a message containing an integer 7 to process 1, then the process 1 should see the same value 7 in its local copy of the integer, despite the different representations of 7 in both processes.

Both PVM and MPI solve this problem using *datatypes*. A datatype corresponds to a simple data type of a programming language: `char`, `int`, `float`, `long` etc.[28] The communication library must have information on datatypes stored in messages. It can then translate the data to the XDR format which is defined by POSIX on the receiver side and encode the XDR representation to the sender's format on the sender side.

PVM and MPI offer packing and unpacking functions for all simple datatype. The assembling of a message in a contiguous buffer in the sender is a sequence of calls to packing functions. The disassembling of a message in the receiver is a sequence of corresponding calls to unpacking functions.[29] Finding a common

[28] MPI also uses more complex datatypes (derived datatypes) which correspond to higher level memory structures in programming languages (`struct`, `array`, ...). They are convenient but not necessary.

[29] PVM and MPI also offer other possibilities for message assembly which do not require the copying of data in a contiguous buffer ("data in place"). This is not supported in TPL.

interface which maps well to both PVM and MPI is not easy because of the differences between these two libraries. The main difference is that PVM uses an internal global buffer for storing the packed data, whereas MPI uses buffers allocated by the application. A consequence of this is that in MPI the application has to determine in advance (before packing a message on the sender side or before receiving a message on the receiver side) how much space the sending and receiving buffers will require. PVM adjusts the buffer sizes on the fly, transparently to the application.

TPL must be able to emulate the packing and unpacking of both PVM and MPI without changing the interfaces of the libraries. The interfaces of TPL's packing/unpacking functions correspond to the interface of MPI. However, TPL internally uses a global sending buffer and a global receiving buffer. The use of global buffers, which is dictated by the need to map onto PVM, is the reason why threads in TPL must already be mutually excluded during the packing phase (see Section 2.7.4)[30]. This can be a potential source of inefficiency in TPL in the situation when several threads attempt to pack and send messages at the same time.

MPI requires the receiving process to allocate the buffer for an incoming message. TPL allocates buffers of a default size during the initialisation phase of each process. If the application sends a message that is larger than size of the the default buffer, the buffer in the receiver must be adjusted. TPL uses an internal protocol to ensure that there is enough space in the receiver. When the sender detects that it is sending a message that might exceed the receiver's buffer size, it first sends a controlling message telling the receiver to increase its buffer size.[31] This leads to an additional communication overhead for large messages but the additional overhead can be avoided by setting the default buffer size sufficiently large.

2.7.7 Error handling and debugging

Error handling in TPL is very strict. It is assumed that the application is correct (for instance, it is assumed that a message is always addressed to an existing process)—there are no sanity checks inside TPL which would significantly influence efficiency. Internal sources of problems such as an unsuccessful attempt to allocate memory lead to the termination of the process.[32]

[30] Although PVM 3.4 can work with multiple buffers, it is not thread-safe on the set of packing and unpacking functions.

[31] Note that the representation of the data can require more memory in the receiver than in the sender (for instance, a `float` can be coded using 4 bytes in the sender but 8 bytes in the receiver). TPL uses a pessimistic upper bound to estimate the ratio of different lengths of simple types.

[32] At the time of writing, the implementation of TPL does not attempt to "correctly" terminate all the processes of the application. A future version of the TPL library may implement

Debugging facilities provided by the underlying communication libraries can be used with TPL. TPL does not provide any additional debugging tools.

2.7.8 Flow control

Some implementations of MPI (e.g. MPICH) use the so-called flow control mechanism. This mechanism avoids the flooding of a process with messages which arrive from other processes. There is a certain fixed amount of memory allocated for the buffering of unexpected incoming messages (an unexpected message is a message for which no receive operation has been posted). Once this buffer is full, the process begins to only receive messages for which a receive operation is posted.

The *buffering at receiver* can increase efficiency of message passing applications in some scenarios. Consider the following sequence of message passing operations which involves three processes, A, B and C:

1. A posts a receive operation which matches any message from B.
2. C sends a message to A.
3. B sends a message to A.
4. A posts a receive operation which matches any message from C.

What happens in step 2? The message being sent by C either remains in C or it is passed to A even though there is no matching receive operation in A. In the latter case the message is stored in the unexpected message queue of A (without the completion of the send operation) and it is retrieved from the unexpected queue in step 4. The retrieval from the unexpected message queue is usually much faster than the transfer of the message from C to A. Hence, the costs of the transfer are amortised in the latter case.

However, if the message being sent in step 2 was larger than the buffering space available in the process A, then the flow control mechanism would not allow the transfer of the message to A in step 2—the transfer would be postponed to step 4.

Note that the scenario above works correctly in both cases. Its result does not depend on whether the process A buffers the message sent in step 2 or not. The following scenario is more dangerous as it may *sometimes* result in a deadlock:

1. A sends a message to B.
2. B sends a message to A.

a global handling of internal fatal errors (if the underlying communication library provides an error handling).

The MPI standard states that this scenario will result in a deadlock if the synchronous send (`MPI_Ssend`) is used in either A or B. The MPI standard also states that the scenario will not result in a deadlock if the nonblocking send (`MPI_Isend`) is used in both A and B. Finally, the standard states that the scenario may result in a deadlock (depending on the implementation of the MPI library) if the default send (`MPI_Send`) is used in both A and B (the same holds for the combination of `MPI_Send` and `MPI_Isend`). The whole truth is even much worse—the second scenario may sometimes deadlock and sometimes not with the same MPI implementation if `MPI_Send` is used in both A and B! The buffer spaces in A and B may be large enough to store one message. In this case both A and B will make progress if their buffers are empty. However, a third process C may already have sent an unexpected message to both A and B before and the unexpected message may already have consumed the entire available buffering space in A and B. In this case the scenario will result in a deadlock. No deadlock occurs if the third process does not participate in this communication scenario.

Remark. The second scenario can often be found in practice, especially in parallel finite-element methods. For instance, consider an application which consists of parallel processes connected to a ring. All the processes run an identical code. The parallel computation runs in rounds. In one round each process sends a value to its two neighbours and receives a value from its two neighbours. The use of the synchronous send results in deadlock unless the symmetry in the ring is broken (see the example solution of the Problem of Dining Philosophers in Section 2.2.1). As the breaking of the symmetry usually involves structural changes in the application code, a solution is preferred which preserves the symmetry. A solution which involves the use of the nonblocking send `MPI_Isend` is given in the MPI tutorial book [GLS95]. This solution is independent on the amount of the buffering space at receiver. •

The previous discussion suggests that flow control mechanism in the combination with buffering at receiver is very useful because it speeds up applications if there is enough buffer space at receiver and at the same time takes care of not exceeding the available buffer space in the processes. As the flow control does not apply to the nonblocking send (`MPI_Isend`), the problem of deadlock in symmetrical scenarios seems to be solved. However, the amount of memory *at sender* is also limited. If there is no throttle on the amount of memory taken by the pending nonblocking sends, then a process which issues too many nonblocking sends runs out of memory.

Flow control is a pessimistic solution to the problem of finite memories of the processes. Flow control assigns a fixed amount of the buffering space to every process. There are scenarios which result in a deadlock even though there may be more available memory in the processes involved than the fixed amount of buffer space.

Active messages

TPL uses an optimistic approach to the problem above. *All the incoming messages are buffered at receiver.* This approach attempts to amortise as much latency and message transfer overhead as possible. The disadvantage is that one of the processes can be overflooded with messages for which no matching receive has been posted at that process.[33] However, this disadvantage is only illusory for the following reasons:

- The communication scenarios of many applications guarantee that no process is overflooded with unexpected messages.
- The applications in which a process may be overflooded with unexpected messages result in a deadlock if flow control is used.
- The implementation of flow control is fairly easy in TPL. Hence, TPL gives the application the possibility to decide whether flow control should be used. Unlike in MPI implementations, the buffer spaces taken by the incoming and outgoing messages can be separately controlled in TPL.

Remark. The flow control mechanism which is implemented in MPICH is a source of inefficiency in TPL if MPICH is used as the underlying communication library. The flow control mechanism of MPICH must be *overcome* in order to implement the optimistic approach in TPL. A part of this overcoming is hidden in the implementation of the function `MPI_Asend` (see Section 2.7.4). The problem of the overcoming of the MPICH's flow control mechanism is only tackled in TPL 2.0 (TPL 1.0 does not work correctly with an underlying MPI library which uses flow control).

PVM 3.4 uses buffering at receiver but it uses no flow control. This simplifies the design of the TPL's operation binding to PVM. •

2.8 Efficiency benchmarks

This section presents comparison of efficiency of the original PVM and MPI implementations against TPL based on the same libraries. We used an "out of the box" build of PVM 3.4 and MPICH 1.2.4 with the quasi-thread-safe extensions described in Section 2.6.6 (the extensions have no impact on the efficiency of applications that do not use them).

[33] The matching receive must be either posted by a thread which is different from the main thread or the message must be consumed by a callback in the main thread. Otherwise the message is inserted into the "unexpected queue" of the receiving process. The growth of this "unexpected queue" beyond the available memory is called overflooding.

All the measurements were run on the Fujitsu-Siemens *hpcLine* cluster in the Paderborn Center for Parallel Computing (PC^2) at the University of Paderborn, Germany. The cluster consists of 96 Siemens Primergy double-processor nodes. The nodes have two independent network interfaces: SCI (500 MBit/second Scalable Coherent Interface by Scali/Dolphin) and Fast Ethernet (100 MBit/sec). We used the Fast Ethernet network which is supported by both "out-of-the-box" PVM 3.4 and MPICH 1.2.4 libraries. Each node of *hpcLine* is a double-processor 850 MHz Intel Pentium III with 512 Mbytes RAM, running Linux Redhat.

We observed various aspects of polling and event-driven approaches on two benchmarks:

- *ONE-SIDED THREADED PINGPONG* and
- *SYMMETRICAL THREADED PINGPONG*.

These benchmarks are new. They cannot be found among standard benchmark programs for PVM and MPI although their structure, especially the structure of *SYMMETRICAL THREADED PINGPONG*, directly corresponds to the structure of all irregular parallel programs. The traditional *PINGPONG benchmark* involves two parallel processes, whereby one of the processes acts as a server which awaits "PING" messages and responds with "PONG" and the other process generates the "PING" and waits for the "PONG" replies. In our benchmarks, especially in the *SYMMETRICAL THREADED PINGPONG*, *both* processes act as client and servers at the same time.

Both benchmarks involve two parallel processes, $PING$ and $PONG$, whereby the process $PING$ consists of two threads, T_1 and T_2. The thread T_1 sends a number of messages to the process $PONG$ and the thread T_2 only receives the same number of messages from the process $PONG$. The two benchmarks only differ in the implementation of the $PONG$ process. In *ONE-SIDED THREADED PINGPONG*, the $PONG$ process is single-threaded. It first waits for a message arriving from the process $PING$ and after the reception of the message it sends an answer back to the process $PING$. In *SYMMETRICAL THREADED PINGPONG*, the $PONG$ process also consists of two threads, T_1 and T_2. The thread T_1 of the process $PONG$ does nothing. The messages are only received and sent in the thread T_2. However, T_2 must be aware of that T_1 *may* also want to communicate. This implies that T_2 must not use a blocking `recv` unless the communication library is thread-safe.

Polling and event-driven versions of both benchmarks (Fig. 2.11 and Fig. 2.12) were implemented and compared in the measurements. The functions `compute` and `service_request` are empty. We did not use the generic polling scheme from Fig. 2.12 for the measurements but an optimised one. Fig. 2.23 shows the optimised pseudo-code of $PING$. The difference between this one and the generic version from Fig. 2.12 is that the optimised version avoids calling `sleep(time)`

when there is a continuous flow of messages arriving in *PING*. (Each `sleep()` call costs at least 0.02 sec, see Section 2.6.4.)

```
thread T1()
{
  while (not_done)
  {
    lock(comm);
    send();
    unlock(comm);
  }
}
```

```
thread T2()
{
  while (not_done)
  {
    lock(comm);
    arrived=probe();
    while (arrived)
    {
      recv();
      arrived=probe();
    }
    unlock(comm);
    sleep(time);
  }
}
```

Figure 2.23: An optimised polling implementation of the *PING* process. The optimal setting of `time` in the `sleep(time)` call is 50 milliseconds (see Section 2.6.4). This optimal setting was used in the measurements

All event-driven measurements with *ONE-SIDED THREADED PINGPONG* were performed using TPL 1.0 and all measurements with *SYMMETRICAL THREADED PINGPONG* were performed using TPL 2.0. However, the influence of the differences between TPL 1.0 and TPL 2.0 on the measurements is neglectable.

In the following, `MPICH/TPL` refers to benchmarks which use the even-driven version of TPL based on quasi-thread-safe MPICH. Similarly, `PVM/TPL` denotes the even-driven version of TPL based on quasi-thread-safe PVM. `PVM/polling` and `MPICH/polling` denote the polling versions of the benchmarks based on the official PVM 3.4 and MPICH 1.2.4 libraries.

2.8.1 *ONE-SIDED THREADED PINGPONG*

In each run of the *ONE-SIDED THREADED PINGPONG* program, 100000 messages were sent from *PING* to *PONG* and 100000 messages were sent in the opposite direction. Each run was repeated using message sizes from 1 Byte to 1 MByte (steps of powers of 2). Moreover, each of these rounds was repeated 10 times in order to exclude external factors related to the operating system and the network.

The absolute running time of one experiment was measured. The timer was started in the process *PING* right before the `while (not_done)` loop and stopped right after the loop. In order to exclude a possible initial message passing latency of PVM and MPICH, the loop was preceded by a "warmup" phase during which the *PING* and *PONG* processes exchanged a few messages. We also measured the number of `sleep()` calls in each run of the polling version of the benchmark.

1 hpcLine node

The graph in Fig. 2.24 shows the average measured throughput over the 10 rounds. Throughput is the total size of all messages sent during a run, divided by the duration of the run (messages in both directions count). Throughput is a measure which is strongly correlated to the efficiency of communication-intensive parallel applications. In this set of measurements, the processes *PING* and *PONG* were run on the same node (a loopback was used for the physical communication, avoiding the networking overhead).

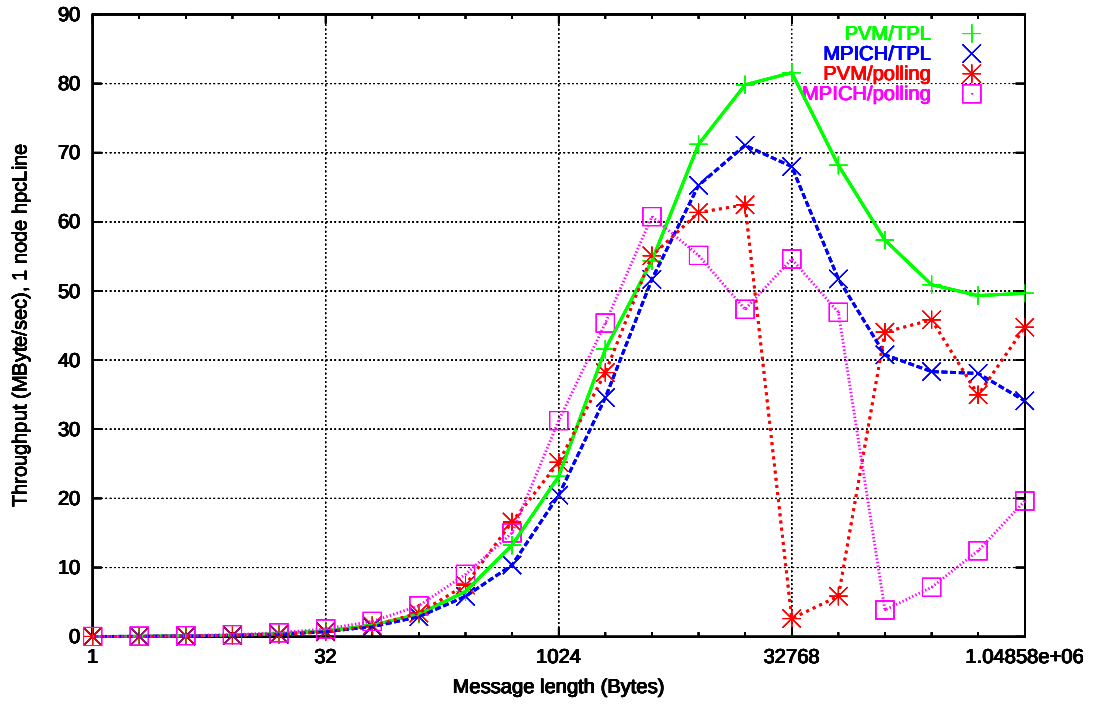

Figure 2.24: Average throughput, 1 node hpcLine

The `PVM/polling` graph must be compared to the `PVM/TPL` graph. The polling version is on average only slightly better than the event-driven version for message sizes up to 1 kByte. For larger message sizes `PVM/TPL` is not only clearly better

but also much more stable.[34] Note the `PVM/polling`'s falloff at the message size of 32 kbytes—this is the message size where the message flow arriving in $PING$ is not continuous and so the thread T_2 calls `sleep(time)` very frequently (see Fig. 2.23). The danger of such falloffs is that they can neither be predicted nor systematically eliminated.

The same holds for the comparison of the `MPICH/polling` to `MPICH/TPL`. `MPICH/TPL` is slightly worse for small message sizes but clearly better for message sizes from 8 kbytes. Also—most importantly—`MPICH/TPL` is much more stable than `MPICH/polling`.

Another measure of stability is the standard deviation of the absolute times, or (after scaling by the total message size) the standard deviation of the throughput over the same 10 runs. The standard deviation of the throughput is shown in Fig. 2.25.

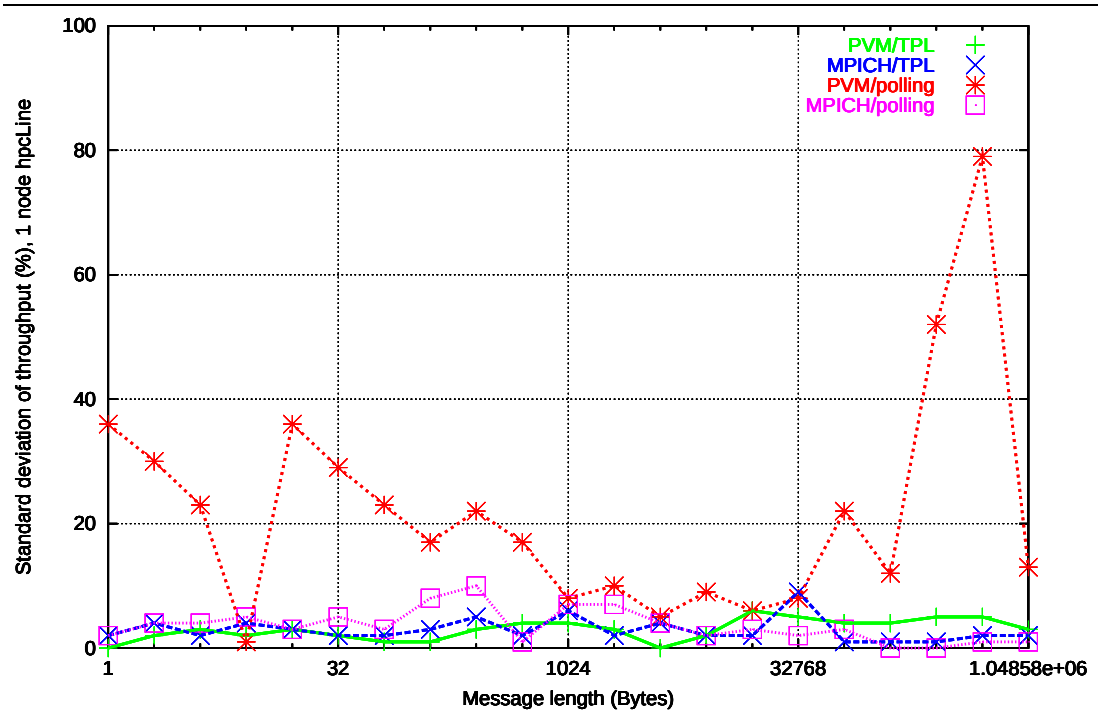

Figure 2.25: Standard deviation of throughput, 1 node hpcLine

The standard deviations of the event-driven `PVM/TPL` and `MPICH/TPL` should theoretically be 0 because the overhead of the interrupt mechanism is constant for every message arriving in $PING$. These deviations are indeed very low but range from 0 to 8%. The non-zero values can be explained by external system factors (e.g. thread scheduling brings certain irregularities to the measurements).

[34]The characteristics of the `PVM/TPL` graph are similar to the characteristics of a simple pingpong PVM benchmark. In a simple pingpong benchmark the process $PING$ is single-threaded. It runs a loop in which it first sends a message and then receives the reply.

`MPICH/polling` is surprisingly stable on this measure. The same experiment always took approximately the same time when it was repeated 10 times. The difference in the number of `sleep()` calls in the runs was also neglectable. This does not correspond to our expectations—the number of `sleep()` calls should be very random. Only a detailed study of MPICH's complex buffering model may lead to a satisfactory explanation. MPICH also exhibits surprising regularities also in specially constructed irregular communication scenarios. [KS02]

`PVM/polling` is as expected very unstable. Different runs of the same experiment produce very different times. There is an *extremely high correlation* between the standard deviation of `sleep()` calls in the runs and the standard deviation of throughput (or absolute time). The graph in Fig. 2.26 is the standard deviation of the number of `sleep()` calls. This graph and the `PVM/polling` graph in Fig. 2.25 are almost identical!

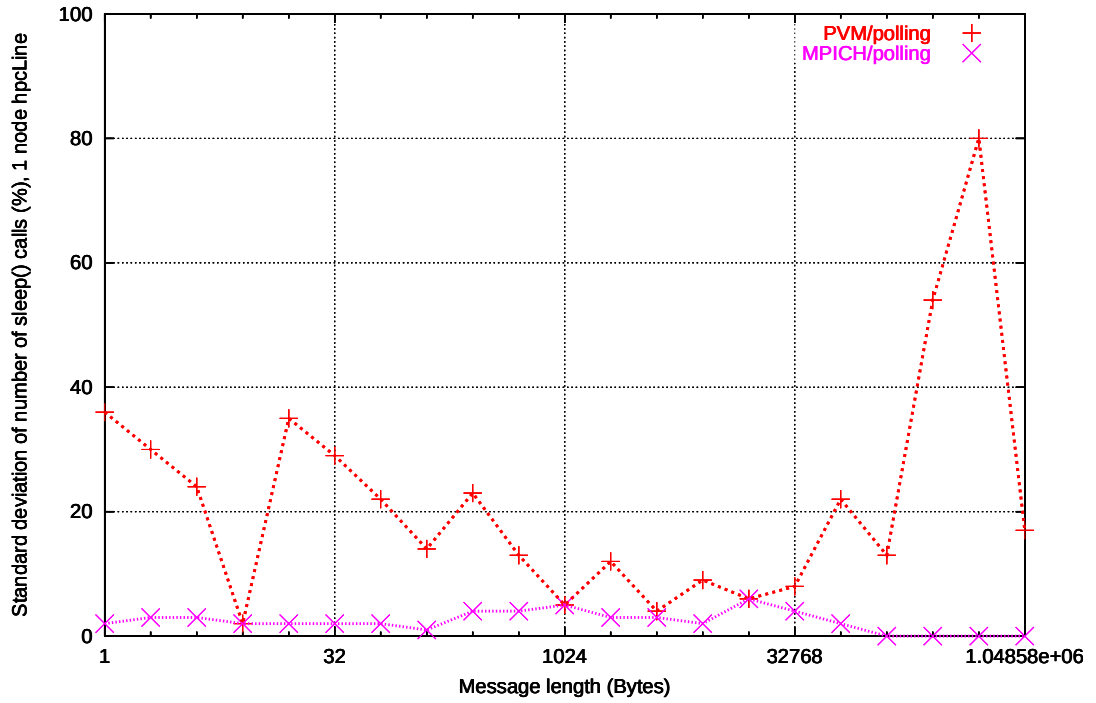

Figure 2.26: Standard deviation of the number of `sleep()` calls in the polling versions of the benchmark, 1 node hpcLine

The polling overhead further depends—when the optimised polling version is used—on the continuity of the message flow. The continuity of the message flow (which is measured as the number of `sleep()` calls) depends on the application that generates the messages, on the buffering model used by the communication library and by the network, and on the polling inside the communication library (see Section 2.6.3). The buffering has a "smoothing" effect—messages that have been sent by *PONG* in short time intervals (there is a short pause

between each two messages sent by the process $PONG$) will be received in the process $PING$ as a continuous flow (see Fig. 2.27). Note that the $PINGPONG$ benchmark (whether $PING$ is multi-threaded or single-threaded is irrelevant) has the potential to generate continuous message flows. However, a continuous stream of large messages leads to buffer overflows which cause "cuts" in the flow. Each "cut" results in a `sleep()` call. Certain settings of message sizes trigger the generation of the "cuts" in message flows, which reverts the "smoothing" effect of the buffering. This leads to falloffs that can be seen in `PVM/polling` and `MPICH/polling` graphs in Fig. 2.24.

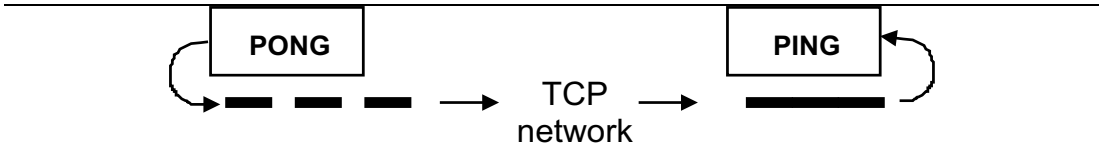

Figure 2.27: Smoothing effect of the TCP protocol (Nagel's algorithm). [Ste94], [WS95] A non-continuous message flow generated by the process $PONG$ is received as a continuous message flow in the process $PING$

The continuity of the message flow is expressed by *burstiness*. Burstiness is the average size of the data transferred between two "cuts" (the size of the data between two successive `sleep()` calls. The graph in Fig. 2.28 shows an average burstiness over the 10 runs (the total transferred data size divided by the measured number of `sleep()` calls) for the polling version of the benchmark. Note the extreme similarity between the graphs in Fig. 2.28 and in Fig. 2.24!

2 hpcLine nodes

We repeated all the experiments on 2 hpcLine nodes, by mapping the processes $PING$ and $PONG$ onto different nodes. The only difference between this and previous scenarios is that the network overhead is added to this scenario.

The graph in Fig. 2.29 shows the average measured throughput over the 10 rounds. (Note that 25 MBits/second is the physical limit of Fast Ethernet.) The average throughputs of `PVM/polling` and event-driven `PVM/TPL` are similar. `MPICH/polling` is even better than `MPICH/TPL`—however, there is a falloff at a message size of 128 kByte.

The graph in Fig. 2.30 depicts the deviation of the throughput. The deviation of `PVM/polling`'s throughput is extremely high for messages up to 1 kByte. This was expected. However, `PVM/polling` behaves very deterministically for large messages. `MPICH/polling` is also deterministic, as on 1 node. We are not able to explain this phenomenon. Without a detailed study of the buffering models of PVM and MPICH and their synchronisation with TCP buffering [Ste94], [WS95] a satisfactory explanation cannot be given. We can only point out that we did not change the "out of the box" distributions of PVM and MPICH which may not

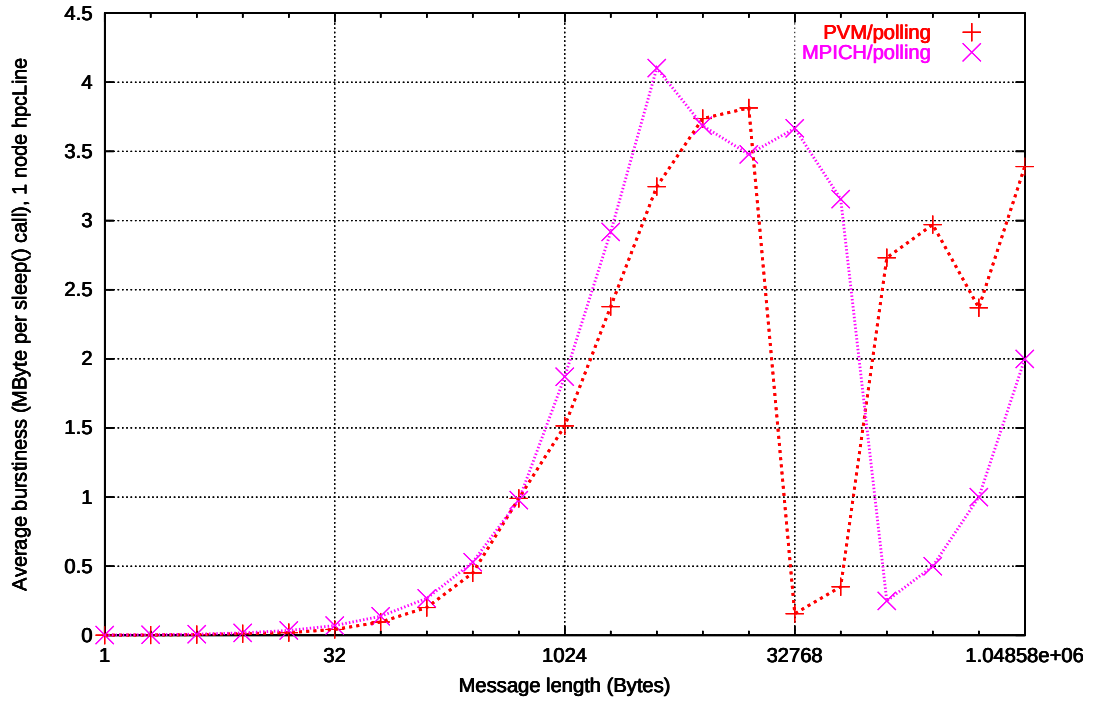

Figure 2.28: Average burstiness (amount of transferred data per `sleep()` call) for the polling versions of the benchmark, 1 node hpcLine

have been optimal for the TPL implementation. (TPL uses these distributions as the underlying communication libraries, see Fig. 2.18.)

A comparison of the graphs in Fig. 2.30 and Fig. 2.31 again reveals an extremely strong correlation between the standard deviation of throughput and the standard deviation of the number of `sleep()` calls for `PVM/polling` and `MPICH/polling`.

Similarly, the polling graphs in Fig. 2.32 (average burstiness) and Fig. 2.29 (average throughput) are almost identical. This means that the throughput of the polling version strongly depends on the continuity of the data flow. Bursts in the data flow cause the more frequent calling of `sleep()` inside the polling loop, which is the source of the efficiency loss.

2.8.2 *SYMMETRICAL THREADED PINGPONG*

The difference between *ONE-SIDED THREADED PINGPONG* and *SYMMETRICAL THREADED PINGPONG* is that in the latter benchmark both the processes *PING* and *PONG* use the same mechanism in order to receive messages. *SYMMETRICAL THREADED PINGPONG* therefore corresponds to a generic implementation of a non-trivial application in which all processes are acting as clients and servers at the same time.

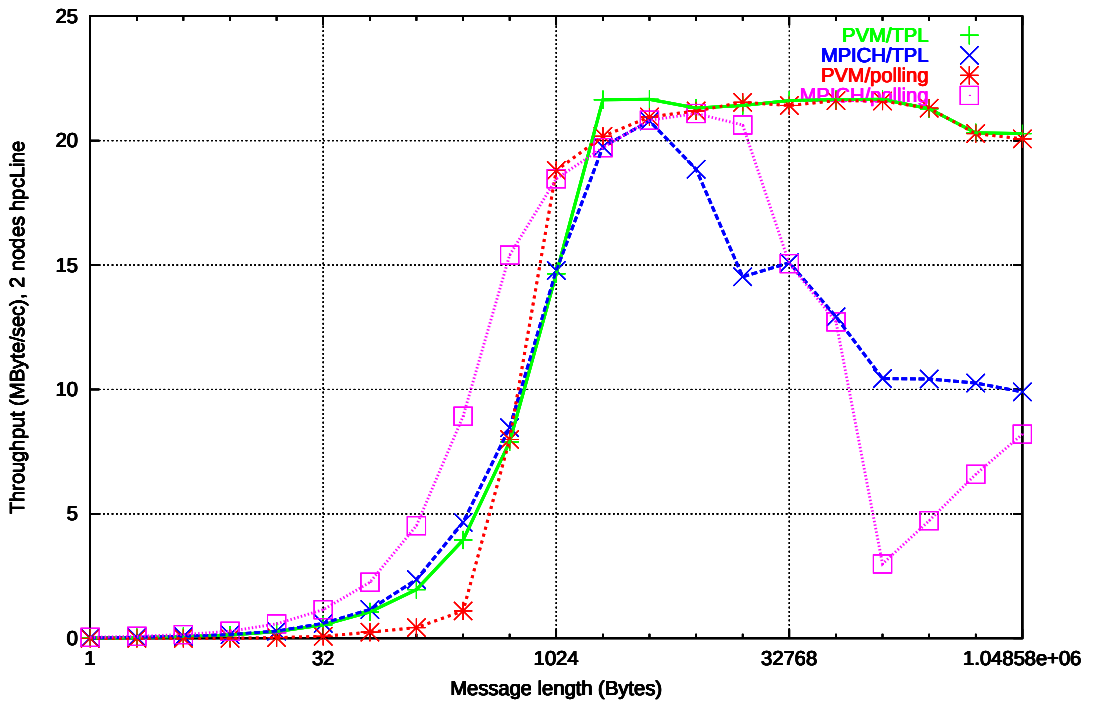

Figure 2.29: Average throughput, 2 nodes hpcLine

It is often believed that a high latency is the main drawback of the event-driven approach. [HSS+98], [PS98] This may be true for applications which do not require thread-safety or asynchronous communication. All other applications must use polling and suffer from a high latency caused by polling. The goal of the measurements with the *SYMMETRICAL THREADED PINGPONG* benchmark was to compare the latencies of the event-driven and the polling versions. The source code of the programs which were used for this comparison is in Appendix B.

Average roundtrip times measured between 2 nodes of hpcLine are shown in Fig. 2.33 as a function of the number of roundtrips. The roundtrips are initiated by the process *PING* and they are independent on one another (a roundtrip begins without waiting for the completion of the previous one). The message size was set to 1 Byte in all experiments, each experiment was repeated ten times.

Latency is usually defined as a single roundtrip time divided by two. Hence, latency corresponds to the first column in Fig. 2.33. Note that the latency of the event-driven `PVM/TPL` and `MPICH/TPL` is very close to zero at this scale. Fig. 2.34 is the same graph, only 100 times magnified. The latencies of the polling versions of this benchmark are 0.17 for `MPICH/polling` and 0.3 for `PVM/polling`—about 500 times higher than the latencies of the event-driven versions (ca. 0.0006 seconds)![35] The polling latencies amortise with a growing number of the number

[35]For a comparison, a single roundtrip time of the raw TCP protocol (with no higher com-

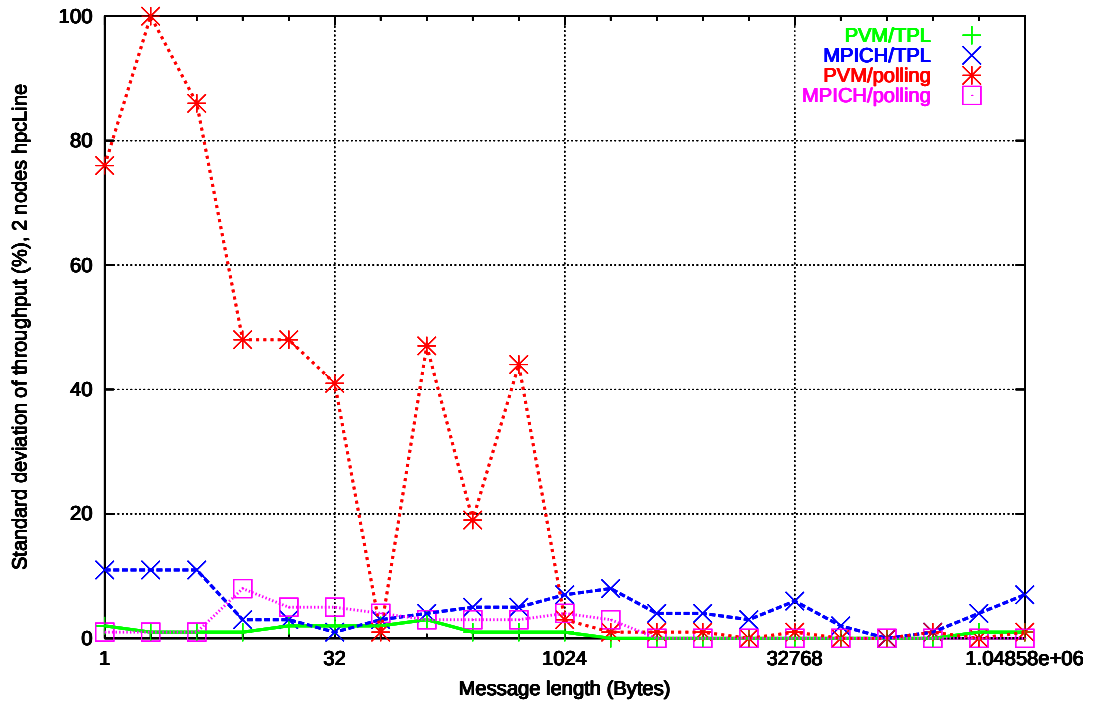

Figure 2.30: Standard deviation of throughput, 2 nodes hpcLine

of roundtrips. However, a usual communication scenario in each process of a non-trivial application is "send a request; wait for the reply", in which case there is only one roundtrip performed at any one time (from the point of view of the process which initiated the request). Even if there was an application which would make use of performing many roundtrips by one process in parallel without waiting for the replies, the process would eventually run out of memory. The reason is the missing flow control mechanism for outgoing messages in PVM and MPICH, see Section 2.7.8.

Theoretically, the latency of the event-polling version should be constant. However, both the `PVM/TPL` and `MPICH/TPL` graphs in Fig. 2.34 contain two unexpected peaks. We are not able to explain these peaks—we only suspect that they are related to the residual amount of polling in the underlying quasi-thread-safe communication libraries, or to the Nagel's algorithm of the TCP protocol, or to the non-optimal thread-switching policy in Linux.

2.8.3 Summary of benchmarking results

Our theoretical expectations from Section 2.6.4 are fully justified by the experiments. The throughput of the polling implementations of the benchmarks is

munication library involved) is ca. 0.00015 seconds.

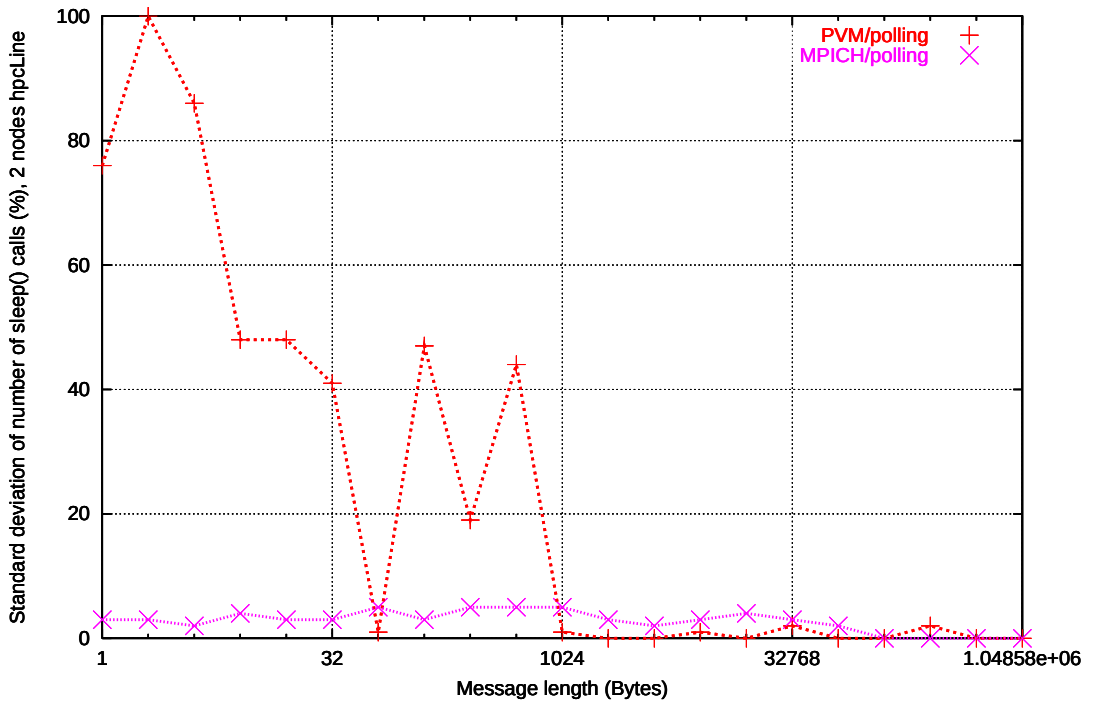

Figure 2.31: Standard deviation of the number of calls in the polling versions of the benchmark, 2 nodes hpcLine

a function of the number of `sleep()` calls in the polling loop. The overhead of every `sleep()` call which adds to the parallel time is 0 to 0.02 seconds (or more)[36]. Thus the total polling overhead in our benchmarking scenarios ranges between 0 and 2000 seconds (= 0.02 seconds * 100000) in each run. This randomness can clearly be observed in Fig. 2.25 and Fig. 2.26 for `PVM/polling`. The absolute times of the runs without this overhead (a simple single-threaded pingpong benchmark) are between ca. 5 seconds and 6000 seconds, depending (only) on the size of the messages used in the benchmark. It is also evident from the `MPICH/polling` graphs that the performance depends solely on the number of `sleep()` calls in the polling loop.

The overhead of the event-driven mechanism is constant in the benchmarks because the mechanism is triggered by every message that arrives in the process *PING*. It does not depend on the continuity of the message flow. This overhead is slightly larger than the polling overhead by continuous message flows, and much smaller than the polling overhead when the smoothing effect of TCP buffering (Fig. 2.27) does not help.

[36]This is only a one-sided overhead (when only the *PING* process runs the polling loop). However, it is likely in a non-trivial application that *all* processes are symmetric in this respect—that means, the process sending a request also runs a polling loop when it is waiting for a reply. In such a case, the overhead of servicing one request ranges from 0 to 0.04 seconds (or more).

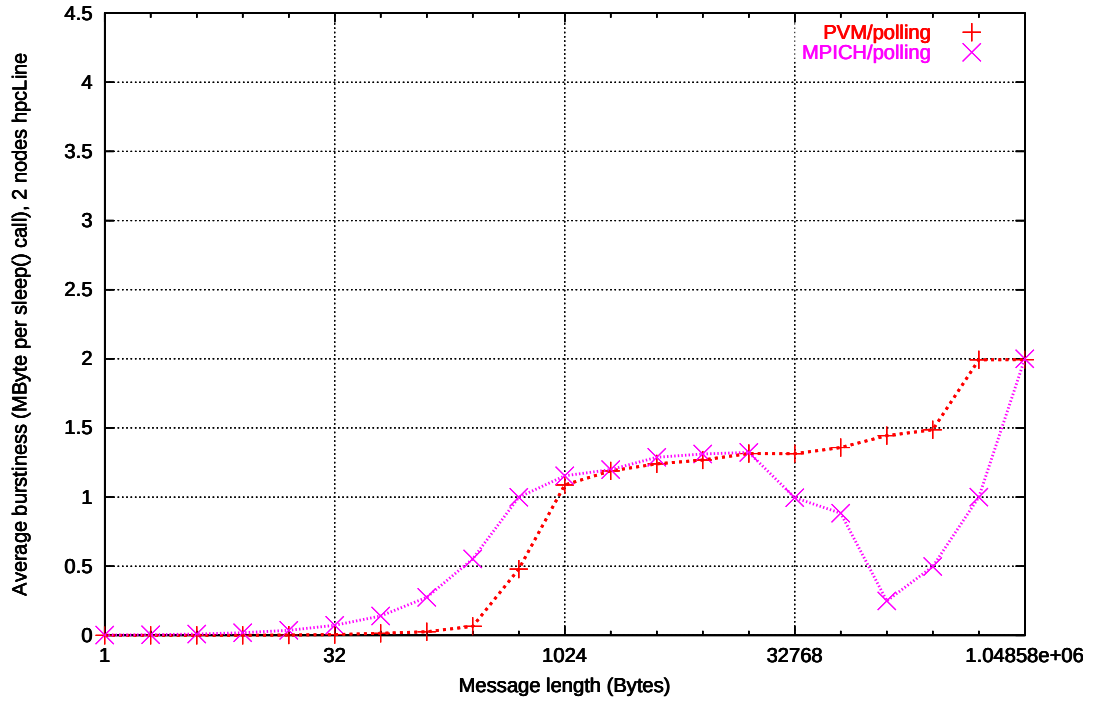

Figure 2.32: Average burstiness (amount of transferred data per `sleep()` call) for the polling versions of the benchmark, 2 nodes hpcLine

The event-driven `PVM/TPL` clearly outperformed `PVM/polling` as regards stability. `PVM/polling` behaved very non-deterministically on 1 as well as on 2 nodes of hpcLine.

`MPICH/polling` was surprisingly very deterministic. However, for certain message sizes there is a falloff in `MPICH/polling`'s performance. Even though `MPICH/TPL` was outperformed by `MPICH/polling` for small message sizes, it did not exhibit any falloffs. Also, `MPICH/TPL` was handicapped in this benchmark because of the polling inside the MPICH library (see Section 2.6.3) which negatively influences the performance of the event-driven mechanism.

We must stress that both benchmarks (one-sided as well as symmetrical) are the best possible representatives of non-trivial applications for the polling communication libraries as it produces continuous message flows that minimise the number of expensive `sleep()` calls in the polling loop. An application which does not produce continuous message flows leads to the calling of `sleep()` after each received message in the polling loop, which drastically decreases performance. In such a case the latency of servicing a request will always be at least 0.02 seconds if polling is used (the stream of requests is likely to be non-continuous). The latency of 0.02 seconds is much higher than the average latency of the event-driven versions of the benchmarks. The latency of the event-driven mechanism does not depend on the rate with which requests are generated. Therefore, if

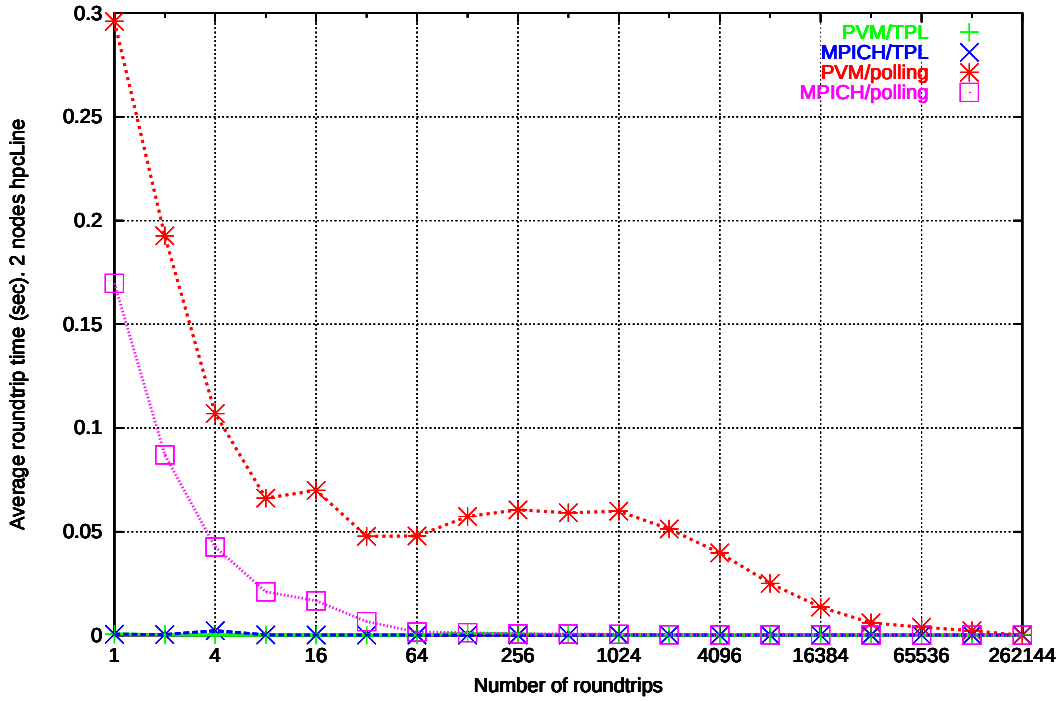

Figure 2.33: Average roundtrip time, 2 nodes hpcLine

the event-driven mechanism is not worse than polling on these benchmarks, it can only perform better in a real non-trivial application which produces many non-continuous message flows.

The event-driven mechanism outperformed polling in terms of message passing latency by two orders of magnitude. The minimisation of latency is essential for decreasing idle periods in processes of many non-trivial applications.

Remark. There is one additional factor which determines the latency of request servicing of the even-driven implementation—*thread scheduling policy*. The following thread scheduling policy is optimal (yielding minimum latency) for the concept depicted in Fig. 2.20. This thread scheduling policy is almost identical to the Transputer's scheduling policy described in Section 2.2.2:

- The main thread cannot be preempted by any other thread.
- The remaining threads $Thread_1 \ldots Thread_N$ are scheduled using any policy, for example round-robin. It is not very important whether the scheduling of the threads $Thread_1 \ldots Thread_N$ is preemptive or not, but the non-preemptive scheduling is a more rigorous choice because it avoids an unnecessary context switching.
- The main thread has a higher priority than $Thread_1 \ldots Thread_N$. This

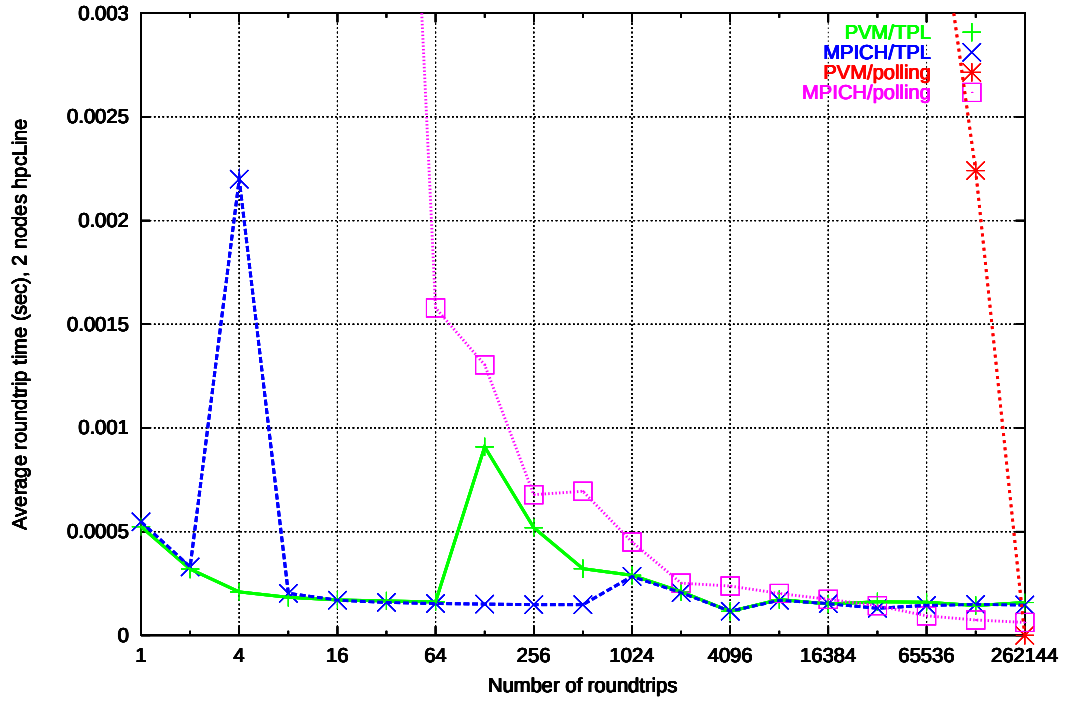

Figure 2.34: Average roundtrip time, 2 nodes hpcLine (a 100x magnification of the graph from Fig. 2.33)

means that the threads $Thread_1 \ldots Thread_N$ are only scheduled when the main thread is blocked. The main thread is immediately scheduled when it unblocks, possibly preempting another running thread (if there is one).

The POSIX standard defines thread scheduling policies which should be supported by operating systems: `SCHED_FIFO` (run to completion), `SCHED_RR` (round-robin), ... However, many operating systems (Solaris, Linux, Ultrix) restrict the choice of the scheduling policy to `SCHED_RR`. Other thread scheduling policies are either not implemented or require super-user privileges. The reason for this is that thread scheduling is mixed with process scheduling in the implementations of the operating systems which are designed for shared-memory multiprocessors.

The round-robin thread scheduling policy is a source of increased message servicing latency in TPL. Unfortunately, this is a problem that cannot be solved without a change to the operating system's scheduler. •

Open questions remain:

- Under what circumstances does the network (more precisely the transport protocol, TCP in this case) produce a continuous stream of messages?

- What are the optimal settings of TCP or other transport protocols for the event-driven mechanism of TPL?

- Would the use of other existing transport protocols significantly influence the answer to the first question?

2.9 Conclusions

We showed that the existing message passing standards such as PVM, MPI and CORBA, do not allow an efficient implementation of a large class of important applications which we refer to as non-trivial. All irregular applications belong to this class, including so-called grand-challenge problems. (Practically every larger parallel application belongs to this class.) The source of inefficiency is polling. Inefficiency is not the only drawback of polling. Further drawbacks include a destruction of the natural structure of the program code, a high execution time variation of the same program on the same input on the same system, limited flow control etc.

We proposed a formal message passing framework which is compatible with existing fundamental abstract models for parallel processing. This framework defines the structure and behaviour of any system which implements message passing but it is very flexible at the same time—it does not dictate whether the physical system architecture is shared or distributed memory; it does not dictate whether the programming language is functional, logical or imperative; it does not exclude fault-tolerance; etc. A similar framework exists for database systems and it is well accepted by all implementors of database systems.

To the best of our knowledge, this is the first formal framework which adheres to the existing abstract message passing models and which also covers practical issues of parallel processing. The reason why non-trivial applications cannot be implemented efficiently in MPI and CORBA is that these standards do not fit into our framework. This makes these standards incompatible with formal message passing models such as Hoare's CSP or Andrews' channel model. Moreover, MPI and CORBA do not define asynchronous message passing, even though they claim that they do. Interestingly, PVM does fit into our framework—however, its operation binding (the current implementation of PVM) does not cover asynchronous communication.

TPL, our message passing library, is a straightforward materialisation of our message passing framework. We implemented the operation binding for clusters with distributed-memory nodes. However, the language primitives defined in TPL are system-independent. This means that a program written using the TPL library will run without a change in the source code on a shared-memory architecture as well (only the operation binding must be added to the implementation of the library). The TPL library is thread-safe and portable on the POSIX

level which is supported by all contemporary platforms. The interface of TPL is very small—it only consists of 11 functions (plus 24 functions for message assembling and disassembling). However, it efficiently (without polling) implements asynchronous communication, one-sided communication, message handlers, active messages, flow control, support for heterogeneous systems etc. We are aware of no communication library which would cover all these features without polling or without a loss of portability.

The current implementation of TPL is based on quasi-thread-safe PVM 3.4 and MPICH 1.2.4-ch_p4. These quasi-thread-safe libraries are the original libraries extended with a novel interrupt mechanism. Quasi-thread-safety is already sufficient for the implementation of non-trivial application without polling. The choice of the underlying library is not very important—a direct use of a socket library instead of PVM or MPICH would be even more efficient. However, our intention was to demonstrate the flexibility of TPL. An application written in TPL can be linked with the quasi-thread-safe PVM or the quasi-thread-safe MPICH (or any lower level library which is thread-safe or at least quasi-thread-safe) without a change in the application's source code. We are aware of no other communication library which can do this.

TPL outperforms the standard PVM and MPI implementations (PVM 3.4 and MPICH 1.2.4-ch_p4) on a standard cluster platform by two orders of magnitude on an irregular benchmark. This benchmark, a threaded pingpong, is derived from our definition of a non-trivial application—in other words, the structure of this benchmark directly corresponds e.g. to the structure of grand-challenge problems. All implementations of this benchmark using the standard PVM 3.4 and MPICH 1.2.4-ch_p4 libraries are forced to use polling.

In order to illustrate the importance of the results above, the following short section presents a short case study which presents mechanisms of the MPI model for overlapping of communication and computation. We compare these mechanisms to the mechanisms of TPL.

2.9.1 Overlapping of Communication and Computation

The MPI model for overlapping of communication and computation is described in the PhD thesis by R. P. Dimitrov, "Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations" [Dim01] in Section 3.2.2 "Statement of Model and Definition of Parameters". One of the goals of the cited work is the minimisation of the effective overhead of "asynchronous communication" defined in the MPI standard by a proper delaying of completion synchronisation. (We claim that MPI does not define message passing, unless MPI's "asynchronous communication" is removed from the specification, therefore the quotation marks.)

Dimitrov's work is technically correct *with respect to the MPI model*. However,

some of the assumptions used throughout the work either do not hold or are irrelevant in models which define message passing.

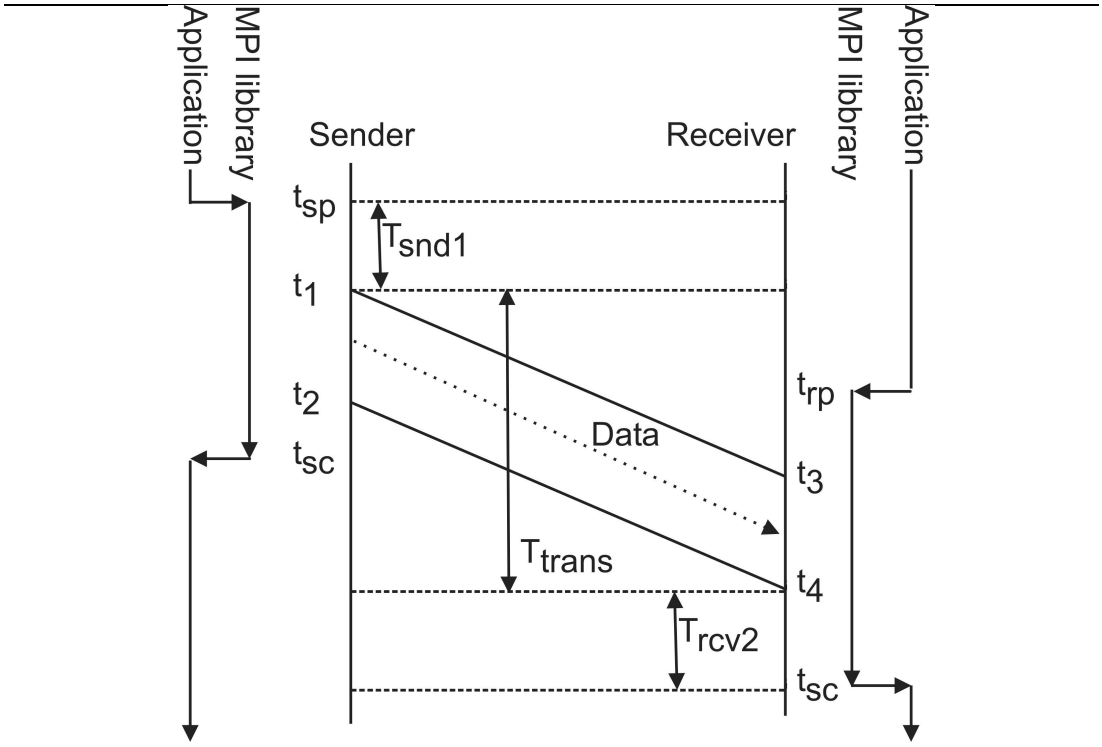

Figure 2.35: Overhead and transmission time in the MPI model. t_{sp} denotes the moment at which the sender posts a send request. t_1 denotes the moment when the first byte of the message is placed on the network. t_2 denotes the moment when the last byte of the message is placed on the network. t_{sc} denotes the moment when the application is notified about the completion of the send request. t_{rp} denotes the moment when the application posts a receive request (which matches the send request). t_3 denotes the moment when the first byte of the message arrives. t_4 denotes the moment when the last byte of the message arrives. t_{rc} denotes the moment when the receiver is notified about the completion of the receive request

In Section 3.2.2, pp. 93–94, Dimitrov explains the idea behind the non-blocking `MPI_Recv` (the notation is explained in Fig. 2.35):

> "Once the non-blocking receive request is posted, the message-passing middleware typically does not perform any processing on this request until a matching message arrives (i.e., until t_3). According to this scenario, T_{rcv1} will simply be shifted in time, and, as a result, the application can perform other computation or communication in the period between the moment when the request is posted and t_3.

> Therefore, this shift of the moment of request posting will not result in an effective overhead increase. In fact, this behavior is encouraged by most MPI implementations because the first component of the receive overhead can be shifted to earlier parts of the parallel algorithm which are not overhead sensitive (e.g., in the initialization phase of the algorithm). ... This will decrease the effective receive overhead incurred by the user process and improve the opportunities for overlapping of communication and computation. These opportunities are further enhanced by message-passing middleware that supports independent message progress. If such a service is available, the user process may delay the completion synchronization (e.g., `MPI_Wait`) until a moment after t_4, which would enable this process to effectively overlap the entire transition time with other activities."

No implementation of MPI may violate the progress rule which is defined in the MPI standard—for instance, MPICH does not comply to the MPI standard (see Section 2.6.3). More generally, if the message passing middleware does not support independent message progress, then no MPI implementation exists which builds on such a middleware.[37] (Nevertheless, these remarks are purely academical because practically all contemporary message passing middlewares do support threading and therefore independent message progress. A similar middleware was already provided by INMOS Transputers.)

The kind of overlapping of communication and computation which is described in the quoted text only applies to regular (trivial) applications. The reason for that is that is impossible to insert the completion synchronisation (`MPI_Wait()`) anywhere into a program with an unpredictable flow of control but at the program's end. This is not desirable because then the lower bound of the latency of the receive perceived by the program will be equal to the execution time of the whole program.

Even if the application is regular—that means, it consists of communication and computation phases which are strictly separated in the program—it does not know when the matching message actually arrives. Therefore it can be only guessed how much computation can be inserted between the posting of a non-blocking request (`MPI_Recv()`) and the completion synchronisation (`MPI_Wait()`). This guess is system-dependent, which means that the application must be tuned after it has been ported to another system or when certain parameters of the system change. This tuning may sometimes require structural changes in the application which is very undesirable.

[37] The interpretation of the progress rule has been a very hot topic in the MPI Forum for many years. According to statements of MPI developers in public forums, the interpretation of the progress rule is still unclear. In order to clarify this, we defined a formal framework which adheres to well-accepted abstract message passing models such as the channel model by Andrews and which clearly defines point-to-point communication.

TPL does not offer non-blocking receive for a simple reason—it is not needed. If the application can proceed in the computation without receiving a message, it does not post any receive. If it cannot proceed, it blocks and waits for that message. It is possible that the message has already arrived at the time of posting a blocking receive, in which case the waiting is not needed. We recall that TPL does make use of the middleware which supports independent message progress and buffering at receiver (see Section 2.7.8). Therefore, if the matching message has already been sent to this process (if the send request has already been posted by the sender), it has either been received or is being received by this process. TPL provides an automatic overlapping of computation and communication. An application written in TPL needs no tuning after it has been ported to another system. The strategy which is described in the quotation can only outperform TPL in the case when the guess of the completion synchronisation point (the insertion of MPI_Wait() into the program) was correct. However, if this synchronisation point can be precisely guessed, then a blocking receive (`MPI_Recv()`) at the synchronisation point can be used instead of the `MPI_Irecv()` and `MPI_Wait()` pair without a significant loss of efficiency. This implies that TPL can only be outperformed in the case when the application exhibits absolutely predictable communication patterns and therefore does not need asynchronous communication at all. (This situation is very rare because also theoretically regular communication patterns are perturbed by external factors in real systems, e.g. by I/O operations, process scheduling etc.)

Dimitrov further explains the idea behind the non-blocking `MPI_Isend`, and defines an objective of his study, pp. 94:

> "The send process can achieve effective overlapping of communication and computation, similarly to the receive process, by shifting the synchronisation procedure of the send request to a later moment, and scheduling computation activities immediately after the send request is registered with the MPI library (i.e., after t_1). This can effectively hide the transmission time (assuming sufficient memory bandwidth) and also move the notification overhead to a non-time-critical segment of the algorithm. The actual benefit of overlapping depends on the capabilities of the computer platform, on the network infrastructure, and the communication software. A main objective of this study is to reveal the factors that affect overlapping, how overlapping efficiency can be improved, and how parallel algorithms can take advantage of overlapping."

No abstract message passing model (except of modern "message passing systems" such as MPI or CORBA) requires the synchronisation procedure of the send request (e.g. `MPI_Wait()`), see Section 2.5.5. This "feature" alone makes the modern systems incompatible with fundamental abstract message passing models. This "feature" obviously does not mean an improvement of the fundamental

models as it forces polling in all irregular applications which use non-blocking send. A further study of this semantics is therefore irrelevant.

TPL does not require any synchronisation procedure. It makes no sense to distinguish between blocking or non-blocking send in TPL because TPL's buffering at receiver guarantees that no send will block forever.

Chapter 3

Global illumination

The goal of rendering is to provide an observer who is watching a computer screen with the same sensation as if the observer was watching a real 3D scene on the screen. The image on the screen is computed from a *3D model*. The 3D model consists of the *geometry* of all 3D objects, the *material properties* of the 3D objects and the properties of the *light sources* which illuminate the scene. The model also contains a description of the virtual *camera* which takes a picture of the scene. The picture of the 3D scene as seen by the camera appears on the screen. In informal terms, the global illumination problem involves computing this picture from the information stored in the 3D model.

The light distribution in the scene does not depend on the camera. Even if the observer is missing, the light is distributed in the scene. The solution to the global illumination problem can therefore be divided into two independent phases:

1. Computation of the light distribution in the 3D scene
2. Measurement of the light distribution by the virtual camera

The first phase may simulate the laws of physics [FLS70] in order to compute the distribution of light in the scene. This involves the simulation of the *physics of light*—interactions of light with the objects and also with the media between them. Note that if this phase is separated from the second phase, then the computed illumination must also be stored. On the other hand, it can be assumed during the computation of the first phase that the picture computed in the second phase will be viewed by a human eye. We already know that the human eye is sensitive to *radiance* and therefore a sufficient product of the first phase provides a knowledge of radiances for all surface points and all directions in the 3D scene.

The second phase deals mostly with the human perception of *colour*. The human eye is a *device* which measures the spectral energy distribution of impinging light. This energy distribution is a function which is defined on the wavelength which ranges from ca. 400 nm to ca. 700 nm. The impinging light seen by

the virtual camera must be reproduced by a physical device (computer monitor, glasses, paper) in order to create the impression of "looking at the 3D scene". The contemporary display devices are not able to display all energy distributions. However, the human eye is also not able to distinguish between many different energy distributions and therefore the simplified models used for the colour synthesis in physical devices (*tone mapping*) are usually sufficient.

3.1 Physics of light

At a macro-level, light behaves as an electromagnetic wave. Light has all the usual properties of waves, such as bending around obstacles and interference. Unlike sound, light also propagates in the vacuum (sound is not an electromagnetic wave). However, not all phenomena can be explained by the wave theory. A simple example is the refraction of light. [Fey88] Newton was not able to explain refraction, even though he attempted to model light with particles. It turned out that some phenomena can be explained using the wave theory, whereas other phenomena must be explained using the particle theory (wave-particle duality). However, it continued to remain unclear as to which cases and why light sometimes behaves as a wave and sometimes as particles. It took several centuries until the quantum electrodynamics theory was developed by Maxwell, Einstein, Feynman and others. [FLS70] Quantum electrodynamics is to date the best existing theory which satisfactorily explains all light phenomena. At a micro-level, a quantum of energy is transported by a particle called a *photon*. This is the reasoning from [Fey88]:

> We know that light is made of particles because we can take a very sensitive instrument that makes clicks when light shines on it, and if the light gets dimmer, the clicks remain just as loud—there are just fewer of them. Thus light is something like raindrops—each little lump of light is called a photon—and if the light is all one color, all the "raindrops" are the same size.

A single photon is itself a "wave" with some frequency. The length of the "wave" *lambda* of a single photon can be computed if its frequency is known:

$$\lambda = \frac{c}{f} \tag{3.1}$$

where c is the speed of light which is constant ($c \approx 2.99 \cdot 10^8$ m/s). The energy E transported by a single photon with the frequency f is equal to

$$E = h \cdot f \tag{3.2}$$

where h is the Planck's constant ($h \approx 6.63 \cdot 10^{-34}$ Js).

Heisenberg's uncertainty principle states that it is impossible to precisely measure both the photon's location and momentum (hence, its energy) at the same time. The photon's momentum p (note that photons have no mass) is defined as

$$p = \frac{E}{c} \tag{3.3}$$

More precisely, Heisenberg's principle states that if one makes a large number of "identical" measurements of the photon's location and momentum under the same experimental conditions, then the measurements will show surprising differences. If Δx denotes the standard deviation of the location and Δp denotes the standard deviation of the momentum measured over the set of "identical" experiments, then the following tradeoff holds:

$$\Delta x \Delta p \geq \frac{h}{4\pi} \tag{3.4}$$

The consequence of Heisenberg's principle is that photons do not (only) travel along straight lines. A photon which moves between two points, A and B (and which is heading towards B) can theoretically choose any path from A to B (also if there are no obstacles between the two points). The length of the path is only limited by the speed of the light c if the photon is supposed to get to B within a limited period of time. However, the probabilistic distribution of the paths is not uniform. A vast majority of the photons will follow a path which is close to the straight line connecting A and B. This fact is widely used in computer graphics which assumes that photons only travel along straight lines unless they interact with an obstacle. The *participating medium* (such as air or water) is often ignored in computer graphics—in other words, it is assumed that the space between object surfaces is filled with a vacuum.

Photons do not interact with each other. However, they interact with object surfaces. When a photon hits an obstacle (an object surface), one of the following events happens:

- The photon is *absorbed*.
- The photon is *reflected*. More precisely, the photon is absorbed and a new photon is emitted from the point of the incidence into the half-space above the surface (pointed to by the surface normal). The new photon carries less energy than the absorbed photon.
- The photon is *refracted*. More precisely, the photon is absorbed and a new photon is emitted from the point of the incidence into the half-space below the surface.

Reflection and refraction differ in the direction of the reemitted photon. The term *scattering* covers both reflection and refraction—a photon is scattered if it is either reflected or refracted.

Most transparent materials cause a *partial reflection* and a *partial reflection* (and a *partial absorption*) of photons. For instance, if a narrow beam of photons (a ray) is shot at a glass surface under a certain angle, then one part of the photons will be reflected, another part of the photons will be refracted and some part of the photons will be absorbed. The same experiment can also be continually repeated using individual photons with the same statistics. Only quantum electrodynamics can explain how an individual photon "makes up its mind" whether to go through the glass or whether to reflect off it (and in which direction). The photon randomly "chooses" one of the three events. The probabilities of the events generally depend on the surface material, on the photon's incoming angle and on the photon's frequency.

A simulation of a large number of photons and their interactions with surfaces and media on a micro-level is very expensive. Computer simulations of lighting bundle many photons into a *beam*. This bundling is already a simplification of real-world physics but allows for the design of algorithms which are more appropriate for lighting simulation in larger environments in a reasonable amount of time. In the following text we will define several notions which are useful in the simulations.

Remark. Whereas a single photon has a single frequency, a beam can contain photons with many different frequencies. Therefore it makes sense to refer to the *light spectrum* of a beam, which is an energy histogram over an interval of (visible) frequencies. •

Definition. The *solid angle* subtended by a 3D surface, viewed from a point x is the area of the projection of the surface onto the unit sphere centered at x. •

The solid angle is measured in steradians (sr) and ranges from 0 to 4π sr. The solid angle is a 3D analogy of the angle in 2D subtended by a curve from a point x (which is the length of the arc of the projection of the curve onto the unit circle centered at this point).

The *differential solid angle* $d\omega$ subtended by a differential surface with area dA and viewed from a point at the distance r is equal to

$$d\omega = \frac{dA \cos\theta}{r^2} \tag{3.5}$$

where θ is the angle between the surface normal and the direction from the point to the surface.

A *direction* in 3D can be expressed using two angles (θ, ϕ). A *differential solid angle* $d\omega$ *around a direction* (θ, ϕ) is equal to

$$d\omega = \sin\theta \; d\theta \; d\phi \tag{3.6}$$

Definition. The *light flux* Φ is the amount of energy which passes through a boundary per unit time over a given range of spectrum. •

The spectrum range in the flux definition is usually an interval of wavelengths $[\lambda_{min}, \lambda_{max}]$. As it is more convenient to talk about energy radiated around a direction rather than through a boundary, another measure is defined:

Definition. *Radiance* (*intensity*) L is the amount of energy which travels at a given point in a given direction, per unit time, per unit area perpendicular to the direction of travel, per unit solid angle (over a given spectrum range). •

From its definition, radiance is the flux leaving a differential area around a given point, which leaves the point in a differential solid angle around a given direction:

$$L(x, \omega) = \frac{d\Phi(x, dA, \omega, d\omega)}{dA \; d\omega \; \cos\theta} \tag{3.7}$$

where x is the given point, ω is the differential angle around the given direction, $d\omega$ is a differential angle around the given direction, θ is the angle between the surface normal at the given point and the given direction.

Definition. We denote $L_i(x, \omega')$ the *incoming radiance* which impinges at the point x from the direction ω'.[1] •

Remark. The physical measures such as radiance, incoming radiance, ... are defined for a wavelength or a range of wavelengths. Equation 3.7 should therefore be written as

$$L^{\Lambda}(x, \omega) = \frac{d\Phi^{\Lambda}(x, dA, \omega, d\omega)}{dA \; d\omega \; \cos\theta} \tag{3.8}$$

where Λ denotes a range of wavelengths. We shall omit the superscript Λ for the remainder of the text.

The usual practice in computer graphics is to write similar equations for three representative wavelengths (such as R, G and B, see Section 3.2.1) and to work with the three equations independently of each other. This means that some phenomena such as fluorescence (which occurs when a photon hits a surface and a new photon is reemitted at a different wavelength) cannot be correctly simulated. •

[1] In order to simplify the notation, we refer to differential solid angles as directions.

Definition. The *Ray-Trace* function $RT(x, \omega)$ is a function which returns the nearest surface point to x in the direction ω. (If there is no surface point in the direction ω from the point x, $RT(x, \omega')$ returns an arbitrary point along the direction ω from the point x.) •

Radiance has a reciprocal property. For any two mutually visible points x and y, the radiance leaving the point x in the direction of y is the same as the incoming radiance at the point y from the direction of the point x. This property can be directly proven from the above definitions: Let us denote dA the differential surface around the point x and let us denote dA' the differential surface around the point y. Let us denote ω the differential solid angle around the direction from x to y and let us denote ω' the differential solid angle around the direction from y to x. Let us denote θ the angle between the surface normal at the point x and the direction ω. Similarly, let us denote θ' the angle between the surface normal at the point y and the direction ω'. Equation 3.7 can then be rearranged as

$$d\Phi(x, dA, \omega, d\omega) = L(x, \omega)\ dA\ d\omega\ \cos\theta \tag{3.9}$$

The substitution of $d\omega$ into Equation 3.9 using Equation 3.5 yields

$$d\Phi(x, dA, \omega, d\omega) = L(x, \omega)\ \frac{dA\ \cos\theta\ dA'\ \cos\theta'}{r^2} \tag{3.10}$$

The Equation 3.10 is called the *fundamental law of photometry.*

The flux which passes from x to y through any area which is a cross section of the solid angle between x and y and a plane perpendicular to the direction from x to y is the same as the flux which arrives at the point y from the direction of the point x (as both fluxes pass through the same boundary):

$$d\Phi(x, dA, \omega, d\omega) = d\Phi(y, dA', \omega', d\omega') \tag{3.11}$$

As the direction of the flux which passes between the differential areas around the points x and y is not important, an equation similar to Equation 3.10 can be derived for the reciprocal flux:

$$d\Phi(y, dA', \omega', d\omega') = L_i(y, \omega')\ \frac{dA\ \cos\theta\ dA'\ \cos\theta'}{r^2} \tag{3.12}$$

From Equations 3.10, 3.11 and 3.12 it follows

$$L(x, \omega) = L_i(y, \omega') = L(RT(x, -\omega'), \omega') \tag{3.13}$$

where $-\omega'$ is the direction opposite to ω' (in this case $-\omega' = \omega$). Consequently, radiance $L(x, \omega)$ does not depend on the distance between the points x and y. This is the reason why the human eye and photographic cameras (which are

sensitive to radiance) perceive the same colour at all viewing distances when they observe a point from the same angle.

Definition. *Radiosity* B is the total energy which leaves a differential area around a given point, per unit area, per unit time (over a given range of spectrum). •

Hence,

$$B(x) = \int_{\Omega} L(x, \omega) \ \cos\theta \ d\omega \tag{3.14}$$

where x is the given point, Ω is the space of all directions leaving x, θ is the angle between a direction ω and the surface normal at x.

3.2 3D modeling

The modeling of real 3D scenes usually makes further simplifying assumptions. Further approximations are required by the algorithms which compute the illumination in the scenes. Some of these approximations are only necessary for the computation of a reasonable illumination in a reasonable amount of time (some applications require real-time) on the current hardware. As the computing power grows, it is important to avoid using the approximations which are "hard-wired" in an algorithm and which cannot be eliminated later unless the algorithm is changed.

3.2.1 Modeling of colour spectrum

Spectrum is an energy histogram over a wavelength interval. A discrete representation of real functions usually involves a sampling of the interval. The sampling used in computer graphics applies the fact that the human eye is an imperfect spectrometer. The whole visible spectrum can be represented using three numbers, R, G and B (red, green, blue). [FvDFH90] Based on physiological experiments, three basis functions (*colour matching functions*), $r(\lambda)$, $g(\lambda)$ and $b(\lambda)$ are defined on the entire visible range of wavelengths. Any spectral function $C(\lambda)$ is approximated as a linear combination of these basis functions:

$$C(\lambda) = R \ r(\lambda) + G \ g(\lambda) + B \ b(\lambda) \tag{3.15}$$

When given a spectrum $L(\lambda)$, the coefficients R, G and B can be computed as

$$R = \int_{\Lambda} L(\lambda) \ r(\lambda) \ d\lambda, \ G = \int_{\Lambda} L(\lambda) \ g(\lambda) \ d\lambda, \ B = \int_{\Lambda} L(\lambda) \ b(\lambda) \ d\lambda \tag{3.16}$$

where Λ is the range of visible wavelengths.

Furthermore, the human eye cannot distinguish between colours which have only slightly different RGB coordinates. This allows a relatively coarse sampling of the RGB coordinates. The values R, G, B are usually represented as integers from 0 to 255 (1 Byte).

Remark. There are other colour models such as CIE XYZ, CMYK, HSV etc. [FvDFH90] However, a conversion exists between any two of these. •

The use of any of the above colour models in a lighting simulation algorithm introduces the assumption that light is monochromatic. The polarisation of light is also ignored.

3.2.2 Modeling of surface geometry

The surfaces in a 3D scene are usually divided into disjoint parts, called *objects*. This division can be hierarchical—an object can consist of smaller objects etc. The objects at the bottom of this hierarchy are surfaces which are of the same material.

The surface geometry describes the shape of a 3D surface. There are two principal approaches to the description of surface geometry:

1. *Polygonal representation* (triangle mesh).

2. *Constructive solid geometry* (CSG).

Polygonal representation

In the polygonal representation, a surface is modeled as a union of polygons, usually triangles. The majority of the triangles are bordering with three other triangles. *Triangle mesh* is a data structure which uses this fact in order to save space which is required for storing the triangles:

$$\mathcal{V} = \{v_1, \ldots, v_{maxv}\},\ v_i \in \mathcal{R}^3 \tag{3.17}$$

is a set (array) of vertices,

$$\mathcal{T} = \{t_1, \ldots, t_{maxt}\},\ t_i = \langle v_a, v_b, v_c \rangle_i,\ a, b, c \in \{1, \ldots, maxv\} \tag{3.18}$$

is a set (array) of triangles. The triangle "vertices" in the set $\mathcal{T}$ are indices to the vertex set $\mathcal{V}$. A triangle mesh can be extended for the storing of additional information. In particular, surface normals are often stored in the vertices in

order to smooth the discontinuities of the normals on the edges of neighbouring triangles.[2] This extension only requires to store an array

$$\mathcal{N} = \{n_1, \ldots, n_{maxn}\},\ n_i \in \mathcal{R}^3 \tag{3.19}$$

where n_i is a surface normal in the vertex v_i. The surface normal at a point inside a triangle can be linearly interpolated using the surface normals stored in the triangle vertices. [Pho75], [Bli78] It is generally agreed that a surface normal always points to the half-space which is outside the surface.

Remark. Some 3D formats or programs which export surface geometry to a 3D format do not allow for the storing of the surface normal information. A common problem involves then distinguishing which side of a triangle is inside and which side is outside of the triangle—in other words, the orientation of the surface normals is unclear. A common solution to this problem involves using the order of the vertices in $\mathcal{T}$ to implicitly determine the direction of the normal for a given triangle t_i—unless the normals are explicitly provided in the array $\mathcal{N}$, the direction of the surface normal corresponds to the direction of the vector product of the three vectors (vertices) indexed by t_i.

Note that a triangle is independently illuminated from its front and back sides. If the mesh structure is extended so that it stores the illumination of the triangles in the triangle vertices, then this information must be stored separately for the front and back sides of the triangle. However, a common practice is not to store the illumination for the inner surfaces of objects such as balls as it is not assumed that the inside of a ball may be of any importance. While this is true, the simplified modeling sometimes leads to unexpected problems during the computation of the illumination and during the visualisation of the illuminated scene. •

One advantage of this representation is that the surface can be parameterised, which is useful for texture mapping (the so-called uv-mapping requires a parameterisation of the surface). [FvDFH90] Another advantage is that this representation is supported by the hardware of the contemporary graphics cards, 3D scanners and other physical devices. A practically arbitrary surface representation can be converted into a triangle mesh—such a conversion is called a *tessellation*. The polygonal representation is also supported in practically all existing 3D formats.

The most serious disadvantage of the polygonal representation is that triangle meshes are only approximations of curved surfaces. The more triangles are stored

[2] This smoothing is desirable for the modeling of curved surfaces such as spheres. However, the smoothing must be avoided for box-like surfaces which actually contain sharp edges, such as a table or a cigarette box. A common workaround involves assigning a so-called crease-angle to an object (or to the entire scene). Normals of neighbouring triangles are then only smoothed if the angle between the natural normals of the triangles does not exceed the given crease-angle.

in a mesh, the better the approximation—however, the resolution of the mesh must be fixed when the scene is being stored in a file at the latest. The chosen resolution may not be sufficient later (for instance, when the surface is viewed from a small distance, the discontinuities which were neglectable before may become visible). However, the resolution cannot be increased further once it has been fixed, see Fig. 3.1.

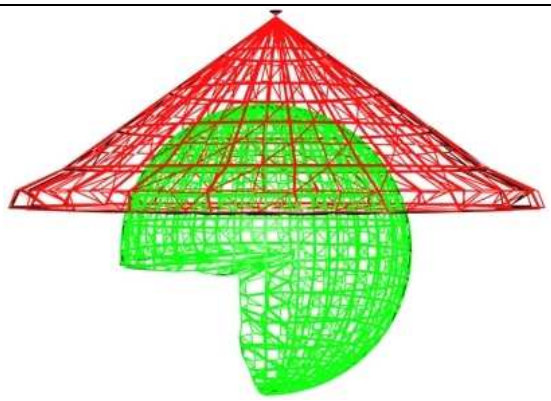

Figure 3.1: An example of a triangle mesh. Note the discontinuities on the top and on the bottom of the cone

Constructive solid geometry

Constructive solid geometry (CSG) is a modeling methodology which allows us to combine basic 3D surfaces (geometric primitives) in order to create more complex ones using boolean set operations. The following binary operations are used to combine two geometric primitives: *union*, *intersection* and *difference*. Unary operations which can be applied to any surface are *inverse* (which is usually used together with intersection in order to avoid the definition of infinite surfaces) and *transformation* (rotation, scaling and translation or any combination of these).

A CSG object can be stored as a tree. The leaves of the tree store the geometric primitives (e.g. spheres, cones, boxes, ...), other nodes store the operations. An example of a CSG tree is depicted in Fig. 3.2.

It is very important to note that *an algorithm which computes all intersections of a line with a CSG object* exists (provided that the line-object intersections can be computed for all geometric primitives used in the CSG tree which defines the object). *The surface normal of a CSG object can also be computed at any surface point* (if it can be computed for all the geometric primitives). [GN71], [Jan86] The two methods which compute the intersections with a line and the surface normals are known for many object primitives. These object primitives include planes, quadrics, blobs, bézier surfaces, sweep surfaces, polygons, height fields etc. [Gla89]

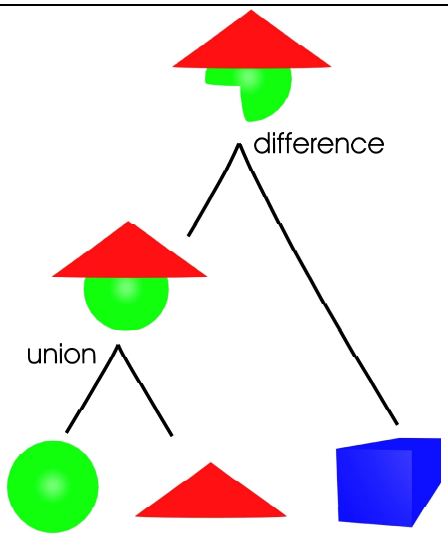

Figure 3.2: An example of a CSG tree. The object shown in the root node of the tree is a result of the union and difference operations. The unary transformation operations are not depicted in the figure (a transformation is applied to each node of the tree)

Remark. The problem with surface normals as mentioned for the polygonal representation must also be solved for the CSG representation. Hence, the computation of normals for the CSG primitives must include the computation of the orientations of the normal vectors using an agreed method. •

3.2.3 Modeling of surface materials

Surface material describes the scattering properties of the surface. Light scattering depends on the microstructure of the surface which is usually not included in the model of the surface geometry. The scattering properties are described using so-called material which is assigned to the surface geometry.

Generally, a ray of light which hits a surface enters the surface and then leaves the surface from a different location. This so-called *sub-surface scattering* is usually ignored and replaced by a model which assumes that the incident ray of light leaves the surface at the point of incidence:

Definition. *The bidirectional scattering distribution function*, $BSDF$ is defined as the ratio of the scattered radiance and incoming radiance:

$$BSDF(x, \omega', \omega) = \frac{dL_s(x, \omega)}{dE_i(x)} = \frac{dL_s(x, \omega)}{L_i(x, \omega') \ \cos\theta' \ d\omega'} \tag{3.20}$$

where x is the point of incidence, ω is a differential solid angle around the outgoing direction, ω' is a differential solid angle around the incoming direction, θ' is the angle between the surface normal and the incoming direction. The subscript of the outgoing radiance L_s underlines the fact that L_s is only the part of the outgoing radiance due to the scattering of the incoming light (the surface at the point x can also emit light in which case the outgoing radiance for a given direction is the sum of the emitted and scattered radiances). •

Remark. Note that $L_s(x, \omega)$ depends on the incoming radiances $L_i(x, \omega')$ from all directions ω'. $\frac{dL_s(x,\omega)}{d\omega'}$ fixes the incoming direction to one particular direction of interest ω'. •

Remark. $BSDF$ covers both the reflection and refraction of light. $BSDF$ is defined for all incoming directions ω' and outgoing directions ω around the point x. •

The following equation, called the *scattering equation*, describes the *local illumination model*. [CW93] If the incoming radiance $L_i(x, \omega')$ is known for all incoming directions ω', then the scattered radiance in the direction of interest ω can be computed as (this follows from Equation 3.20)

$$L_s(x, \omega) = \int_\Omega BSDF(x, \omega', \omega)\ L_i(x, \omega')\ \cos\theta'\ d\omega' \tag{3.21}$$

There are physical constraints on $BSDF$. A surface cannot reflect more light than it receives (Equation 3.22 and Equation 3.23). Furthermore, the reciprocal property also applies to $BSDF$ (Helmholtz's principle, Equation 3.24):

$$\int_\Omega BSDF(x, \omega', \omega)\ \cos\theta'\ d\omega' \leq 1,\ \forall\omega \in \Omega \tag{3.22}$$

$$\int_\Omega \int_\Omega BSDF(x, \omega', \omega)\ L_i(x, \omega')\ \cos\theta'\ d\omega'\ d\omega \leq \int_\Omega L_i(x, \omega')\ d\omega' \tag{3.23}$$

$$BSDF(x, \omega', \omega) = BSDF(x, \omega, \omega') \tag{3.24}$$

The $BSDF$ function can be directly represented as a set of values defined for sampled surface points and incoming and outgoing directions. However, such a representation would probably consume a lot of memory. In practice, $BSDF$ is described using a set of parameters of a chosen reflection model. Commonly used reflection models were proposed by Gouraud [Gou71], Phong [Pho75], Torrance-Sparrow [TS67], Blinn [Bli77], Schlick [Sch93] and others. We will briefly introduce the Phong model.

Phong reflection model

In the Phong reflection model, the reflective material properties are described by four scalars k_d (diffuse coefficient), k_s (specular coefficient), k_a (ambient coefficient) and s (shininess). The model does not actually define the $BSDF$—it replaces Equation 3.21 with another one. [Pho75] The scattered radiance $L_s(x, \omega)$ is expressed as (we generalise the original Phong formula slightly)

$$L_s(x,\omega) = k_a \int_{\Omega \setminus \Omega_L} L_i(x,\omega') \cos\theta' \, d\omega' + \int_{\Omega_L} L_i(x,\omega') \left(k_d \cos\theta' + k_s \cos^s \alpha\right) d\omega' \tag{3.25}$$

The integration domain is split into two parts. The part Ω_L denotes all incoming directions from a light source to the point x which are not blocked by any other object. In other words, Ω_L is a set of directions from which the point x is directly illuminated.

The Phong model is purely empirical. Its parameters have no physical meaning. The splitting of the integration is already wrong—the real $BSDF$ function makes no distinction between direct and indirect incoming light (the surface material has no means of distinguishing between direct and indirect incoming light—it reflects both in the same way).

The term $k_s \cos^s \alpha$ in Equation 3.25 depends on the position of the camera, as α is the angle between the perfectly mirrored direction of ω' around the normal at x and the viewing direction. (This is another flaw of the model—the real $BSDF$ function does not depend on the camera.) This term simulates so-called specular highlights which are caused by a direct reflection of light from a metalic surface onto the camera. The parameter k_s controls the intensity of the highlight and the parameter s controls its "tightness".[3]

The indirect illumination term $\int_{\Omega \setminus \Omega_L} L_i(x,\omega') \cos\theta' \, d\omega'$ is sometimes approximated with a constant in some illumination algorithms (local illumination algorithms).

The reason why we deal with the Phong model is that it is assumed in the majority of the existing 3D formats—in which the material description consists of the four scalars k_d, k_s, k_a and s. The description of materials is a very serious problem of contemporary computer graphics.

Modified (more realistic) Phong reflection model

Fortunately, the four parameters used in the Phong model can be given a more realistic interpretation than that of Equation 3.25. [LW94] The modified Phong

[3] We assume here silently that the surface of each object consists of the same material and that each object is assigned its own $BSDF$. It would be possible to describe the surfaces of all objects using one global $BSDF$ but in such a case the parameters k_d, k_s, k_a and s would be functions of x.

model obeys Equation 3.21 and defines the $BSDF$ function as a sum of specular and diffuse components:

$$BSDF(x,\omega',\omega) = BSDF_d(x,\omega',\omega) + BSDF_s(x,\omega',\omega) = k_d \frac{1}{\pi} + k_s \frac{s+2}{2\pi} \cos^s \alpha \tag{3.26}$$

where α is the angle between the perfect specular reflective direction and the outgoing direction. (The parameter k_a is ignored.)

This $BSDF$ model does not include light transmission. The adding of light transmission usually requires an inclusion of additional parameters such as IOR (index of refraction) and the transparency coefficients of the model (and also an inclusion of the additive term which corresponds to Equation 3.26).

3.2.4 Modeling of light sources

A light source is an area which emits light without being illuminated from the outside. A light source can be characterised by the placement and geometry of the area and by its directional radiant properties. It is usually assumed that these properties do not change over time. A light source i is characterised by its radiant emittance $l_e^i(x,\omega)$. There is a finite set of light sources in the 3D model. The whole set of light sources is described using the function $L_e(x,\omega) = \sum l_e^i(x,\omega)$.[4]

The idealised light source types widely used in computer graphics are a *point light source* and an *area diffuse light source*. The area of a point light source is a differential area around a point and the energy emitted in all directions is equal. (A simple modification of a point light source is a *spot light source* which is a differential area around a point which emits energy in a cone around the point. The emitted radiance is maximal in the direction of the cone axis and zero for the directions outside the cone.) The area of an area diffuse light source is a non-zero (usually planar) area, point of which emits energy equally in all directions.

A more realistic description of light sources is provided by the ANSI/IESNA standard "IES Recommended Standard File Format for Electronic Transfer of Photometric Data". [LM-02] Characteristics of many real luminaires (light fixtures) by various manufacturers are stored in the IES format (the description of a luminaire is essentially the radiance function sampled in many points of the luminaire in many directions). This format is being adopted by the computer graphics community. For instance, the *RADIANCE* rendering system [War94] can import light sources which are described using the IES format and work with them.

[4] We can assume that the areas of light sources do not overlap. (The area of a light source i is the set of points x for which $l_e^i(x,\omega) \neq 0$ for some ω.)

3.2.5 Modeling of camera

A camera is a device which is sensitive to radiance. The radiance is measured using a finite set of *sensors*. A sensor i is characterised by its *sensor responsiveness function* $w_e^i(x, \omega')$ which returns 1 if the radiance impinging at the point x in the direction ω' directly reaches the measuring device (e.g. a film or an eye). Otherwise it returns 0. The total response measured by the sensor i in a differential area around a point x is

$$\int_\Omega w_e^i(x, \omega')\ L_i(x, \omega')\ \cos\theta'\ d\omega' \tag{3.27}$$

where θ' is the angle between the incoming direction ω'.

The set of all sensors is characterised by the function $W_e(x, \omega') = \sum w_e^i(x, \omega')$.[5] The picture seen by the camera consists of a finite number of pixels which are organised in a rectangular 2D grid. Each pixel is covered by one sensor. The total response measured over a pixel is thus

$$\begin{aligned} & \int_S \int_\Omega W_e(x, \omega')\ L_i(x, \omega')\ \cos\theta'\ d\omega'\ dA \\ = & \int_S \int_\Omega W_e(x, \omega')\ L(RT(x, -\omega'), \omega')\ \cos\theta'\ d\omega'\ dA \end{aligned} \tag{3.28}$$

where dA is a differential area around the point x, θ' is the angle between the incoming direction ω' and the surface normal at the point x and S is the area covered by the pixel. The interpretation of this equation is: "The total response of a sensor is the radiance which directly reaches the measuring device."

3.3 The global illumination problem

An instance of the global illumination problem is a tuple

$$\langle G, BSDF, L_e, C \rangle \tag{3.29}$$

where G is a description of the surfaces, $BSDF$ is a description of the material properties of the surfaces, L_e is a description of the light sources and C is a description of the camera. The problem is to compute the values measured by the camera sensors.

[5] We can assume that the areas of sensors do not overlap. (The area of a sensor i is the set of points x for which $w_e^i(x, \omega') \neq 0$ for some ω'.)

3.3.1 Rendering equations

Radiance equation

A mathematical definition of the global illumination problem is comprised in the Equation 3.28. The calculation of the total response of a camera sensor (colour of a pixel) depends on the knowledge of the function L_i over the set of points x and directions ω' of interest (for which $W_e(x, \omega') \neq 0$).

The unknown incoming radiance L_i (or radiance L, see Equation 3.13 which relates L and L_i) can be calculated using the scattering equation 3.21. We recall that $L_s(x, \omega)$ on the left side of Equation 3.21 is the *scattered radiance* at the point x. If the point x also emits light, then the *total outgoing radiance* $L(x, \omega)$ which leaves the point x due to emission *and* scattering is equal to

$$\begin{aligned} L(x,\omega) &= L_e(x,\omega) + L_s(x,\omega) \\ &= L_e(x,\omega) + \int_\Omega BSDF(x,\omega',\omega)\ L(RT(x,-\omega'),\omega')\ \cos\theta'\ d\omega' \end{aligned} \tag{3.30}$$

Equation 3.30 is called the *radiance equation*. In order to solve the global illumination problem, Equation 3.30 must be solved (the function L must be calculated) for those x and ω which contribute to the integration in Equation 3.28 at least.

Potential equation

The global illumination problem can also be looked at from another point of view, using an abstract measure which expresses the visual importance.

Definition. *(Visual) potential* (also called *visual importance*) $W(x, \omega')$ is defined as the percentage of the incoming radiance $L_i(x, \omega')$ at the point x in the direction ω' which reaches the measuring device. •

Remark. The percentage of the incoming radiance $L_i(x, \omega')$ at the point x in the direction ω' which *directly* reaches the measuring device is equal to $W_e(x, \omega')$. •

Let us denote $W_s(x, \omega')$ the percentage of the incoming radiance $L_i(x, \omega')$ at the point x in the direction ω' which leaves the point x and *indirectly* reaches the measuring device—that means after one or more scatterings. From the definitions of $BSDF$ (Equation 3.20) and the potential it follows that

$$W(RT(x,\omega),\omega)\ BSDF(x,\omega',\omega)\ \cos\theta$$

is the percentage of $L_i(x, \omega')$ which leaves the point x in the direction ω due to scattering and then (directly or indirectly) reaches the measuring device. θ is the angle between the surface normal at the point x and the direction ω. The total percentage of $L_i(x, \omega')$ which reaches the measuring device is thus equal to

$$\begin{aligned} W(x, \omega') &= W_e(x, \omega') + W_s(x, \omega') \\ &= W_e(x, \omega') + \int_\Omega BSDF(x, \omega', \omega)\ W(RT(x, \omega), \omega)\ \cos\theta\ d\omega \end{aligned} \tag{3.31}$$

Equation 3.31 is called the *potential equation*. There is a strong structural similarity between the potential equation and the radiance equation. Indeed, the solving of the potential equation also solves the global illumination problem. If the function $W(x, \omega')$ is known, then the total response of a sensor can be calculated as

$$\int_S \int_\Omega W(RT(x, \omega), \omega)\ L_e(x, \omega)\ \cos\theta\ d\omega\ dA \tag{3.32}$$

where dA is a differential area around the point x, θ is the angle between the direction ω and the surface normal at the point x and S is the area of all the light sources. The interpretation of this equation is: "The total response of a sensor is the percentage of the emitted radiance times the percentage of this emitted radiance which eventually reaches the measuring device".

3.4 Approaches to the global illumination problem

Both the radiance equation 3.30 and the potential equation 3.31 are Fredholm integral equations of the second kind and cannot be (except for a few special cases) solved analytically. [Atk76] There are two classes of methods which numerically solve the equations:

- *Direct methods* which directly solve the integral equations. These include Monte Carlo and Quasi Monte Carlo integrations in higher dimensions.
- *Approximation methods* which make additional simplifying assumptions and solve simplified equations. These include (eye-) ray tracing and finite element methods such as radiosity.

The main advantage of the direct integration methods is that they work without approximations with the original model. This means for instance that if the integration method guarantees a certain error bound, then this error bound also applies to the computed images.

An important practical question by designing a global illumination algorithm is whether the algorithm requires an explicit storage of the radiance or the potential function over the space of all surface points and directions. As this space is infinite, *the explicit representation of the function radiance or the potential function in finite memory is already an approximation.* The error bound of such an approximation is usually difficult to predict. This means that direct methods should not rely on the explicit storing of the radiance or the potential functions in order to guarantee the error bound provided by the underlying integration method.

3.4.1 Direct methods

Gathering path integration

The radiance equation 3.30 can be regarded as a recurrent definition of the unknown function L. Let us denote $\mathcal{R}$ the *radiance transport operator*

$$(\mathcal{R}L)(x,\omega) = \int_\Omega BSDF(x,\omega',\omega)\ L(RT(x,-\omega'),\omega')\ \cos\theta'\ d\omega' \tag{3.33}$$

Using this operator, the radiance equation 3.30 can be written as

$$\begin{aligned} L &= L_e + \mathcal{R}L \\ &= L_e + \mathcal{R}(L_e + \mathcal{R}L) = L_e + \mathcal{R}L_e + \mathcal{R}^2 L \\ &\dots \\ &= \left(\sum_{i=0}^{n} \mathcal{R}^i L_e\right) + \mathcal{R}^{n+1} L \end{aligned} \tag{3.34}$$

Fig. 3.3 and Fig. 3.4 depict the geometry of the integrands of $(\mathcal{R}L)(x,\omega)$ and $(\mathcal{R}^2L)(x,\omega)$, respectively. If the operator $\mathcal{R}$ is a contraction (and it is—thanks to the underlying physics, see Equation 3.22) then $\lim_{n\to\infty} \mathcal{R}^{n+1}L = 0$. Therefore

$$L = \lim_{n\to\infty} \sum_{i=0}^{n} \mathcal{R}^i L_e = \sum_{i=0}^{\infty} \mathcal{R}^i L_e \tag{3.35}$$

The terms $\mathcal{R}^i L_e$ have the following structure:

$$\begin{aligned} (\mathcal{R}^0 L_e)(x,\omega) &= L_e(x,\omega) \\ (\mathcal{R}^1 L_e)(x,\omega) &= \int_{\Omega_1} BSDF(x,\omega_1',\omega)\ L_e(RT(x,-\omega_1'),\omega_1')\ \cos\theta_1'\ d\omega_1' \\ (\mathcal{R}^2 L_e)(x,\omega) &= \int_{\Omega_2}\int_{\Omega_1} BSDF(RT(x,-\omega_1'),\omega_2',\omega_1')\ BSDF(x,\omega_1',\omega) \\ &\qquad L_e(RT(RT(x,-\omega_1'),-\omega_2'),\omega_2')\ \cos\theta_1'\ \cos\theta_2'\ d\omega_1'\ d\omega_2' \\ &\dots \end{aligned} \tag{3.36}$$

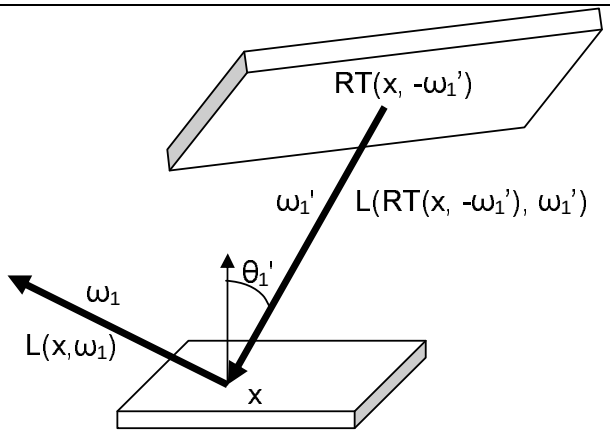

Figure 3.3: Gathering path integration: The geometry of the integrand of the term $(\mathcal{R}L)(x,\omega)$

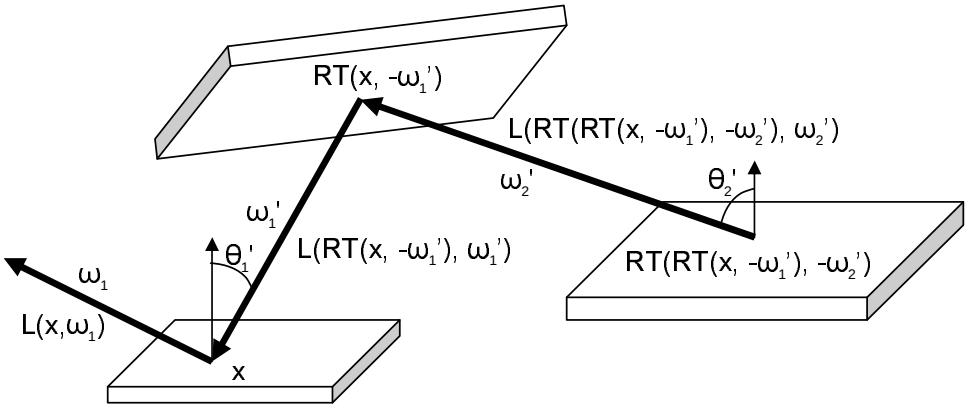

Figure 3.4: Gathering path integration: The geometry of the integrand of the term $(\mathcal{R}^2L)(x,\omega)$

These terms have a physical interpretation: "$(\mathcal{R}^iL_e)(x,\omega)$ is the radiance at the point x in the direction ω which has been scattered exactly i-times after it had left a light source (including the scattering at the point x)."

As the operator $\mathcal{R}$ is contractive (a part of the transported radiance is absorbed in each scattering), the values of $(\mathcal{R}^iL_e)(x,\omega)$ get smaller as i increases:

$$(\mathcal{R}^0L_e)(x,\omega) \geq (\mathcal{R}^1L_e)(x,\omega) \geq (\mathcal{R}^2L_e)(x,\omega) \geq (\mathcal{R}^3L_e)(x,\omega) \geq \ldots \qquad (3.37)$$

The most rigorous algorithms which use Equation 3.35 in order to solve the global illumination problem are *path tracing* [Kaj86] and *distributed ray tracing* (also called *Monte Carlo Ray Tracing*) [CPC84]. These algorithms do not explicitly store the computed radiance function—instead they directly compute the integral 3.28 using Monte Carlo integration. Both path tracing and distributed

ray tracing generate paths which consist of line segments (rays). The first ray starts at the measuring device and goes through a pixel (its direction is randomly generated). If a ray does not hit a surface, the path will be terminated.[6] Otherwise a decision is made whether the path will be prolonged by another ray (the direction of this scattered ray is generated randomly) or whether the path is terminated.[7] The probability of the prolongation of a path is proportional to an estimated contribution of the new ray to the integral 3.28 (as the radiance transport operator $\mathcal{R}$ is a contraction, this contribution decreases as the length of the path increases). When a path is terminated, its contribution to the integral 3.28 is added and the path is discarded. The difference between path tracing (Fig. 3.5) and distributed ray tracing (Fig. 3.6) is that path tracing only collects the direct lighting L_e from the light sources in the last point of the path which is being terminated, whereas distributed ray tracing collects the direct lighting in the last point of every segment of the path. (Note that the collection of the direct lighting is a necessity if the scene contains point light sources, because the probability of hitting a point light with a randomly generated ray is zero.)

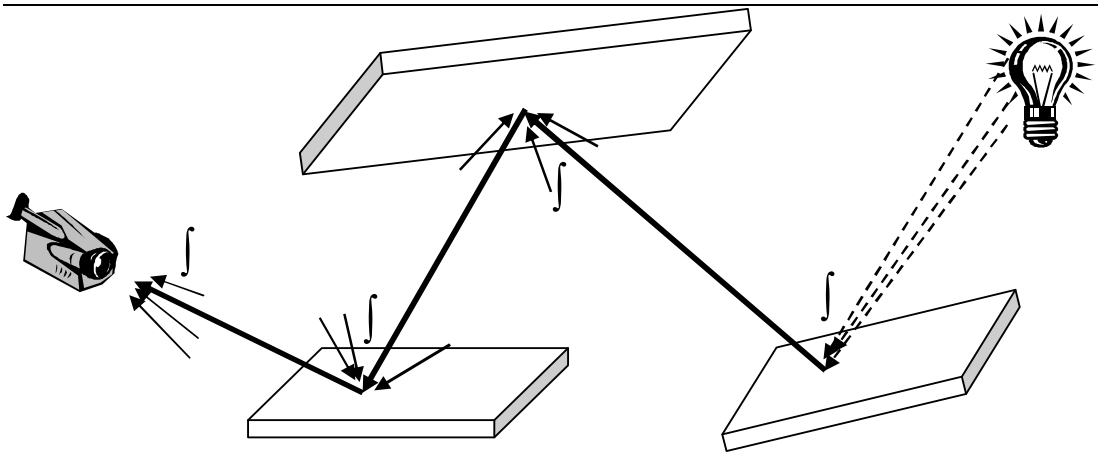

Figure 3.5: Camera tracing with a single collection of the direct radiance (path tracing)

Shooting path integration

The potential equation 3.31 can be expanded in a similar way to that of Equation 3.34. Let us denote $\mathcal{P}$ the *potential transport operator*

[6] Instead of the termination of the path in this case, the algorithm may shorten the path and recursively trace other rays.

[7] If a ray hits a surface of a light source, the path is terminated in order to avoid a multiple addition of the direct lighting. Another possibility involves excluding the surfaces of light sources from the computations of these intersections.

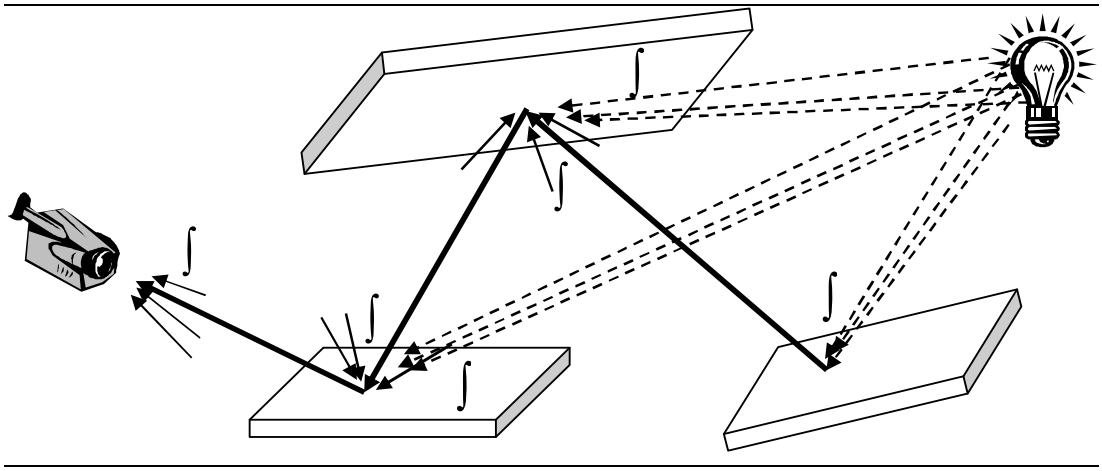

Figure 3.6: Camera tracing with a multiple collection of the direct radiance (distributed ray tracing)

$$(\mathcal{P}W)(x,\omega') = W_e(x,\omega') + \int_\Omega BSDF(x,\omega',\omega)\ W(RT(x,\omega),\omega)\ \cos\theta\ d\omega \quad (3.38)$$

The potential equation 3.31 can be written as (note that the operator $\mathcal{P}$ is a contraction)

$$\begin{aligned} P &= W_e + \mathcal{P}W \\ &= W_e + \mathcal{P}(W_e + \mathcal{P}W) = W_e + \mathcal{P}W_e + \mathcal{P}^2W \\ &\dots \\ &= \left(\sum_{i=0}^{n} \mathcal{P}^i W_e\right) + \mathcal{P}^{n+1}W \\ &= \sum_{i=0}^{\infty} \mathcal{P}^i W_e \end{aligned} \quad (3.39)$$

Fig. 3.7 and Fig. 3.8 depict the geometry of the integrands of $(\mathcal{P}W)(x,\omega')$ and $(\mathcal{P}^2W)(x,\omega')$, respectively.

The terms $\mathcal{P}^iW_e$ of the infinite sum have the following structure:

$$\begin{aligned} (\mathcal{P}^0W_e)(x,\omega') &= W_e(x,\omega') \\ (\mathcal{P}^1W_e)(x,\omega') &= \int_{\Omega_1} BSDF(x,\omega',\omega_1)\ W_e(RT(x,\omega_1),\omega_1)\ \cos\theta_1\ d\omega_1 \\ (\mathcal{P}^2W_e)(x,\omega') &= \int_{\Omega_2}\int_{\Omega_1} BSDF(RT(x,\omega_1),\omega_1,\omega_2)\ BSDF(x,\omega',\omega_1) \\ &\qquad W_e(RT(RT(x,\omega_1),\omega_2),\omega_2)\ \cos\theta_1\ \cos\theta_2\ d\omega_1\ d\omega_2 \\ &\dots \end{aligned} \quad (3.40)$$

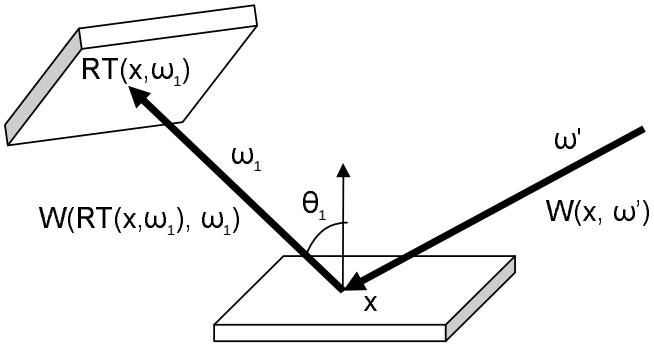

Figure 3.7: Shooting path integration: The geometry of the integrand of the term $(\mathcal{P}W)(x, \omega')$

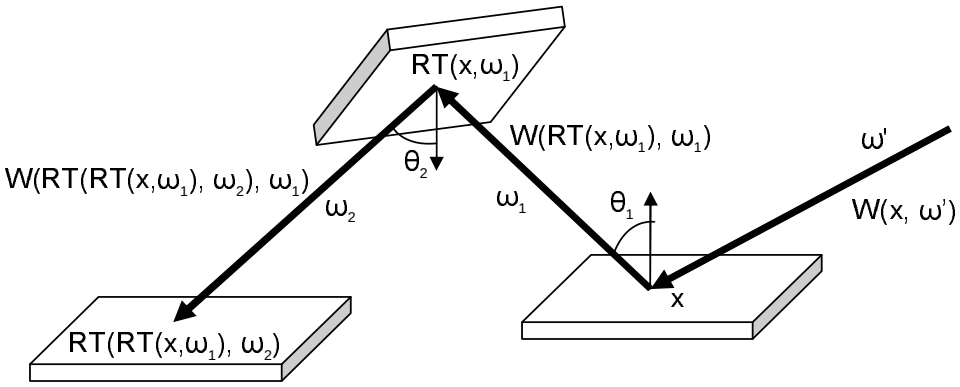

Figure 3.8: Shooting path integration: The geometry of the integrand of the term $(\mathcal{P}^2W)(x, \omega')$

The physical interpretation of these terms is: "$(\mathcal{P}^i W_e)(x, \omega')$ is the percentage of the incoming radiance impinging at the point x in the direction ω' which reaches the measuring device after exactly i scatterings (including the scattering at the point x)."

The most rigorous algorithm which uses Equation 3.39 in order to solve the global illumination problem is *light tracing*. [DLW93] Light tracing does not explicitly store the computed potential function—instead it directly computes the integral 3.32 using Monte Carlo integration. The algorithm generates paths which consist of line segments (rays). The first ray starts at a randomly chosen point of a randomly chosen light source in a randomly chosen direction. If a ray does not hit a surface, the path is terminated. Otherwise a decision is made as to whether the path will be prolonged by another ray (the direction of this scattered ray is randomly generated) or whether the path will be terminated.[8] The probability

[8]Instead of the termination of the path in this case, the algorithm may shorten the path and recursively trace another rays.

of the prolongation of a path is proportional to an estimated contribution of the new ray to the integral 3.32 (as the potential transport operator $\mathcal{P}$ is a contraction, this contribution decreases as the length of the path increases). When a path is terminated, its contribution to the integral 3.32 is added and the path is discarded. The direct potential W_e from the camera can be collected either at the last point of the path which is being terminated (Fig 3.9) or at the last point of every segment of the path (Fig 3.10).

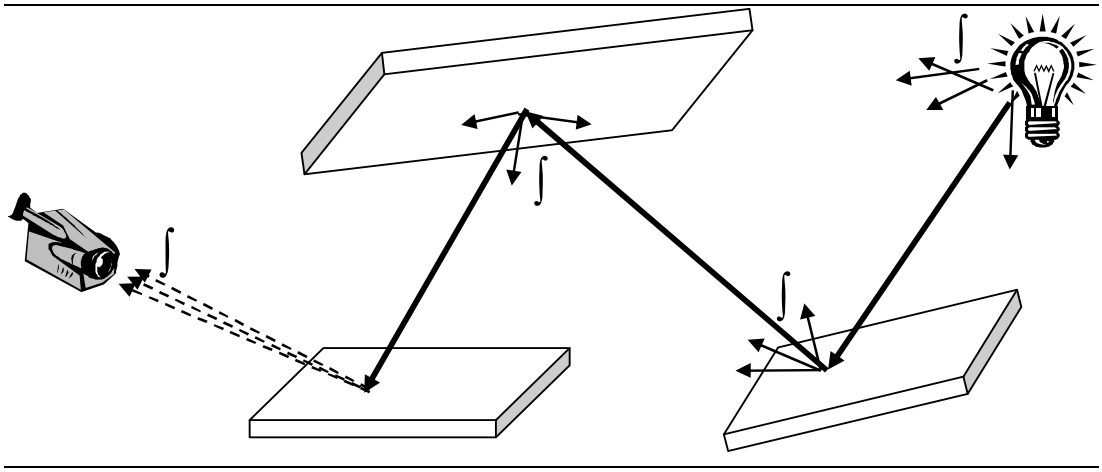

Figure 3.9: Light tracing with a single collection of the direct potential

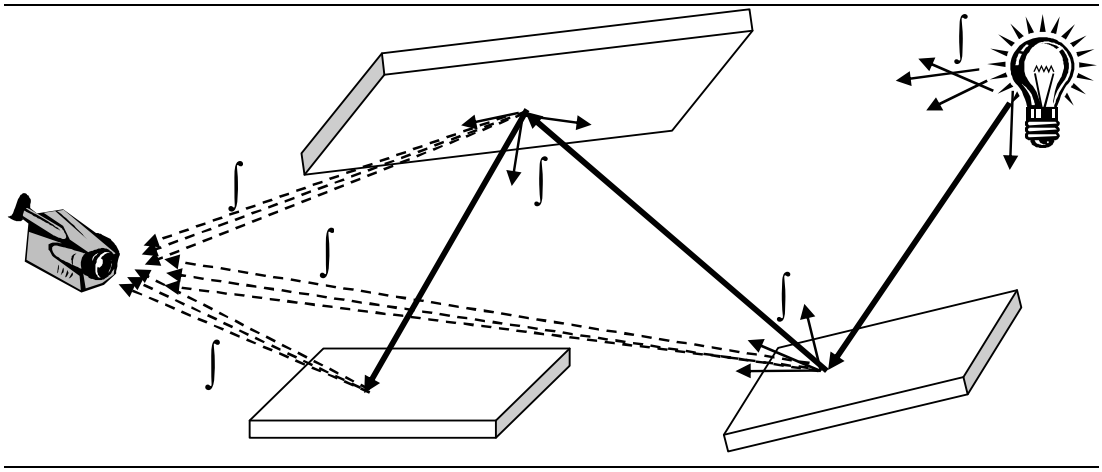

Figure 3.10: Light tracing with a multiple collection of the direct potential

Bidirectional path integration

A disadvantage of the gathering and shooting path integrations are their slow convergence. In practice, this slow convergence means that certain light phenomena, e.g. caustics, take a long time to compute—however, we must stress that if

the computational time is unlimited then any of the direct methods can correctly solve the global illumination problem (the expected solution is equal to the exact solution with probability 1). Bidirectional path integration combines the gathering and shooting paths into global paths which connect the measuring device with a light source. Note that the gathering paths generated by distributed ray tracing also connect the measuring device with a light source—however, the length of the "shooting path segment" (which collects the direct lighting) is limited to the length one. Similarly, the shooting paths generated by light tracing also connect the measuring device with a light source, but in this case the "gathering path segment" (which collects the direct potential) is limited to the length one. Bidirectional path integration connects shooting and gathering paths of arbitrary lengths, see Fig. 3.11 and Fig. 3.12 [LW93], [VG94], [Vea97]

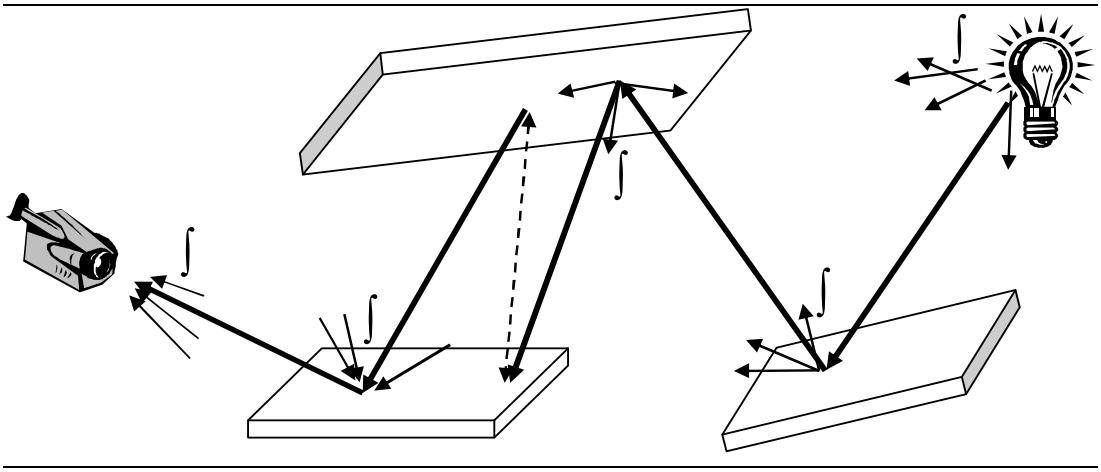

Figure 3.11: Bidirectional path tracing

The above path integration methods as well as other direct methods (such as stochastic iteration, [SK99b], [SK99a]) are stochastic methods. As such they suffer from stochastic errors which are perceived as noise in the computed images. However, if approximations are avoided in the algorithms which are used within the methods, then probabilistic guarantees can be given on the computed results. The most important guarantee is that the computed results are on average correct and that the stochastic errors can be eliminated by using more random samples (e.g. more paths or more iterations).

3.4.2 Approximation methods

Most approximation methods restrict the modeling of surface material properties to perfect diffuse or perfect specular reflectors. The idea is to simplify the structure of one of the rendering equations and to apply a deterministic method which solves the modified equation. The disadvantage of this is that practically

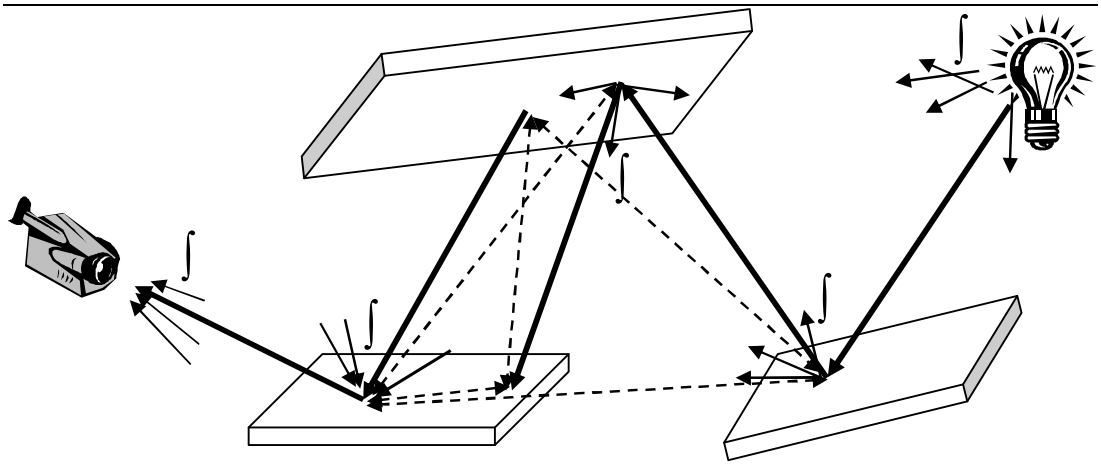

Figure 3.12: Bidirectional path tracing with multiple connections of the gathering and shooting paths

no surface of nature is perfectly diffuse or perfectly specular (or a linear combination of both). Therefore the solution to the modified problem usually differs from the solution to the original (physically more correct) problem and this difference cannot usually be bounded. Two well-known methods of this kind are (eye-) ray tracing and radiosity. The next two chapters are devoted to these two methods. We sketch the simplifying assumptions which they make below.

(Eye-) Ray tracing

(Eye-) ray tracing solves a simplified version of Equations 3.28 and 3.30 using the expansion 3.35. It is assumed that all object surfaces are perfect specular reflectors (or perfect specular transmitters, or both) for the purpose of the computation of the indirect illumination (the terms $\mathcal{R}^i L_e(x,\omega),\ i \geq 2$).

A *perfect specular reflector* is a surface which is characterised by the following $BSDF$:

$$BSDF_r(x,\omega',\omega) = k_s\ \Delta(\omega, \omega' - 2(N\cdot\omega')\cdot N) \tag{3.41}$$

where x is a surface point, ω' is the incoming direction (a normalised 3D vector), ω is the outgoing direction (a normalised 3D vector), $k_s \in \langle 0,1)$ is a specular coefficient, N is the surface normal at the point x and Δ is a slightly modified Dirac function (for directions in 3D):

$$\forall \omega_2 \neq \omega_1 :\ \Delta(\omega_1,\omega_2) = 0$$
$$\forall \omega_1 : \int_\Omega \Delta(\omega_1,\omega_2)\ d\omega_2 = 1$$

The direction $\omega' - 2(N \cdot \omega') \cdot N$ in Equation 3.41 is the mirror direction which lies in the same plane as ω' and N and the angle between this mirror direction and the surface normal is equal to the incoming angle, see Fig. 3.13.

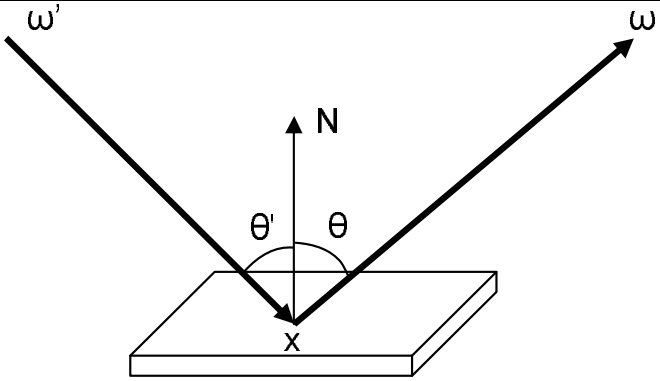

Figure 3.13: Perfect specular reflection. $\theta = \theta'$

A *perfect specular transmitter* is a surface which is characterised by the following $BSDF$:

$$BSDF_t(x, \omega', \omega) = \begin{cases} k_t \ \Delta\left(\omega, k_{ior}\ \omega' + \left(k_{ior}\cos\theta - \sqrt{1 + k_{ior}^2(\cos^2\theta - 1)}\right) \cdot N\right) \\ \qquad \text{if} \quad 1 + k_{ior}^2(\cos^2\theta - 1) \geq 0 \\ \\ k_s \ \Delta(\omega, \omega' - 2(N \cdot \omega') \cdot N) \\ \qquad \text{if} \quad 1 + k_{ior}^2(\cos^2\theta - 1) < 0 \end{cases} \tag{3.42}$$

where x is a surface point, ω' is the incoming direction (a normalised 3D vector), ω is the outgoing direction (a normalised 3D vector), $k_t \in \langle 0, 1)$ is a specular transmission coefficient, $k_{ior} \in (0, 1\rangle$ is the index of refraction[9] between the surrounding medium and the surface material at x, N is the surface normal at x, Δ is the modified Dirac function and θ is the angle between the incoming direction and the surface normal at x (hence, $\cos\theta = N \cdot (-\omega')$), see Fig. 3.14.

The direction $k_{ior}\ \omega' + \left(k_{ior}\cos\theta - \sqrt{1 + k_{ior}^2(\cos^2\theta - 1)}\right) \cdot N$ is the perfect direction of refraction (Snell' Law). The first split term of $BSDF_t$ represents the perfect specular refraction, the second split term represents the *total internal reflection* which occurs when the incoming angles is small.

The resulting $BSDF$ is the sum of the perfect specular reflection and the perfect specular transmission:

[9]The index of refraction may depend on the wavelength of the incoming light. This is why a glass prism divides the refracted white light into a rainbow. [New52] A varying index of refraction can be included in this model.

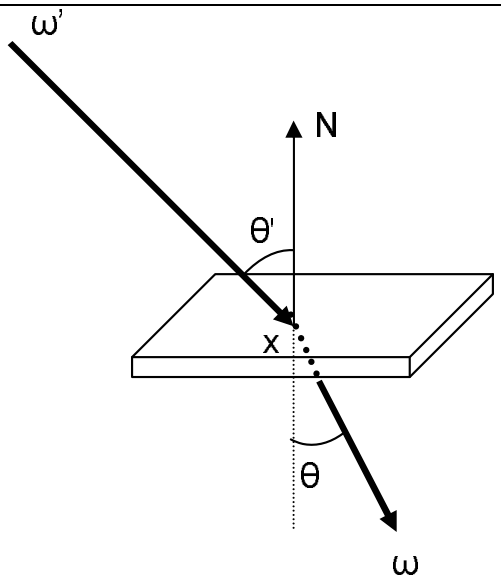

Figure 3.14: Perfect specular refraction. $\sin\theta = k_{ior}\ \sin\theta'$

$$BSDF(x,\omega',\omega) = BSDF_r(x,\omega',\omega) + BSDF_t(x,\omega',\omega) \tag{3.43}$$

The $BSDF$ above is used in the computation of the terms $\mathcal{R}^i L_e(x,\omega),\ i \geq 2$ in the potential equation 3.31. For the computation of the direct lighting (the terms $\mathcal{R}^0 L_e(x,\omega)$ and $\mathcal{R}^1 L_e(x,\omega)$), either the Phong model (Equation 3.25) or the modified Phong model (Equation 3.26) are used.[10] This is not quite correct but it saves computational time. Eye ray tracing is very similar to distributed ray tracing which is schematically depicted in Fig. 3.6. The difference between the two is that eye ray tracing only computes the integral which corresponds to the direct camera rays (the computation of this integral is called *anti-aliasing*) and the integral which corresponds to the dashed direct light rays (the computation of this integral is called *shading*). All the integrals in-between are only approximated by sampling the radiance function in two principal directions (the direction of the perfect reflection and the direction of the perfect transmission).

Radiosity

The radiosity method consists of two steps in order to compute the picture viewed by the camera. In the first step (the so-called view-independent step), the unknown radiance function is computed using a simplified version of Equation 3.30. Unlike (eye-) ray tracing, the radiosity method explicitly represents the radiance

[10]The term $\mathcal{R}^0 L_e(x,\omega)$ is often ignored. This term corresponds to the direct lighting which impinges at the camera and it is responsible for the effect known as "lens flare". This effect can be observed when a picture is taken against the sun. The sunlight which directly hits the camera lens creates colourful circles.

function. In the second step (the view-dependent step) the picture viewed by the camera is computed using Equation 3.28.

The first of the simplifying assumptions which are made by radiosity is that all object surfaces are perfect diffuse reflectors. A *perfect diffuse reflector* is a surface which is characterised by the following $BSDF$:

$$BSDF(x,\omega',\omega) = \begin{cases} \frac{k_d(x)}{\pi} & \text{if } \omega \text{ lies in the half-space of reflection } (\omega' \text{ and } N) \\ 0 & \text{otherwise} \end{cases} \tag{3.44}$$

where $k_d(x) \in \langle 0,1)$ is a diffuse reflection coefficient, see Fig. 3.14. Note that this $BSDF$ does not depend on ω' or ω (as the basic radiosity method ignores the transmission of light, we will in the following text restrict the set of incoming and outgoing directions Ω to the directions in the half-space of reflection). This means that the incoming radiance is equally reflected in all outgoing directions. This assumption allows us to write Equation 3.30 as

$$L(x,\omega) = L_e(x,\omega) + \frac{k_d(x)}{\pi} \int_\Omega L(RT(x,-\omega'),\omega') \cos\theta' \, d\omega' \tag{3.45}$$

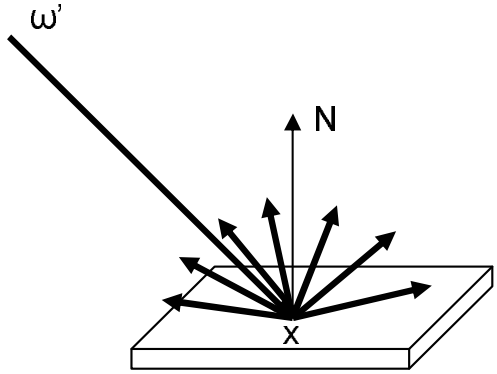

Figure 3.15: Perfect diffuse reflection. The incoming radiance is equally scattered in all outgoing directions in the half-space of reflection, independently of the incoming direction ω'

The second simplifying assumption is that all surfaces are modeled as planar areas (so-called *patches*) and that the 3D model only contains a finite number of these patches. Moreover, the radiance over all points of one patch is assumed to be constant. Let $P_1,\ldots,P_n$ denote the patches. These patches include light sources. The radiant emittance of all light sources is assumed to be perfectly diffuse ($L_e(x,\omega)$ is only a function of x) and constant at every point of one patch. In other words, all light sources are area light sources. We define *exitance* at a point of light source (which is characterised by its radiant emittance $l_e^i(x,\omega)$, see Section 3.2.4) as

$$E(x) = \int_{\Omega} l_e^i(x,\omega)\ \cos\theta\ d\omega \tag{3.46}$$

If all light sources are area light sources (perfect diffuse emitters) and if their areas do not overlap, then the above equation can be simplified by using the fact that $L_e(x,\omega)$ does not depend on ω (Equation 3.6 is used to express the direction ω as polar angles):

$$\begin{aligned} E(x) &= \int_{\Omega} L_e(x,\omega)\ \cos\theta\ d\omega = L_e(x,\omega) \int_{\Omega} \cos\theta\ d\omega \\ &= L_e(x,\omega) \int_0^{\pi} \int_0^{2\pi} \cos\theta\ \sin\theta\ d\theta\ d\phi = \pi\ L_e(x,\omega) \end{aligned} \tag{3.47}$$

We recall here the definition of radiosity (Equation 3.14). If a point x lies on a surface of a perfect diffuse reflector (or a perfect diffuse emitter), then $L(x,\omega)$ does not depend on ω. Hence, radiosity at the point x is equal to

$$B(x) = \int_{\Omega} L(x,\omega)\ \cos\theta\ d\omega = L(x,\omega) \int_{\Omega} \cos\theta\ d\omega = \pi\ L(x,\omega) \tag{3.48}$$

The multiplication of Equation 3.45 by π yields

$$B(x) = E(x) + k_d(x) \int_{\Omega} L(RT(x,-\omega'),\omega')\ \cos\theta'\ d\omega' \tag{3.49}$$

Note that not all incoming directions ω' contribute equally to the integral in the above equation. Those directions for which the function $RT(x,-\omega')$ does not find a surface point do not contribute at all. For all other directions the point $y = RT(x,-\omega')$ lies on a surface of a perfect diffuse reflector or emitter. For the point y it holds that

$$L(y,\omega') = L(RT(x,-\omega'),\omega') = \frac{B(y)}{\pi} \tag{3.50}$$

The above equation and the substitution of $d\omega'$ into Equation 3.49 using Equation 3.5 yield

$$B(x) = E(x) + \frac{k_d(x)}{\pi} \int_S B(y)\ \frac{\cos\theta\ \cos\theta'}{r(x,y)^2}\ V(x,y)\ dA \tag{3.51}$$

where θ is the angle between the surface normal at the point y and the direction from y to x, $r(x,y)$ is the distance between the points x and y and dA is a differential area around the point y. The integration domain S are all surface points. The function $V(x,y)$ is a *visibility function* which is needed in order to avoid the collection of multiple contributions to the integral after the change to the integration domain. Note that in the integral of Equation 3.49, the function $RT(x,-\omega)$ returns the nearest point to x in the direction of $-\omega$, whereas

the integration over all surface points also includes further points. The visibility function $V(x, y)$ returns 1 for the nearest point and 0 for all further points:

$$V(x,y) = \begin{cases} 1 & \text{if the points } x \text{ and } y \text{ are mutually visible} \\ 0 & \text{otherwise} \end{cases} \tag{3.52}$$

Radiosity $B(y)$ in Equation 3.51 depends on the patch on which the point y lies. However, we assume that the radiosity in each point of one patch is constant. The integration domain S is the union of all patches, $S = \bigcup_{i=1}^{n} P_i$ and so the integral of Equation 3.51 can be divided into a sum of integrals:

$$B(x) = E(x) + \frac{k_d(x)}{\pi} \sum_{i=1}^{n} \int_{y \in P_i} B(y) \frac{\cos\theta \ \cos\theta'}{r(x,y)^2} V(x,y) \ dA_i \tag{3.53}$$

Equation 3.53 is not quite correct because it violates the assumption of the constant radiosity over all points of one patch—the values of $B(x)$ which are computed for different points x of one patch using Equation 3.53 are not necessarily equal. To overcome this problem, we define *patch radiosity* $B_i, \ i = 1, \ldots, n$ as an area-weighted average of the point radiosities $B_i(x), \ x \in P_i$:

$$B_i = \frac{1}{A_i} \int_{x \in P_i} B(x) \ dx \tag{3.54}$$

where A_i is the area of the patch P_i. Similarly, we define *patch exitance* $E_i, \ i = 1, \ldots, n$ as an area-weighted average of the point exitances $E(x), \ x \in P_i$:

$$E_i = \frac{1}{A_i} \int_{x \in P_i} E(x) \ dx \tag{3.55}$$

A correct version of Equation 3.53 is therefore

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} \frac{\cos\theta \ \cos\theta'}{\pi \ r(x,y)^2} V(x,y) \ dA_j \ dA_i \tag{3.56}$$

where $\rho_i = k_d(x)$ at any point $x \in P_i$ (the diffuse reflection coefficient $k_d(x)$ is constant for all points of a patch).

If we denote

$$F_{ij} = \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} \frac{\cos\theta \ \cos\theta'}{\pi \ r(x,y)^2} V(x,y) \ dA_j \ dA_i \tag{3.57}$$

then Equation 3.56 can be written as

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j \tag{3.58}$$

Equation 3.58 is called the *radiosity equation*. Note that the terms F_{ij} only depend on the geometry of the patches in the 3D model and can therefore be computed independently of the illumination. The terms F_{ij} are called *form factors*.

The radiosity equation is a linear equation system with the unknowns B_i, $i = 1, \ldots, n$. The system can be written in an equivalent matrix form as

$$\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \ldots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \ldots & \vdots \\ \vdots & \ldots & \ddots & \vdots \\ -\rho_n F_{n1} & \ldots & \ldots & 1-\rho_n F_{nn} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} \tag{3.59}$$

Any known method for solving linear equation systems (e.g. Gauss method) can theoretically be used to solve this equation system. The main practical difficulties are the size of the system (very large models consist of millions of patches) and the computation of the matrix elements (the computation of the form factors F_{ij}).

When the radiosity equation is solved, Equation 3.28 is used to compute the picture. The patch radiosities are stored, therefore the model can quickly be rendered for many different cameras without the need to recompute the illumination.

Remark. A simplification of the potential equation leads to a similar linear equation system to that of Equation 3.58. The unknowns in this case are "diffuse patch potentials" ("diffuse patch potential" is a counterpart of radiosity). After the modified equation system has been solved, pictures of the model viewed by one camera can quickly be rendered for different lighting conditions (for different sets of light sources) using Equation 3.32.

The knowledge of both the patch radiosities and the "diffuse patch potentials" allows for a quick rendering of the model for different cameras under different lighting conditions. The preprocessing phase in this case consists of the solving of two (similar) linear equation systems.

Furthermore, the two equation systems (the radiosity equation and the "diffuse potential equation") can be combined in order to speed up the rendering for one camera and for one set of light sources. [SAS92], [SP94], [BW95] •

3.5 Conclusions

A realistic simulation of global illumination is a challenging problem. An accurate simulation of the underlying physics on an atomic level is computationally not feasible for the purpose of the illumination of large-scale scenes. The state of the

art mathematical formulation of the global illumination problem involves two adjoint Fredholm integral equations of the second kind: radiance and potential equations. These equations can be extended in order to include further light phenomena without destroying their structure.

The methods which solve the global illumination equations can be divided into two categories. The methods in the first category attempt to directly solve the equations, without further approximations. The essence of these methods is a Monte Carlo integration in high dimensions which at least provides probabilistic guarantees of the accuracy of the computed images. The methods in the second category make additional approximations. Two popular examples of such methods are ray tracing and radiosity. Ray tracing is more flexible than radiosity as it does not make approximations on the modeling level and can be extended in order to compute the full global illumination with no approximations. Also, some light phenomena which are not covered in the radiance and potential equations—for instance participating medium—can be incorporated into a ray tracing algorithm (much more easily than into a radiosity algorithm).

Some textbooks on computer graphics make a distinction between "view-dependent" and "view-independent" methods. (Note that the illumination never depends on the camera, therefore the quotation marks.) From a theoretical point of view, it is not very important as to whether a method explicitly stores the computed illumination ("view-independent") or only computes the information which is necessary in order to render a picture viewed by the camera ("view-dependent"). However, the explicit storage of illumination may consume a large amount of memory and may invalidate error bounds given by the numerical methods which are used to solve a rendering equation.

Contemporary 3D standards do not reflect the requirements of global illumination algorithms. Commercial modeling programs use proprietary 3D formats which are incompatible with other modeling programs. The 3D models can be exported into one of the existing open formats such as VRML [ISO97] but this leads to a loss of information in the 3D model. For instance, it is very paradoxal that although *practically no modeling system internally works with triangles* (human 3D artists do not model a surface using non-overlapping triangles, they use Constructive Solid Geometry instead), the only representation which can be exported is a triangle mesh. The lack of a *portable* 3D standard is in our opinion a very serious problem because imperfections in input data cause imperfections in rendered images. We sketch a solution to this problem in Section 6.1.

Chapter 4

Ray tracing

The ray tracing method (also referred to as eye-ray tracing) computes an image of a 3D scene by recursively tracing rays from the eye through the pixels of a virtual screen into the scene, summing the light path contributions to pixels' colors. The main idea behind this method is to only follow those photon paths which contribute to the image seen by the camera. The basic ray tracing algorithm was proposed by Whitted in 1980.[1] [Whi80] This algorithm is still very popular in real-world rendering systems. It can serve as a basis for all direct methods which are introduced in Section 3.4 and also, perhaps less apparently, for the radiosity method which we discuss in Chapter 5.

This chapter focuses on the basic ray tracing algorithm which only traces rays in the directions of perfect specular reflection and perfect specular refraction in order to evaluate the higher-order integrals of the radiance equation (see Section 3.4.2). We present the existing optimisation techniques which accelerate the computation of the function $RT(x, \omega)$ (which is defined in Section 3.1). [Gla89] In spite of these optimisation techniques, sequential computation times can range from minutes to hours. A parallelisation of the algorithm is therefore very desirable.

Many research papers on parallel ray tracing are only interested in the performance of the algorithms. Software engineering issues are often ignored. Among these belong questions such as:

- Can the parallel algorithm be easily integrated into an existing sequential code?
- Will it be possible to continue the development of the sequential code without the need of reimplementation of the parallel version?

[1] The idea of tracing rays from the camera to the 3D scene was first described in [App68] in the context of hidden-surface removal.

- Which existing sequential optimisation techniques can be reused in the parallel version (and which can not)?

We keep these questions in mind throughout this chapter. The parallelisation method which we propose is based on the method of Green and Paddon. [GP89], [Gre91] We propose a better screen subdivision algorithm which improves the existing screen subdivision algorithms. *We show that false conclusions may be drawn if the performance of the parallel algorithms is only compared empirically and if active polling is used in the underlying communication library.*

4.1 The basic ray tracing algorithm

The basic ray tracing algorithm (sometimes referred to as backwards ray tracing or eye ray tracing) solves the radiance equation 3.30. [Whi80], [Gla89] Instead of the computation of the radiance $L(x, \omega)$ at all surface points x and all directions ω, ray tracing only computes the radiance function at points and directions which contribute to the integral of Equation 3.28 (which corresponds to the image viewed by the camera). The computed radiance values are typically not permanently stored. Ray tracing traces rays through the camera pixels in order to compute the integral of Equation 3.28. These rays are called *primary rays*. The direct lighting contribution to the radiance function is computed by the *ray tracing shader* at the closest intersection points between these rays and the 3D surfaces in the direction to the camera (the computation of the closest intersection points equals to the computation of the function RT which is defined in Section 3.1). At these intersection points new rays are generated. The basic ray tracing algorithm only generates two secondary rays, in the directions of perfect specular reflection and perfect specular transmission. These secondary rays are recursively traced until the direct lighting in the new intersection points has no significant contribution to the integral of Equation 3.28 (or until a user-defined recursion limit is reached).

The computation of the direct illumination contributions in the intersection points involves the generation of so-called *shadow rays* (the dashed rays in Fig. 3.6). The shadow rays sample the directions from the points on the surfaces of light sources to the intersection point in order to compute the direct illumination terms of the integrals of Equation 3.25 (the Phong illumination model). The basic ray tracing algorithm usually assumes the use of point light sources only, which reduces the number of the shadow rays to the number of light sources for each intersection point.

4.2 Sequential optimisation techniques

Already Whitted identified the repeated evaluation of the function $RT(x, \omega)$ (which returns the closest intersection of the ray starting at point x in direction ω with the 3D scene) as the far most expensive activity (ca. 90% of the processing time) in the ray tracing algorithm. [Whi80] This evaluation involves the computation of a number of ray-object intersections. We will refer to these intersection computations as *ray tracing operations (RTOPs)*.

Although not all secondary rays must be generated (not all materials are specular reflectors or transmitters), the number of the RTOPs is still very high. The following estimation can be found in [DS84] Denote an average number of secondary rays spawned at an intersection point N and assume that the average depth of the ray tracing recursion tree over all primary rays is D. (One ray tracing recursion tree is assigned to one pixel. The root of the ray tracing recursion tree corresponds to the primary ray generated for that pixel. The remaining nodes of the tree correspond to the rays generated by the ray tracing recursion.) The average number of nodes in one tree is then equal to

$$\sum_{i=1}^{D} N^{i-1} = \frac{N^D - 1}{N - 1}$$

Denote L the average number of shadow rays over all intersection points (if we assume that the scene only contains point light sources, then L is equal to the number of point light sources). Denote the number of primary rays W (W is equal to the number of camera pixels). Denote the number of objects in the scene X (we assume here that objects are geometric primitives). Then the total number of RTOPs is equal to

$$\#RTOP = W \cdot X \left(\frac{N^D - 1}{N - 1} \right) (1 + L)$$

For a realistic setting $W = 720 \times 576$, $X = 1000$, $L = 2$, $N = 1.2$, $D = 5$ the total number of RTOPs is approximately equal to $9.26 \cdot 10^9$. This section describes some of the techniques which reduce this number.

4.2.1 Bounding volumes

A simple optimisation technique are bounding volumes. Bounding volumes do not actually reduce the number of ray tracing operations. The idea behind bounding volumes is to replace many expensive ray tracing operations with less expensive ones. Each finite object is enclosed into a volume whereby the computation of the intersections of a ray with the volume is much cheaper than the computation of the intersections of the ray with the object enclosed. The intersections with

the object must only be computed for those rays which intersect the object's bounding volume.

Good candidates for bounding volumes are boxes and spheres for which the intersection calculations are very fast. [RW80] Bounding volumes can be computed automatically in the preprocessing phase of the ray tracing algorithm. The more tightly the bounding objects enclose the original objects, the more computational effort is saved during the ray tracing algorithm.

4.2.2 Bounding slabs

A very important technique for the reduction of ray tracing operations are bounding slabs—a space subdivision hierarchy. The idea behind this technique is a construction of a tree (or, more generally, a DAG) which subdivides the 3D space containing the scene into a hierarchy of non-overlapping volumes. The leaves of this tree are either objects or their bounding volumes. [RW80]

When intersections of a ray with the scene are to be computed, the ray is first tested for an intersection with the root of the tree (the volume which contains all the objects). If the ray intersects this volume, the successors of the root node are recursively tested for intersections. The recursion is terminated when either the ray does not intersect any of the current node's successors or the current node is a leaf and the intersections for all objects comprised in the volume have been computed.

The theoretical upper bound on the cost of the tree traversal is $O(\sqrt[3]{N})$ for any balanced subdivision tree with the number of leaves equal (or proportional) to the number of objects. [RKJ98] Even though the worst case does not practically happen, the use of bounding slabs means a certain tradeoff. A bad situation occurs when a node of the tree corresponds to a large volume which contains several small objects (or one small object). In this case the costs of the tree traversal may be higher than the costs of the direct intersection of the objects' bounding boxes.

It is important that the bounding slabs can be constructed automatically in the preprocessing of the ray tracing algorithm. Non-uniform space subdivision based on BSP trees [Kap85] or octrees [Gla84] adapts better to general scenes than uniform space subdivision [FTI86]. A hybrid spatial subdivision is described in [CDP95].

Remark. It makes sense for some object types (e.g. triangle meshes) to build a local subdivision tree for the object's volume. This speeds up the local intersection calculations and does not influence the traversal of the main tree. •

4.2.3 Light buffers

Light buffers are a technique which is specific to the reduction of ray tracing operations for shadow rays. [HG86] In order to determine whether an intersection point is in a full shadow in a given direction with respect to a light source, it is not necessary to compute all the intersections along the direction. The light source is surrounded by a cube the surface of which is discretised into cells. (The choice of the number of the cells only influences the performance of this technique, not its correctness.) Bounding boxes of all objects are projected onto this cube in the preprocessing step and a list is created for each cell which contain the pointers to the bounding boxes which hit the cell. This list of objects' bounding boxes is sorted by their distances from the light source.

In order to determine whether a given point lies in a full shadow with respect to the light source, the intersections of the ray *from* the light source to the given point are only computed with the objects which are stored in the list of the cell through which the ray passes. This computation is terminated when either an intersection of the ray with an opaque object is found, or the distance of the next stored object is greater than the distance from the light source to the given point. (Note that if the given point is not in a full shadow with respect to the light source, the algorithm returns the list of intersections between the light source and the point. This information is needed by the ray tracing shader in order to approximate the attenuation of the emitted radiance with respect to the given point.)

4.3 Persistence of Vision Ray Tracer

It is not particularly difficult to implement a sequential ray tracer. However, it is not easy to implement a *good* ray tracer because of technical pitfalls which arise when several optimisation techniques are combined together and when further extensions have to be built in. The freeware (sequential) Persistence of Vision Ray Tracer [PT] is state of the art for several reasons:

- All important existing ray tracing optimisation techniques are comprised in POV-Ray. These include the use of bounding volumes, bounding slabs (a space subdivision hierarchy), light buffers and vista buffer.

- POV-Ray supports a variety of geometry primitives, light source types, cameras and materials. Constructive Solid Geometry is used in order to create more complex objects (see Section 3.2.2).

- The scene description language is a macro language which adds power to the CSG modeling.

- The implementation is portable. POV-Ray has been ported to practically all existing platforms. It does not rely on any graphical interface (although it is possible to use one) or external libraries which may cause a loss of portability.
- Although the program is relatively large (ca. 100000 lines of ANSI C code), the object-oriented way of coding make it robust and extensible. The source code is freely available.
- POV-Ray implements several extensions to the basic ray tracing algorithm. One of these extensions is the computation of the indirect diffuse illumination using distributed ray tracing (see Section 3.4.1). The implementation is based on the algorithm proposed by Ward, Rubinstein and Clear. [WRC88]
- POV-Ray is used and supported by many people. Most of the contributions of the Internet Ray Tracing Competition (IRTC) use POV-Ray as the final rendering system. [IRT]

POV-Ray has been developed by POV-Team, a group of volunteer programmers. The original implementation of POV-Ray (version 0.5, released in 1991) was based on DKBTrace by David Kirk Buck. The current official version is 3.5. However, the version 3.5 leaves the original principles of POV-Ray. In particular, the implementation of caustics in the version 3.5 is not only far from being perfect but also enormously increases the complexity of the implementation. [Lüc03] In our opinion, the last extensible official version of POV-Ray is 3.1g which we chose for our parallelisation and experiments.

4.4 Parallel ray tracing

Unless stated otherwise, we assume throughout this section that message passing is used for the communication between parallel processes.

4.4.1 Existing approaches

Parallel ray tracing algorithms can be roughly divided into two classes [Gre91]:

- *Image space subdivision (or screen space subdivision) algorithms* exploit the fact that the primary rays sent from the camera through the pixels of the virtual screen are independent of each other. Tracing of primary rays can run in parallel without a communication between processes. The problem of an unequal workload in processes must be considered. Another problem arises by rendering large scenes—a straightforward parallelization requires a copy of the whole 3D scene to be stored in the memory of each process. On the other hand, these algorithms are usually easy to implement.

- *Object space subdivision algorithms* geometrically divide the 3D scene into disjunct regions which are distributed in process' (processors') memories. The computation begins with passing of the primary rays to processes storing the regions through which the primary rays pass first. The rays are then recursively traced by the processes. If a ray leaves a process's region, it is passed to the process which stores the adjacent region (or discarded if there is no adjacent region in the ray's direction). An advantage of object space subdivision algorithms is that the maximum size of the rendered 3D scene is theoretically unlimited because it depends only of the total memory available in all processes. Potential problems are an unequal workload and a heavy communication between processes. Moreover, an implementation of these algorithms may be laborious.

- *Functional decomposition and hybrid algorithms*, extend the data-driven approach of object space subdivision algorithms with additional demand-driven tasks in order to achieve a better load balance.

We will briefly present the works which belong to the last two classes. (We recommend [RCJ98] and [CDR02] for further reading.) Then we will return to image space subdivision which is the base of our parallelisation.

Object space subdivision algorithms

One of the first parallel algorithms was proposed in [DS84] Their algorithm is based on a geometrical subdivision of 3D space into convex 3D regions. Each region is assigned to one process. Rays are traced by processes and when a ray leaves the region assigned to the process, it is passed to the neighbour which maintains the region in the direction of the ray. The authors give a theoretical estimation of the speedup, $O(\sqrt[3]{S^2})$ where S is the number of regions. (An analysis of the object space subdivision is given in [CWBV85] for an empty scene.) The authors also propose to adjust the boundaries of the regions in run-time in order to achieve a better balance. This algorithm has never been implemented.

The cost estimation prediction given in [RKC98] assumes an octree spatial subdivision (see Section 4.2.2) and predicts costs of parallel ray tracing for voxels of the octree. This prediction can be useful for a balanced static assignment of objects to processors e.g. in image space subdivision algorithms which work with a distributed object database (see Section 4.4.4).

The use of pyramidal-shaped regions is proposed in [BP88] and [PB89]. The pyramidal regions begin in the eye point and ensure that primary rays never leave their initial regions, which saves some communication between processes.

A mapping of 3D regions onto a hypercube is given in [KNS87]. Their idea is to achieve an efficient implementation of object space subdivision on a hypercube multiprocessor architecture. In [KNK+88], a load balancing strategy is described

in which each 3D region is maintained by a cluster of several processes. This allows for parallel intersection calculations inside one region.

The idea behind Jevan's work [Jev89] is to immediately send a prolonged ray to the neighbouring process which maintains the next region, before computing intersections with the ray with the current region. If the ray is intersected with the current region, the results of the neighbour's computations are canceled.

In [Pit93], an implementation of an object space algorithm is described. The experiments showed that the fine-grain strategy leads to a low (56%) efficiency and that the regular 3D space subdivision which was used in the implementation leads to a work imbalance. The regions which contain light sources are responsible for a high load in these regions.

Functional decomposition and hybrid algorithms

The algorithm described in [SC88] uses a functional decomposition in order to parallelise ray tracing. Each process stores a copy of the entire scene together with a bounding volume hierarchy (see Section 4.2.2). The tree of bounding volumes is divided into an upper and lower parts. All processes perform the intersection computations with the upper part of the tree for different rays in parallel. However, an overloaded process does not perform the intersection computations with the lower part of the tree—instead of that it sends this work to an underloaded process. Experimentally measured efficiencies range from 68% to 88%.

The algorithm presented in [RC97] is based on object space subdivision (the scene is distributed in process' memories). This algorithm distinguishes between several task types (such as shading tasks and intersection calculation tasks). Some of these tasks are generated on-the-fly and as not every task can be performed by any process, it is impossible to predict the loads in the processes. Other tasks are generated by a master process. Whenever a process is idle (that means, its queue of external requests is empty), it sends a work request to the master process. A similar approach is described in [NL96].

An algorithm which uses functional decomposition in order to make use of programmable hardware is proposed in [PBMH02]. (This work assumes a triangle representation of objects.)

4.4.2 Image space subdivision

The computations on the primary rays (pixels of the virtual screen) are independent of each other which suggest an assignment of screen areas to parallel processes. Therefore, the processes which perform the computations on non-overlapping screen areas do not need to communicate at all. However, there are several additional issues which must be considered in order to make this approach efficient and general:

1. The computational times for different primary rays are not equal and they cannot be reliably predicted before the computations have actually been performed.[2] It must also be said that even though the computations on primary rays are independent of each other, there is a coherence between primary rays which are close to each other in the image space.

2. The sum of computational times for all primary rays is much greater than the computational time for any single primary ray.

3. The communication between processes cannot be neglected even if the messages exchanged are very short.

4. Some sequential optimisation techniques such as saving of the primary rays in anti-aliasing schemes do lead to dependencies between pixels.

5. The parallelisation should assume that the complete scene description does not fit into memory of each process.

The first three issues are dealt with in this section. The problem of memory limitations is addressed separately, as it is suggested in [GP89], [Gre91]. Section 4.4.4 is devoted to the design of a distributed database.

A *static subdivision of the image space* is proposed in [Woo84]. This static subdivision scheme assigns the same amount of pixels to processes. The pixels assigned to one process are spaced at regular intervals across the screen. Woodwark's work assumes that the whole scene description fits into the memory of each process.

Theoretical as well as experimental arguments for why any static subdivision scheme is inadequate in order to achieve a good efficiency (over 90%) are given in [HA98]. The granularity of screen subdivision is not fine enough to yield the desired efficiency when the law of large numbers is applied.

Several works use *chunking* (which is sometimes referred to as *tiling*) in order to distribute the work among processes. Chunks are screen regions of a constant size which are assigned to idle processes on demand by a central master process. [CT96], [GP89], [Gre91], [BBP94], [FFB99], [FHK97], [KH95][3]

We found only one paper on screen space subdivision which reports experience with *work stealing*. [BBP94] The processes are connected to a logical ring. Initially, each process is assigned an equal amount of work. When a process finishes a job, it sends a job request to the ring. If the same job request returns from the

[2]Some results concerning *cost control* rather than cost estimation for primary rays are given in [CC02]. This cost control is based on the psychological observation that different pixels in the computed image have a different visual importance when the image is perceived by a human observer.

[3]The work by Keates makes use of hardware-supported shared memory in order to address the scene storage problems in the processes.

other side of the ring without having been satisfied, then the process knows that it can terminate. Otherwise the process gets a new screen part to compute.

The abstract model for scheduling parallel loops is very similar to screen space subdivision of parallel ray tracing (assuming enough memory to store the whole scene in memories of all processes) in the sense that there is a static pool of tasks which can be computed in parallel but for which the computational times cannot be predicted. [KW85], [FHSF91], [FHSF92], [FHBWW95], [FHSUW96] The cited papers give an analysis of *factoring* (also referred to as *fractiling*) which is similar to chunking but the chunk size is gradually decreased. The latest four papers propose a halving of the chunk size and compare such a factoring with a uniform size chunking. Assuming a normal probability distribution of the tasks' computational times, an analysis on the expected imbalance is given. The problem with these results is that the halving of the chunk size does not lead to a perfect balance of load. The knowledge of the expected imbalance does not improve the actual parallel time.

The algorithm which we propose below is a generalisation of the previous approaches. The uniform size chunking and the factoring into halves are special cases of our algorithm. The parameters of our algorithm are intuitive. We can characterise the setting of the parameters which yields a perfect load balance (while minimising the communication costs). The only remaining question is how to find this setting.

Perfect load balancing algorithm

We will assume a farming model which consists of one master process and N worker processes (see Fig. 4.1). The master process assigns non-overlapping screen areas to workers, collect the results from the workers and updates the frame buffer (the image which is being computed). A worker is initially waiting until it receives a job from the master. Then the worker traces the primary rays for the given area, returns the computed subimage to the master and waits for another job.

The farming scheme can be extended with a load balancing process (loadbalancer), which takes over one of the master's responsibilities—the distribution of jobs to workers (see Fig. 4.1). The only information which is needed by the loadbalancer process is the size of the image (this information is passed from the master to the loadbalancer at the very beginning of the computation). The introduction of the loadbalancer process effectively reduces the idle times in the worker processes. When a worker becomes idle (or shortly before it becomes idle), it sends a job request to the loadbalancer and the computed subimage to the master. [Pla98]

The problem is to determine an appropriate granularity of the assigned parts. The choice of granularity influences the load balance and the number of exchanged messages. There are two extreme cases: 1. The assigned parts are minimal (pix-

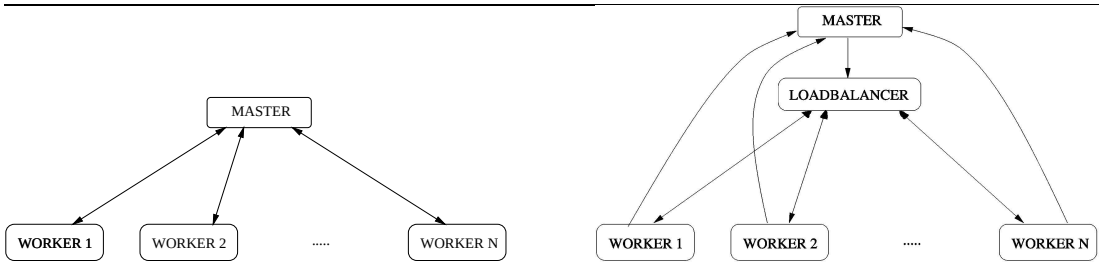

Figure 4.1: Left: A process farm. Right: A process farm extended with a load balancing process

els). In this case the load is balanced perfectly but the number of work requests is large (equal to the number of pixels which is usually much greater than the number of workers). 2. The assigned parts are maximal (the whole image is partitioned into as many parts as the number of workers). In this case the number of work requests is low but the load imbalance may be great. Fig. 4.2 depicts these two extreme cases.

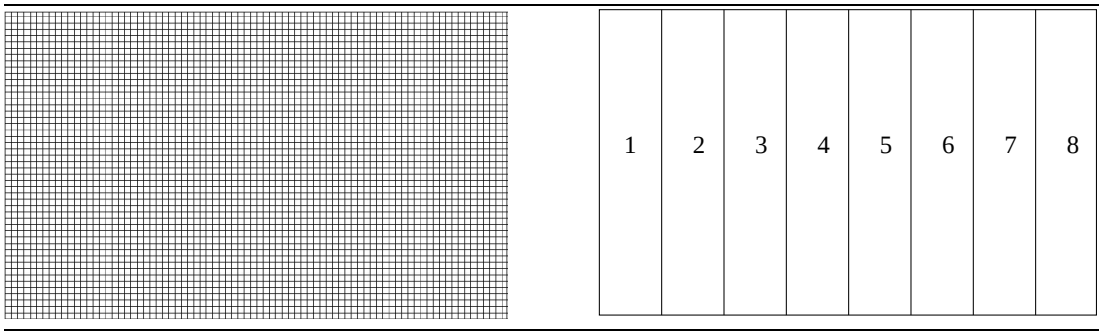

Figure 4.2: The extreme cases of chunking. Left: Minimal chunks. Right: Maximal chunks

Let W denote the total number of atomic parts (e.g. image pixels or image columns), let N denote the number of workers (a homogeneous parallel machine is assumed). The task of a load balancing algorithm is to compute the W atomic parts on N workers in the shortest possible parallel time. Let us assume that a minimal constant T is known which bounds the maximal ratio of the computational times on any two atomic parts ($T \geq 1.0$):

$$\frac{\text{processing time on part 1}}{\text{processing time on part 2}} \leq T \tag{4.1}$$

If this assumption holds, then the algorithm in Fig. 4.3 is perfect in the sense that it guarantees a perfect load balance (the maximal imbalance is not larger than the processing time of the atomic job with the largest processing time). At

the same time, the number of work requests is minimal. [Pla02a][4]

```
loadbalancer(float T, int W, int N)
  int part_size;
  int work = W;
  while (work > 0)
    part_size = max (1, ⌊work/(1 + T · (N − 1))⌋);
    for (counter = 0; counter < N; counter++)
      wait for a work request from an idle worker;
      if (work > 0)
        send job of size part_size to the worker;
        work = work − part_size;
  collect work requests from all workers;
  send termination messages to all workers;
```

Figure 4.3: The perfect load balancing algorithm (used in the loadbalancer process)

Claim. The algorithm in Fig. 4.3 always assigns as much work as possible to idle workers, while still ensuring the best possible load balance.

Proof. The algorithm works in rounds, one round being one execution of the while-loop. In the first round the algorithm assigns image parts of size

$$s_{max} = \max\left(1, \lfloor \mathrm{W}/(1 + T \cdot (\mathrm{N} - 1)) \rfloor\right)$$

(measured in the number of atomic parts). In each of the following rounds the parts are smaller than in the previous round. Obviously, the greatest imbalance is obtained when a processor p_{max} computes a part of the size s_{max} from the first round as long as possible (whereby the case of $s_{max} = 1$ is trivial and will not be considered here) and all the remaining $N - 1$ processors compute the rest of the image as quickly as possible (in other words, the load of the remaining $N - 1$ processors is perfectly balanced). The number of parts computed in parallel by all processors except of p_{max} is $W - s_{max}$. The ratio of the total workload (in terms of the number of processed atomic parts) of one of the $N - 1$ processors (let p_{other} denote the processor and let s_{other} denote its total workload) and s_{max} is then

$$\frac{s_{other}}{s_{max}} = \frac{\frac{W - s_{max}}{N-1}}{s_{max}} = \frac{\frac{W - \lfloor \mathrm{W}/(1+T\cdot(\mathrm{N}-1)) \rfloor}{N-1}}{\lfloor \mathrm{W}/(1 + T \cdot (\mathrm{N} - 1)) \rfloor}$$

[4] A similar algorithm was independently published in [PMTR95].

This ratio is greater or equal to T. This means that the processor p_{other} does at least T times more work than the processor p_{max} in this scenario. From this and from our assumptions about T and about the homogeneity of processors follows that the processor p_{max} must finish computing its part from the first round at the latest when p_{other} finishes its part from the last round. Thence, a perfect load balance is achieved even in the worst case scenario.

It follows directly from the previous reasoning that the part sizes s_{max} assigned in the first round cannot be increased without affecting the perfect load balance. (For part sizes assigned in the following rounds a similar reasoning can be used, with a reduced image size.) This proves the optimality of the above algorithm. •

Claim. The number of work requests (including final work requests that are not going to be fulfilled) in the algorithm in Fig. 4.3 is equal to

$$N \cdot (r+1) + \left\lceil W \cdot \left(1 - \frac{N}{1+T\cdot(N-1)}\right)^r \right\rceil$$

where

$$r = \max\left(0, \left\lfloor \log_{1-\frac{N}{1+T\cdot(N-1)}} (N/W) \right\rfloor\right)$$

Proof. It is easy to observe that

$$\frac{N\cdot W}{1+T\cdot(N-1)} \cdot \left(1 - \frac{N}{1+T\cdot(N-1)}\right)^{i-1}$$

atomic parts get assigned to workers during the i^{th} execution of the while-loop and that

$$W \cdot \left(1 - \frac{N}{1+T\cdot(N-1)}\right)^{i}$$

atomic parts remain unassigned after the i^{th} execution of the while-loop.

r is the total number of executions of the while-loop minus 1. The round r is the last round on the beginning of which the number of yet unassigned atomic parts is greater than the number of workers N. r can be determined from the fact that the number of yet unassigned atomic parts after r executions of the while-loop is at most N:

$$W \cdot \left(1 - \frac{N}{1+T\cdot(N-1)}\right)^{r} \leq N$$

which yields (r is an integer greater than or equal to 0)

$$r = \max\left(0, \left\lfloor \log_{1-\frac{N}{1+T\cdot(N-1)}} (N/W) \right\rfloor\right)$$

There are N work requests received during each of the r executions of the while-loop, yielding a total of $N \cdot r$ work requests. These do not include the work requests received during the last execution of the while-loop. The number of work requests received during the last execution of the while-loop is equal to

$$\left\lceil W \cdot \left(1 - \frac{N}{1 + T \cdot (N-1)}\right)^r \right\rceil$$

Finally, each of the workers sends one work request which cannot be satisfied. Summed up,

$$N \cdot (r+1) + \left\lceil W \cdot \left(1 - \frac{N}{1 + T \cdot (N-1)}\right)^r \right\rceil$$

is the total number of work requests. •

The perfect load balancing algorithm in Fig. 4.3 is a compromise between the two extreme chunking cases in Fig. 4.2. The two extremes are obtained when $T \to \infty$ (minimal chunks), or when $T = 1$ (maximal chunks), respectively. Fig. 4.4 illustrates the work assignment for $N = 2$ and $T = 3$.

Figure 4.4: Left: Illustration of the work assignment in the perfect load balancing algorithm for $N = 2$ and $T = 3$. Right: The exact number of work requests in the perfect load balancing algorithm as a function of the number of workers and the number of atomic parts

4.4.3 Setting of parameters in the perfect load balancing algorithm

Two parameters must be tuned in the perfect load balancing algorithm. The first parameter is the job time ratio parameter T. The second parameter has not been introduced yet. It may be useful to pack more pixels into a single job

towards the end of the algorithm because a computation on several pixels may cost much less than sending several messages instead of one. We will define M as the size of the minimal job which should be assigned to a worker. M is measured as the number of the original atomic work parts ($M \geq 1$). One line in the load balancing algorithm in Fig. 4.3 will be modified:

$$part\_size = \max\left(1, \lfloor work/(1 + T \cdot (N - 1))\rfloor\right)$$

will be replaced with

$$part\_size = \max\left(M, \lfloor work/(1 + T \cdot (N - 1))\rfloor\right)$$

It is obvious that the chunking approaches are special cases of our algorithm. Indeed, if M is equal to the chunk size and $T \to \infty$, then chunks of the size M will be distributed among workers on demand. The algorithm is also a generalisation of factoring which assigns a half of the still unassigned work equally to all workers—factoring in halves is obtained by setting $T = (2N - 1)/(N - 1)$ and $M = 1$.

The tuning of the parameters T and M should be fully automatical. The parameters can be tuned independently of each other. Unfortunately, both must be set before the computation begins and they both depend on the amount of the computation which is unknown before the computation finishes.

Setting of the atomic job size M

The parameter M controls the sizes of the smallest jobs which will be distributed in the last round of the algorithm. An overestimation of M results in a potential imbalance. The extreme setting of $M = W/N$ yields (independently of T) a static distribution of load which is obviously the worst case in terms of load balance.

The optimal setting of M depends on the communication costs and computational times for the original jobs. If M is too small, then communication costs can dominate the computation of jobs of the size M. Moreover, the loadbalancer process can become a bottleneck when many workers send their job requests frequently. This also influences the optimal setting of M.

Our suggestion is to run the load balancing algorithm with $M = 1$ and adapt M according to the measurements performed in the run-time. This involves an extension of the protocol between the loadbalancer and worker processes. The worker process can measure the computational time T_{job} spent on the last job and report this time to the loadbalancer together with a job request. The loadbalancer process measures the time from the moment t_{start} when a job was assigned to the worker to the moment t_{finish} when it becomes another job request from that worker. The communication time T_{comm} (for that particular job) is then equal to $T_{comm} = t_{finish} - t_{start} - T_{job}$. $T_{comm} \geq T_{job}$ is an indication of that the job size in the load balancing algorithm should not be further decreased in the next rounds.

(The fixation of M can be postponed e.g. until $T_{comm} \geq T_{job}$ is measured for *all* jobs which were assigned in the same round.)

Remark. The constant M can also be used in the worker process in order to prefetch another job. If the worker is computing a job and detects that the number of parts remaining does not exceed M at some moment, it can send a work request to the loadbalancer before its current job is finished. This prefetching can hide a part of the communication overhead. •

Setting of the job time ratio T

The parameter T controls the sizes of the largest jobs which will be distributed already in the first round of the algorithm. An underestimation of T results in a potential imbalance. The extreme setting of $T = 1$ yields (independently of M) a static distribution of load. The problem is that if T was underestimated at the beginning of the algorithm, then an adjustment of T in run-time does not help (unlike a run-time adjustment of M) unless jobs which have already been assigned can be taken away from workers.

The optimal setting of T depends on the ratio between the computational times on the longest and shortest job assigned. If T is too great, then an unnecessary communication overhead will add to the parallel time.

A conservative approach to the tuning of T is to assign jobs of the size M among the N workers in the first round and estimate T from the statistics which are collected after a worker finishes. T is then set to the maximum job time ratio measured on previous jobs. This alone does not prevent an underestimation of T—the small jobs from the first round only cover a small part of the image and therefore they are not a representative sample of the computational times across the whole image. However, the statistics collected during the first round allow for setting a limit on the computational times of jobs which are assigned in the next round. If a worker which computes a job detects during the computation that it has spent more time on the job than the limit allows (this detects an underestimation of T), then it stops the computation, sends a partially computed part of the job to the master and returns the part which has not been computed back to the loadbalancer. The loadbalancer updates its estimation of T and at an appropriate time it notifies the workers for which this update is relevant.

We suggest an optimistic approach which allows an underestimation of T by using an empirical constant for T which remains constant during the algorithm. The potential imbalance can be eliminated by the use of work stealing as an additional phase which begins immediately after the loadbalancer has no more parts to distribute. The work stealing phase involves a higher overhead than the farming but it overlaps with the farming phase and it is only initiated when an imbalance is detected—this means, when a worker sends a work request to the

loadbalancer and receives a "no more work" reply. The work stealing phase can overlap with the farming phase. The constant T controls the amount of work stealing which is needed to balance the load at the end. $T \to \infty$ yields a pure work stealing. The closer the estimation of T is to the optimal T, the shorter will be the work stealing phase.

4.4.4 Distributed object database

It is desirable that the screen subdivision algorithm proposed in the previous section also works if the entire scene description does not fit into the memory of each processor. This problem can be overcome by the use of a database which is capable of storing all the scene data. The access to an external database (such as a standard client-server database system or a disk storage in general) may be prohibitively expensive because the frequency of queries is very high. A careful reordering of operations in the ray tracing algorithm may overcome the problem of the slow communication with an external database [PKGH02] but this may result in a special implementation of an one-purpose ray tracer.

Green and Paddon suggest a different approach [GP89], [Gre91] which we decided to follow. It is assumed that the sum of the memories of all the processes involved in the parallel computation (the worker processes) is sufficient to hold the entire scene description. The function of each worker process is twofold: 1.it performs the recursive computations on the primary rays; 2.it serves as a database server for all the remaining workers, which means that it accepts data requests from the other workers and provides them with the requested data. **This makes parallel ray tracing a non-trivial application, see Section 2.1.**

Before coming to a design of the distributed database, we will make some observations. Ray tracers can support a variety of object types most of which are very small in memory. Polygon meshes are one of a few exceptions. If a scene does not fit into memory of one process, it is usually because it consists of several large polygon meshes. We therefore distribute mesh objects in our implementation but the implementation does not exclude a distribution of other object types.

We assume that any single object does fit into memory of each process. (It should also be said that a majority of scenes do fit into memory of each worker.) Some amount of memory is required by the program code, stack, image buffer, acceleration ray tracing structures (such as bounding boxes, bounding slabs, light buffers) etc. Most of these acceleration structures can be switched off so that they do not consume any memory.

The scene description stored in the database does not usually change during the computations of one image. This allows to decide during the preprocessing whether the amount of memory is sufficient to store the scene or not. Initially, the scene is stored in a file. The scene description in the file does not necessarily correspond to the storage of the scene in the (main) memory—we recall that

e.g. POV-Ray uses a macro language which allows a procedural creation of the objects. During the parsing of the scene file by all worker processes, objects are created in the memories of the worker processes. After an object has been created in memory, the workers synchronise and decide whether the object will be replicated in memory of all workers or whether the object will only be stored in the memory of one worker. In the latter case the worker with minimal current memory load is selected to become the *owner* of that object. All the remaining workers delete the *data* which belong to that object (e.g. vertex coordinates, vertex normals, triangle indices etc. belong to the data of a mesh object). However, they do not delete the *object's envelope*. This envelope contains a global object identifier, the identifier of the object's owner (e.g. the rank of the process which stores the object's data), the object's bounding box, a flag whether the object is currently present in the memory etc. After this, workers continue in parsing the scene file. Note that none of the workers consumes more memory than it is necessary at any one time.

If the parsing phase was successful, then each worker is able to store the objects which it owns and it has at least as much free memory as it is needed to store the data of the largest object in the distributed database. The worker's memory which is unused after the parsing stage will be used as a *cache* for the objects which are not owned by the worker. A worker never deletes the data of the objects which it owns.

Object's data are needed at two places in the ray tracing algorithm: in the intersection computations and in the shading. It must be ensured that the data of the object are in the memory before they are referenced—if the data are not in the memory, a data request is sent to the owner. It is likely that the object which is being referenced at the moment will also be referenced in a near future because of the coherence of the primary rays.[5] The sole fact that a cached object is being referenced is useful for the bookkeeping of the *cache policy* which uses this information in order to decide which objects will be released from the cache when new object's data are to be inserted into the cache. Fig. 4.5 depicts the pseudo-code of the function `Fetch_Object_Data` which is inserted in the ray tracing code immediately before the object's data will be referenced.

An object's owner acts as a server for all other workers which eventually need the object's data. A worker *must* run a separate thread which reacts to mesh requests by sending the data independently of the worker's computations.

Remark. The shading computations always follow the intersection computations for the same object—however, not immediately. A large number of other objects may be referenced meanwhile. The object which was referenced in the intersection computations may be released from the cache before it is referenced again in the shading. In order to save the latter data request, all the information

[5] A more sophisticated method can be used to predict the future object references. [RKC98].

```
Fetch_Object_Data(object)
{
 if (! is_in_memory(object))
 {
  send_data_request(object->owner, object->id);
  insert_into_cache(object);
  wait_for_data(object->owner, object);
 }
 else
 {
  if (object->owner != my_rank)
   cache_hit(object);
 }
}
```

Figure 4.5: The pseudo-code of the function `Fetch_Object_Data`. The function `insert_into_cache` makes space for the requested object data by removing other object's data according to the cache policy, and then increases the requested object's importance. The function `cache_hit` increases the object's importance

which is needed for the shading can be precomputed at the moment when the object is referenced for the first time. This information is stored in the object's envelope which always remains in the memory. By doing so an eventual expensive communication can be avoided for the price of a much less expensive unnecessary computation (not all intersected objects are going to be shaded).

Another efficiency improvement involves a prepacking of the object's data to ready-to-send buffers. However, there are several reasons for why this optimisation should not be used. One reason is that the prepacked buffers consume memory which can otherwise be used for the cache. Another reason is that this technique may limit the implementation to homogeneous parallel machines, unless the data encoding used for the prepacking is platform-independent. Furthermore, the time which is needed for the packing of the data is usually much shorter than the communication overhead. •

4.4.5 Experiments

This section first illustrates the tuning of the constants M and T of the load balancing algorithm in Fig. 4.3. Our experiments simulate the automatical tuning which is described in Section 4.4.3. Then we present results of experiments with a distributed object database. We compare three caching policies which attempt

to exploit coherence in object references in the ray tracing algorithm in order to save the number of expensive data requests. We also present efficiency measurements of the parallel ray tracing implementation which uses a distributed object database.

Throughout this section, the efficiency of a run of a parallel program will be measured as

$$\frac{\text{sequential time}}{\text{parallel time with } N \text{ workers} \cdot N}$$

Unless stated otherwise, the resolution of all the images computed during the experiments of this section was 720x576 (PAL). All images were computed with default POV-Ray 3.1g settings (with no anti-aliasing).

Setting of the atomic job size M

In order to show how the efficiency of a parallel implementation depends on the atomic job size, we used a chunking job assignment in the loadbalancer process which always assigns a constant chunk of the screen to an idle worker on demand. Our goal was to (manually) find the optimal chunk size for two given scenes, `BLOB` and `HAUS6`. The `BLOB` scene is extremely simple—it only consists of one object and one point light source. The `HAUS6` is fairly complex—it consist of ca. 600 objects and 8 point light sources.

For these experiments, we used a configuration with 90 workers which were running on a partition of 92 processors of the hpcLine (see Section 2.8 for the machine description). This is the maximal number of workers which can be mapped onto the allocated partition so that each process is mapped onto a single node of the machine (the remaining two nodes are used by the loadbalancer and the master processes). The optimal chunk size for this configuration determines the upper limit on the efficiency of any parallel computation which uses 90 worker processes. We recall that the chunk size which is smaller than the optimal chunk size will result in the domination of the communication overhead over the computation times of the smallest jobs which are assigned in the load balancing algorithm in Fig. 4.3 extended with the constant M, see Section 4.4.3. However, the smaller the *optimal* chunk size is, the better balance of load can be expected.

We compared two programs in these experiments, an event-driven one and a polling one. These programs are *identical on the binary level*. They both use the very same implementation of POV||Ray. The only difference is in the implementation of the TPL library. The event-driven program is POV||Ray linked with the TPL implementation which uses the interrupt mechanism described in Section 2.6.6. The polling program is (the same) POV||Ray linked with the TPL implementation which uses the polling mechanism described in Section 2.6.2. The two TPL implementations are based on the same source code and the difference

between the two are only a few lines of code which are conditionally selected using `#ifdef` directives. In order to make the comparison fair, neither the polling nor the event-driven version uses special optimisations—they both are generic implementations of the polling and event-driven mechanisms from Chapter 2.

The results of the experiments are shown in Fig. 4.6. The optimal chunk size for the `BLOB` scene is 720 pixels for both the polling and event-driven versions. The optimal chunk size for the `HAUS6` scene is 720 pixels for the polling program and 72–720 pixels for the event-driven program.

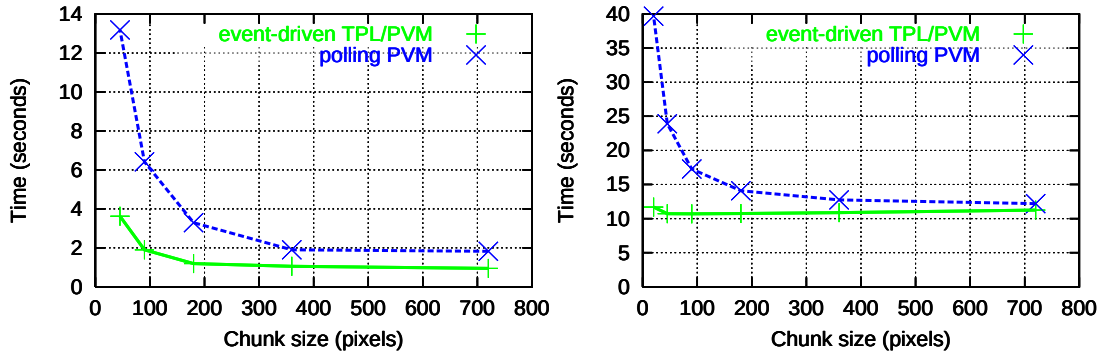

Figure 4.6: Absolute parallel times for 90 workers for a varying chunk size. Left: `BLOB` scene. Right: `HAUS6` scene

Fig. 4.7 shows the efficiencies of event-driven and polling programs for a pure chunking job assignment. The optimal constant chunk sizes (720 pixels for the `BLOB` scene and 360 pixels for the `HAUS6` scene) were used for these measurements.

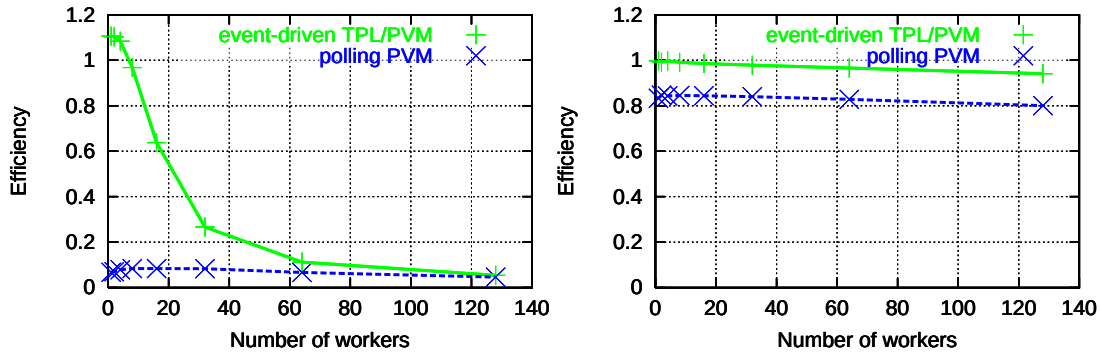

Figure 4.7: Efficiency of the chunking algorithm for a constant chunk size and varying number of worker processes. Left: `BLOB` scene (chunk size 720 pixels). Right: `HAUS6` scene (chunk size 360 pixels)

Setting of the job time ratio T

The optimal settings of M from Fig. 4.8 were used in order to determine the optimal settings of T for the same two scenes. We only performed these measurements for the event-driven program. The loadbalancer process used the algorithm of Fig. 4.3 extended with the constant M defined in Section 4.4.3. (No work stealing was used.)

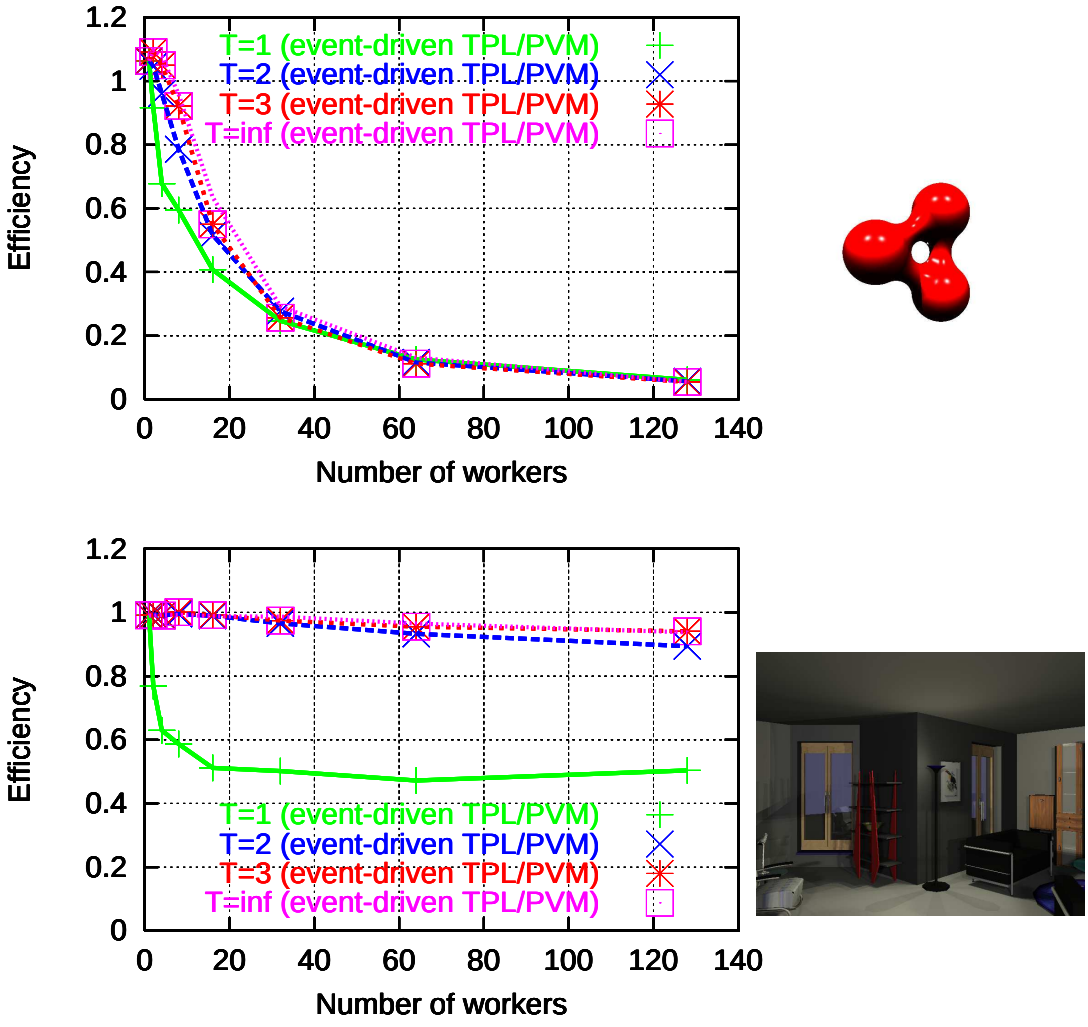

Figure 4.8: Efficiency of the perfect load balancing algorithm with the optimal chunk size and varying number of worker processes. Top: `BLOB` scene ($M = 720$ pixels). Bottom: `HAUS6` scene ($M = 360$ pixels)

These measurements show that the maximal efficiency is obtained for these two scenes for $T \to \infty$. However, the setting of $T = 3$ already yields the maximal

efficiency.[6] Note the poor efficiency for the setting of $T = 1$ (a static load assignment).

Choice of the cache policy

Together with P. Gramblička, our student, we compared the efficiencies of several cache policies on several scenes. [Gra98] These policies differ in how they react when an object is being referenced (`cache_hit` in Fig. 4.5) and in how they select the objects for the removal from the cache (so-called *victims*) in order to make space for a requested object (`insert_into_cache` in Fig. 4.5). An object request is also called a *cache miss*. The efficiency of a caching policy is measured using the *miss ratio*:

$$\frac{\#\text{object requests}}{\#\text{object references}}$$

The lower the miss ratio is, the more efficient is the policy. The object references in the definition of the miss ratio include references to objects which are owned by the process. We will only present the policies which are fully automatical and do not require any tuning:

- `RANDOM` does nothing when an object is being referenced. On an object request, victims are selected randomly until there is enough space in cache to store the requested object.
- `LRU` (Last Recently Used) maintains a linked list of the cached objects. When an object is being referenced, it is moved to the beginning of the list. On an object request, always the last object in the list is removed from the cache until there is enough space in cache to store the requested object.
- `LRU-COUNTER` assigns a counter to each object. This counter is initially zero and it is increased when the object is being referenced. On an object request, always the object with the lowest counter is removed from the cache.

Fig. 4.9 shows the results of the measurements on three scenes which were rendered sequentially in the resolution 640x480, using a simulated memory limit. The graphs may suggest that the `RANDOM` policy as approximately as good as `LRU`. This is only because the total number of references is relatively high. In fact, the absolute number of cache misses of `LRU` is ca. 25% lower than the absolute number of cache misses of `RANDOM` in all the graphs. `LRU` *performs very well* although its obvious disadvantage is that objects which are only referenced a few times remain a long in the cache. `LRU-COUNTER` uses counters in order to prevent

[6] The efficiency greater 1 for the `BLOB` scene is caused by the short sequential time which does not allow an amortisation of I/O calls in the sequential program.

this situation. The weakness of `LRU-COUNTER` is that objects which are referenced often during a short time interval remain very long in the cache.

Distributed object database

Fig. 4.10 shows efficiencies of parallel ray tracing with distributed object database which were measured with our old implementation of POV||Ray. The old implementation of POV||Ray was based on the GOLEM communication library. [Ree97] The GOLEM library uses PVM for inter-process communication and it uses polling in order to allow for an implementation of non-trivial applications.

The LRU cache policy was used in these measurements. The cache miss ratio was under 1% in the 20% case (only 20% of all objects are relevant for the rendering of this image) and the total number of data request was ca. 3000. The cache miss ratio increased only by 0.1% in the 10% case but the total number of data requests increased to ca. 500000. In the 5% case, the cache miss ratio was ca. 15% and the total number of data requests was ca. 7000000.

Our recent POV||Ray implementation differs from the old one in many details which make a direct comparison impossible. We compared efficiencies of two programs which only differ in the mechanism which is used for the communication (similarly as in the subsection above on the settings of the constants M and T). One program uses the TPL library with an event-driven mechanism, the other one uses the TPL library with polling. The programs are otherwise identical (even on the binary level). We only made the measurements for 90 workers where the memory limit was set to ca. 5%. We used a slightly modified `HAUS6` scene in this experiment (with fewer light sources and a different camera). We used the setting of $M = 5$ (pixels) and $T \to \infty$ (and no work stealing). This setting (chunking) excludes a significant imbalance and a bottleneck at loadbalancer. We measured an efficiency of ca. 0.0178 for the event-driven version and ca. 0.0015 for the polling version.

4.4.6 Further extensions and improvements

It may take a long time to read the scene description from a large file to the memories of the worker processes, especially if the objects are modeled as triangle meshes. This issue becomes critical when the parallel program is running on a disk-less machine which is connected to a disk server via a slow communication link. For instance, if 100 processes read the same file of size 100 MB, then 100x100 MB=10 GB must be transferred via the slow link. This transfer can take longer than the parallel computation itself. The solution involves the reading of the file in one of the workers only. This worker broadcasts the data which it reads among other workers. Broadcasting is usually much faster than disk operations involving the same volume of data. [Pla98]

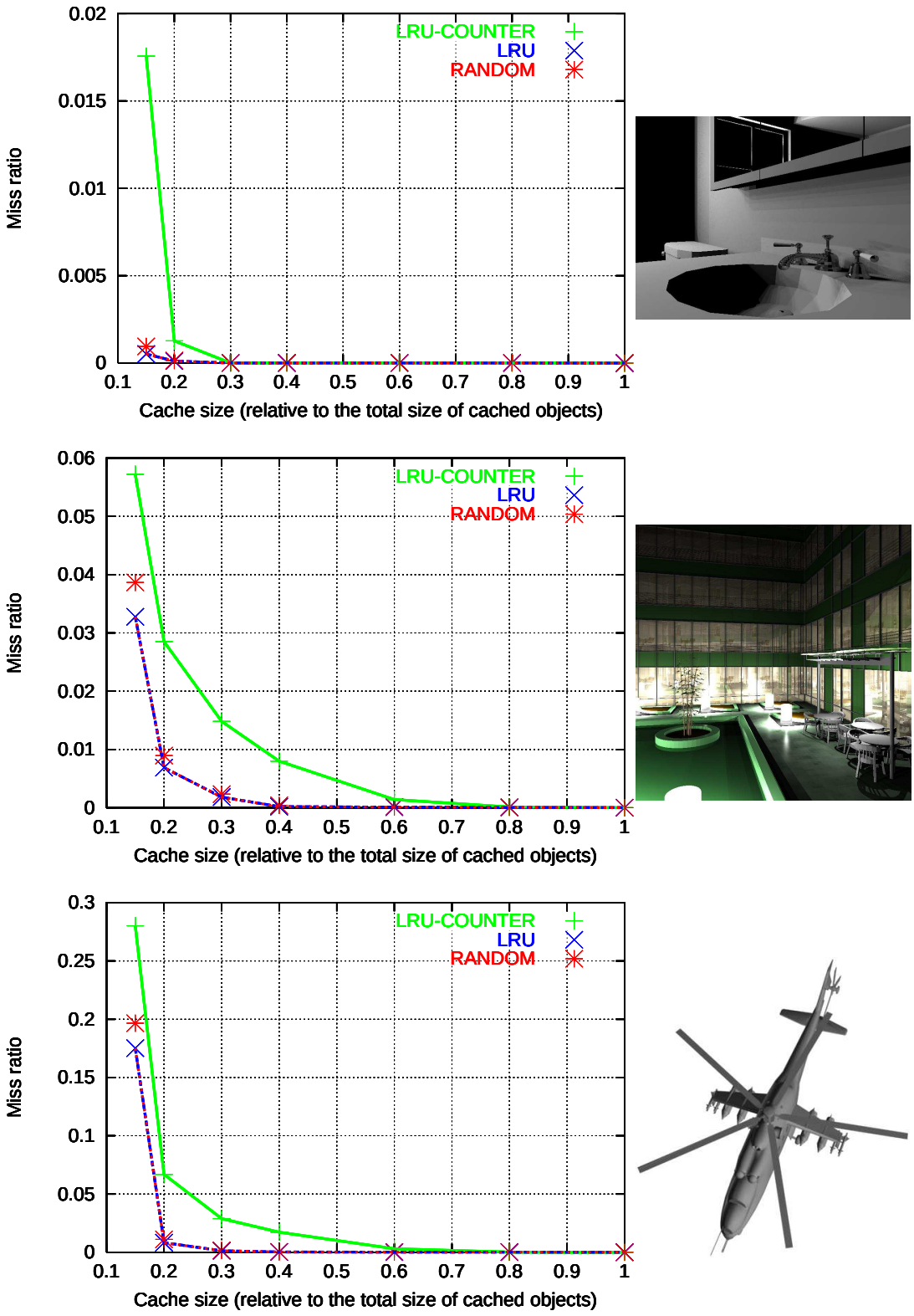

Figure 4.9: Cache miss ratios. Top: `BATH`, 353 objects. Centre: `ROSENTHALERHOF`, 2215 objects. Bottom: `HELICOPTER`, 167 objects

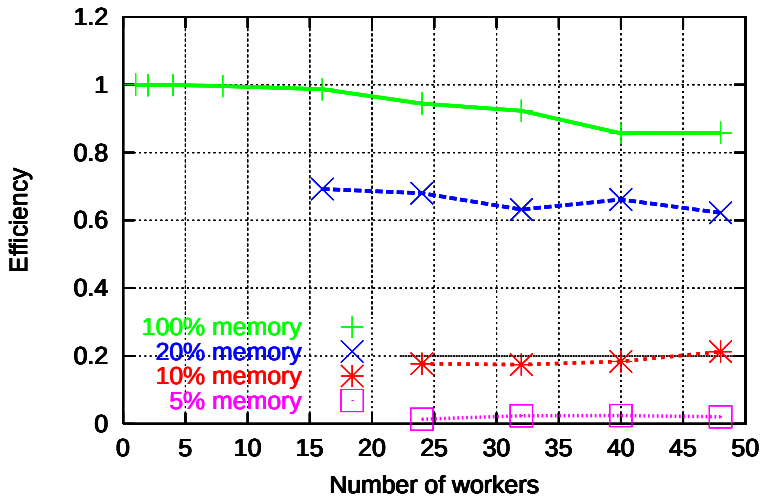

Figure 4.10: Efficiency of POV||Ray (an old polling version) with a distributed object database. The memory percentage states how large part of the sum of all object data sizes is allowed to be stored in the memory of each worker. A worker is only allowed to use this amount of memory for the storage of objects which it owns and for the object cache. The missing data in the graph indicate the cases where this simulated memory limit was exceeded in some worker

The sequential anti-aliasing optimisation technique which measures the differences between already computed pixel colours in order to reduce the number of supersampling primary rays cannot be directly used in parallel screen subdivision without a loss of efficiency. If this technique is applied directly, then primary rays for the pixels on the job borders will be computed twice. We use additional communication in order to avoid this double computation. [Pla02a]

An interesting practical issue is persistence of data. The workers can remain running after they have finished a computation of one image, and keep the scene description in their memories. It is possible to define a protocol between an external frontend program (which implements the user interface) and the backend (the parallel program itself) which allows the user to interact with the parallel program. We used this scenario in order to render camera animations without the need of re-reading the entire scene from a file (only a short camera description must be parsed by the workers in order to render the next frame). The use of persistent data in the context of animation allows for further optimisations of the parallel ray tracing algorithm which exploit the temporal coherence between subsequent frames. [FHK97]

Our current implementation of POV||Ray does not address the diffuse interreflection [WRC88] which is implemented in the sequential POV-Ray. The mechanism which is needed in order to share the irradiance database distributed in the memories of workers is similar to the sharing of the distributed object data.

The difference between the two is that the irradiance database quickly changes during the parallel computation and a change in one worker must be passed to other workers as soon as possible. [Rei96], [RCJ99]

4.5 Conclusions

We presented a simple and robust parallelisation of ray tracing. The parallelisation is based on screen space subdivision. We proposed a demand-driven load balancing algorithm which is a generalisation of chunking approaches. We proved that this algorithm guarantees a perfect load balance while minimising the number of work requests if its two parameters are optimally set. The optimal setting of these two parameters is generally unknown before the actual computation finishes. However, both the parameters have an intuitive interpretation and their optimal setting can be characterised. We proposed an automatical tuning procedure for these parameters and illustrated the procedure on experiments.

We addressed the problem of large scenes which cannot be copied into memories of the parallel processes. The parallel program uses a distributed object database which maintained by the same processes which use the data. (The use of a distributed object database makes the parallel ray tracing application non-trivial—each worker process performs the ray tracing computations and independently it acts as a database server for other worker processes.) We compared several cache strategies out of which we presented three which do not require any tuning. The LRU (Last Recently Used) strategy performed best.

We showed that the choice of the communication library strongly influences the performance of the parallel program. We compared two programs which are identical on the binary level—the only difference between the two is the choice of a mechanism inside the communication library. One program uses polling in order to achieve thread-safety and an independent message progress, the other program uses an event-driven mechanism. None of these mechanisms uses special optimisation techniques, the implementation of the mechanisms is practically identical to the code snippets from Chapter 2. The event-driven version outperformed the polling version in all experiments (for all problem instances and in all runs).

Chapter 5

Radiosity

The radiosity method solves a discretised version of the radiance equation 3.30 or a discretised version of the potential equation 3.31 or a combination of both as sketched in Section 3.4.2. The discretised radiance and potential equations are linear equation systems. We will focus on the discretised radiosity equation Equation 3.59 which can be written as

$$\mathbf{MB} = \mathbf{E} \tag{5.1}$$

where $M_{ij} = \delta_{ij} - \rho_i F_{ij}$ are the elements of the radiosity matrix $\mathbf{M}$. The symbol δ_{ij} denotes the Kronecker delta which is 1 for $i = j$ and 0 otherwise.[1]

There are several approaches to solving a linear equation system. An example of a simple approach is Gauss elimination which was used e.g. in [GTGB84], (the first paper which describes the use of radiosity for image synthesis). One disadvantage of Gauss elimination (or other full-matrix methods) is that the matrix of the system must be completely computed—this means that the form factors between all pairs of patches must be computed. This approach is not feasible for large matrices because the computation of each form factor involves a double integration (Equation 3.57).

Most radiosity algorithms use *Southwell relaxation* as the underlying linear equation solver.[2] [CCWG88], [SP94] The idea is to compute an acceptable approximation of the radiosity solution without having to compute the whole radios-

[1] The form factor $F_{ii} = 0$, therefore $M_{ii} = 1$ for $i = 1, \ldots, n$.

[2] An interesting alternative approach is presented in [Ash01]. The idea is to express the (slightly modified) radiosity matrix $\mathbf{M}$ using the spectral decomposition theorem as $M = \sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^T$, where λ_i are the eigenvalues of the matrix M and v_i are the corresponding eigenvectors. The radiosity matrix can be approximated as $M \approx \sum_{i=1}^{p} \lambda_i \mathbf{v}_i \mathbf{v}_i^T$, $p < n$, where the eigenvalues have been sorted in a decreasing order: $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$. It has already been shown that only the first few largest eigenvalues carry significant information for typical radiosity matrices. Hence, even if p is much less than n, the approximated radiosity solution is close to the exact radiosity solution. From a practical point of view, it is important that the computation of the first few largest eigenvalues (and the corresponding eigenvectors) does not require full knowledge of the matrix M if e.g. the Block Lanczos algorithm is used. [GU77]

ity matrix. Southwell relaxation is an iterative method which computes a series of approximations of the radiosity solution. The more iterations are performed, the better the final approximation is. As a result of the physical constraints, the radiosity matrix $\mathbf{M}$ is diagonally dominant which ensures the convergence of Southwell relaxation.

5.1 Southwell relaxation

In order to give a convenient physical interpretation of Southwell relaxation, Equation 5.1 can be slightly reformulated. We denote the total energy radiated by the patch P_i (A_i is the area of the patch P_i) as $\beta_i = B_i A_i$. Similarly, we denote the total energy emitted by the patch P_i as $\epsilon_i = E_i A_i$. In this notation, the radiosity equation can be written as

$$\mathbf{K}\beta = \epsilon \tag{5.2}$$

where $K_{ij} = \frac{A_i}{A_j} M_{ij}$. The vector of the unknowns β_i after k-th iteration of Southwell relaxation is denote as $\beta^{(k)}$. The residual vector after k-th iteration is denoted as $\mathbf{r}^{(k)} = \epsilon - \mathbf{K}\beta^{(k)}$. (The matrix $\mathbf{K}$ is diagonally dominant, therefore the iteration process converges: $\lim_{k\to\infty} \|\mathbf{r}^{(k)}\| = 0$.) In each iteration, Southwell relaxation choses the maximal residuum $r_s^{(k-1)}$ from the residual vector of the previous iteration and sets the new residuum to zero: $r_s^{(k)} = 0$. This yields the following recurrent formulas for β and $\mathbf{r}$: [SP94]

$$\beta_s^{(k)} = \beta_s^{(k-1)} + r_s^{(k-1)} \tag{5.3}$$

$$r_i^{(k)} = r_i^{(k-1)} - K_{is}\, r_s^{(k-1)} = \begin{cases} 0 & \text{for } i = s \\ r_i^{(k-1)} + \rho_i\, F_{si}\, r_i^{(k-1)} & \text{for } i \neq s \end{cases} \tag{5.4}$$

The natural initial settings of β and $\mathbf{r}$ are $\beta^{(0)} = \mathbf{0}$ and $\mathbf{r}^{(0)} = \epsilon$. Note that after k-th iteration, $\frac{\beta_i^{(k)} + r_i^{(k)}}{A_i}$ is a good estimate of the patch radiosity B_i. These radiosity estimates rapidly converge to the exact radiosity solution already after a few iterations. [CCWG88]

Several points are worth mentioning:

- The residua $r_i^{(k)}$ correspond to the *yet unshot patch energies* (the energies to be distributed in the future iterations $k+1, k+2, \ldots$).
- The values $\beta_i^{(k)}$ correspond to the *accumulated patch energies*.
- In one iteration k, the unshot patch energy $r_s^{(k-1)}$ is "shot" to all other $n-1$ patches and the contributions of this "shot" are added to the residua of the $n-1$ patches. The patch P_s whose patch energy is being shot is called a *shooting patch*.

- In each iteration k, the radiosity estimates $\frac{\beta_i^{(k)}+r_i^{(k)}}{A_i}$ are updated for all patches P_i except for the shooting patch P_s with the maximal residuum $r_s^{(k)}$.
- The same patch P_s can have the maximum residua $r_s^{(k_1)}$ and $r_s^{(k_2)}$ in different iterations k_1 and k_2. Hence, the same patch can be selected several times to be the shooting patch during the iteration process.
- The *total yet unshot energy* $\|\mathbf{r}^{(k)}\| = \sum_{i=1}^{n} r_i^{(k)}$ is a good estimate of the global error after k iterations of Southwell relaxation. The fraction $\frac{\|\mathbf{r}^{(k)}\|}{\|\epsilon\|}$ is the percentage of the initial total unshot energy which has not yet been shot. This percentage can serve as a convenient termination criterium of the iteration process.
- One iteration only involves the computation of $n-1$ form factors (the form factors between the shooting patch P_s and all other $n-1$ patches). As the number of iterations performed is usually much smaller than n, the majority of the form factors do not need to be computed in order to obtain the desired approximation of the radiosity solution.

5.1.1 Shooting radiosity algorithm

The algorithm in Fig. 5.1 is based on Southwell relaxation which has been described above. This algorithm is referred to as the *shooting radiosity algorithm*. The variables B_i represent patch radiosities ($B_i = (\beta_i^{(k)} + r_i^{(k)})/A_i$ after k executions of the `while` loop). The variables ΔB_i represent patch unshot radiosities ($\Delta B_i = r_i^{(k)}/A_i$ after k executions of the `while` loop).

The most expensive part of the algorithm is the computation of the form factors between the shooting patch P_s and the receiving patch P_r (the function `compute_form_factor`). The larger the equation system, the more iteration steps are required in order to reach the termination criterium and the more form factors must be computed in each iteration step. It is therefore desirable to use as few patches as possible for the scene representation (for instance, a rectangular wall should be modeled as two triangles, not more).

The radiosity algorithm in Fig. 5.1 explicitly enumerates the receiving patches (this happens in the *for* loop which follows the shooting patch selection). A different approach—direction sampling (an implicit form factor computation)—can be found e.g. in [Kel94] and [Kel96]

5.2 Form factor computation

The computation of form factors between pairs of patches in Fig. 5.1 (Equation 3.57) is the basis of the shooting algorithm. The approaches to the compu-

```
/* Input:
   patches P_1, ..., P_n
   patch reflectances ρ_1, ..., ρ_n
   patch emittances E_1, ..., E_n
   desired accuracy p
*/
shooting_radiosity()
{
   /* Initialisation. */
   for (i = 1; i < n; i++)
   {
      B_i = 0;
      ΔB_i = E_i;
      A_i = patch_area(P_i);
   }

   /* Southwell iterations. */
   while (∑_{i=1}^{n} A_i ΔB_i > p)
   {
      P_s = select_shooting_patch(); /* A_s ΔB_s = max_i A_i ΔB_i */
      for (r = 1; r < n; r++)
      {
         F = compute_form_factor(P_s, P_r);
         inc = ρ_r F A_s ΔB_s / A_r;
         B_r += inc;
         ΔB_r += inc;
      }
      ΔB_s = 0;
   }
}
/* Output:
   patch radiosities B_1, ..., B_n
*/
```

Figure 5.1: The basic shooting radiosity algorithm

tation of form factors can be divided into four classes: [SP94]

1. **Direct analytical integration.** Exact form factors are known for many special configurations of the two patches. [How82] The known configurations in Howell's catalogue are divided into three groups: differential area to differential area (e.g. two differential areas in an arbitrary configuration), differential area to finite area (e.g. a differential planar element to finite parallel rectangle) and finite area to finite area (e.g. two identical, parallel, directly opposed rectangles). Howell's catalogue assumes that the two patches are not occluded by any other patch. Some radiative heat transfer applications work with 3D models without occlusions. Unfortunately, in most models relevant to computer graphics applications it is even impossible to predict whether there is an occlusion between two patches or not (there generally is).

2. **Contour integration.** Using Stokes' theorem, the area integral of Equation 3.57 can be transformed into a contour integral:

$$F_{sr} = \frac{1}{2\pi A_s} \oint_{C_s} \oint_{C_r} \ln r \; dc_r \; dc_s \tag{5.5}$$

   where C_s and C_r are the contours of the patches P_s and P_r.

   A form factor between a point x (a differential area) and a polygon $P = [v_1, v_2, \ldots v_{maxp}]$ can be analytically carried out using contour integration if no occlusion exists between the point and the polygon: [HS67], [SH93]

$$F_{xP} = \frac{1}{2\pi} \sum_{i=1}^{maxp} N \cdot \frac{\angle(R_i, R_{i\oplus 1})}{|R_i \times R_{i\oplus 1}|}(R_i \times R_{i\oplus 1}) \tag{5.6}$$

   where N is the normal at the point x and $\oplus$ is the "circular next" operator on $1, \ldots, maxp$. $\angle$ denotes the (signed) angle between two vectors. R_i is the vector from x to v_i.

3. **Projection.** Projection methods are based on Nusselt's analogy which provides an alternative definition of a form factor between a differential area around a point x and a finite surface patch P: [Nus28], [SP94]

   > The form factor F_{xP} is the fraction of the area in the base plane[3] which is obtained by projecting the patch P onto the unit hemisphere centered at the point x, and then orthogonally down onto the base plane.

[3] The base plane is the plane which contains the point x and which is perpendicular to the normal at the point x.

A popular algorithm which is based on the above projection is the hemicube algorithm (a hemicube or an arbitrary surface around the point x can be used instead of the hemisphere in the above definition). [CG85] The advantages of the hemicube algorithm are that it can deal with an occlusion between two patches, it is relatively simple and it can use the hardware of contemporary graphics cards (the z-buffer algorithm is implemented in the hardware). One disadvantage is that the precision of the form factor approximation depends on the discretisation of the hemicube. An insufficient discretisation leads to severe inaccuracies.

4. **Monte Carlo integration (ray casting).** Monte Carlo integration [ES00] is the most straightforward method of computing the form factor between two patches P_s and P_r which may be occluded by other patches. The advantage of Monte Carlo integration is generally its flexibility. Care must be taken when choosing the estimator. Estimators which are unbiased and which have a low variance are preferred. Estimators which have both these properties are proposed in [Pie93] and [Bek99]. Unlike the projection methods, Monte Carlo integration is not confronted with aliasing problems. The only parameter which must be tuned is the number of samples.

We will focus on the methods of Monte Carlo integration in the next section.

5.2.1 Monte Carlo form factor computation

Direct area estimator

A direct approach to the computation of the factor F_{sr} uniformly generates N_s points $x_i,\ i = 1, \ldots, N_s$ on the patch P_s and N_r points $y_j,\ j = 1, \ldots, N_r$ on the patch P_r and it uses the estimator

$$\hat{F}_{sr} = \frac{1}{N_s\ N_r\ \pi\ A_s} \sum_{i=1}^{N_s} \sum_{j=1}^{N_r} \frac{\cos\theta_{ij}\ \cos\theta'_{ij}}{r(x_i, y_j)^2}\ V(x_i, y_j) \tag{5.7}$$

where θ_{ij} is the angle between the surface normal of the patch P_s and the direction from x_i to y_j, θ'_{ij} is the angle between the surface normal of the patch P_r and the direction from y_j to x_i, $r(x_i, y_j)$ is the distance between the points x_i and y_j and V is the visibility function defined in Equation 3.52.

The estimator in Equation 5.7 is unbiased. This means that its expected value is correct ($E[\hat{F}_{sr}] = F_{sr}$). However, its variance can be very high and even unbounded: [Bek99]

$$var[\hat{F}_{sr}] = \frac{A_r}{\pi^2\ A_s} \int_{x \in P_s} \int_{y \in P_r} \left(\frac{\cos\theta\ \cos\theta'}{r(x, y)^2}\ V(x, y) \right)^2 dA_r\ dA_s - F_{sr}^2 \tag{5.8}$$

The problem with the unbounded variance is caused by the factor $r(x,y)^4$ in the denominator of 5.8 and it shows up for abutting patches. In this case, increasing the number of samples does not necessarily improve the form factor estimate.

Delta area estimator

The approach of [WEH89] uniformly generates N_s points $x_i,\ i = 1,\ldots,N_s$ on the patch P_s and N_r points $y_j,\ j = 1,\ldots,N_r$ on the patch P_r. For a pair of points x_i and y_j, an analytical form factor from the differential area around the point x_i to a disk around the point y_j is calculated in order to approximate the inner form factor integral (the sum of the N_r disk areas is equal to A_r). This yields the estimator

$$\hat{F}_{sr} = \frac{1}{N_s\ N_r\ \pi\ A_s} \sum_{i=1}^{N_s} \sum_{j=1}^{N_r} \frac{\cos\theta_{ij}\ \cos\theta'_{ij}}{r(x_i,y_j)^2 + \frac{A_r}{N_r}}\ V(x_i,y_j) \tag{5.9}$$

This estimator is biased ($E[F_{sr}] \neq F_{sr}$) but it is consistent ($E[F_{sr}] = F_{sr}$ for $N_r \to \infty$). The bias can only be neglected if N_r is large enough.

Directional estimator

The inner form factor integral (Equation 3.57) can be written as a directional integral, which yields an equivalent form factor formula

$$F_{sr} = \frac{1}{\pi\ A_s} \int_{x\in P_s} \int_{\omega\in\Omega_r(x)} \cos\theta\ d\omega\ dA_s \tag{5.10}$$

where $\Omega_r(x_i)$ denotes the solid angle subtended by the visible part of the patch P_r as seen from the point x, θ is the angle between the surface normal at x and the direction ω and dA_s is a differential area around the point x.

The determination of the integration domain $\Omega_r(x_i)$ involves solving the visibility problem. The visibility can be analytically solved in this case but this analytical computation is very expensive. [BRW89] Equation 5.10 can be reformulated as

$$F_{sr} = \frac{1}{\pi\ A_s} \int_{x\in P_s} \int_{\omega\in\Omega'_r(x)} \cos\theta\ V'(x,\omega,P_r)\ d\omega\ dA_s \tag{5.11}$$

where $\Omega'_r(x)$ denotes the solid angle subtended by the patch P_r as seen from the point x. The function $V'(x,\omega,P_r)$ returns 1 if $RT(x,\omega) \in P_r$ and 0 otherwise.

The directional integration uniformly generates N_s points $x_i,\ i = 1,\ldots,N_s$ on the patch P_s. For each point x_i, directions $\omega_j,\ j = 1,\ldots,N_r$ are uniformly generated over the solid angle $\Omega_r(x_i)$ subtended by the patch P_r as seen from the point x_i. The resulting form factor estimator is

$$\hat{F}_{sr} = \frac{1}{N_s \; N_r \; \pi \; A_s} \sum_{i=1}^{N_s} \sum_{j=1}^{N_r} \cos\theta_{ij} \; V'(x_i, \omega_j, P_r) \tag{5.12}$$

This estimator is unbiased and its variance is always bounded: [Bek99]

$$\begin{aligned} var[\hat{F}_{sr}] &= \frac{1}{\pi^2 \; A_s} \int_{x \in P_s} \Omega'_r(x)^2 \int_{\omega \in \Omega'_r(x)} \left(\cos\theta_{ij} \; V'(x, \omega, P_r)\right)^2 \; d\omega \; dA_s - F_{sr}^2 \\ &\leq 4 \end{aligned} \tag{5.13}$$

A technical problem remains and that is how to uniformly sample the directions ω in the solid angle $\Omega'_r(x)$. If the patch P_r is a triangle, then P_r can be projected onto a hemisphere around the point x and the sampling technique for spherical triangles can be applied. [Arv95]

Weighted analytical estimators

The idea behind the following estimators is to uniformly generate N_s points $x_i, \; i = 1, \ldots, N_s$ on the patch P_s and to analytically compute the unoccluded point-to-patch form factor $F_{x_i P_r}$ (Equation 5.6) for each point x_i. This form factor is weighted by the visibility sampling. For each point x_i, N_r points $y_{ij}, \; j = 1, \ldots, N_r$ are uniformly generated on the patch P_r. The visibility function $V(x_i, y_{ij})$ is computed for the point x_i and the points $y_{ij}, \; j = 1, \ldots, N_r$.

The resulting estimator which is proposed in [Pie93] is

$$\hat{F}_{sr} = \frac{1}{N_s \; N_r} \sum_{i=1}^{N_s} F_{x_i P_r} \sum_{j=1}^{N_r} V(x_i, y_{ij}) \tag{5.14}$$

A similar estimator is proposed in [Bek99] (weighted area sampling):

$$\hat{F}_{sr} = \frac{1}{N_s} \sum_{i=1}^{N_s} F_{x_i P_r} \frac{\sum_{j=1}^{N_r} \left(\frac{\cos\theta_{ij} \;\; \cos\theta'_{ij}}{\pi r(x_i, y_{ij})^2} \; V(x_i, y_{ij}) \right)}{\sum_{j=1}^{N_r} \frac{\cos\theta_{ij} \;\; \cos\theta'_{ij}}{\pi r(x_i, y_{ij})^2}} \tag{5.15}$$

Both of the estimators above are unbiased and their variance is bounded by $\frac{1}{N_s} \int_{x \in P_s} F_{xP_r}^2 \, dA_s$. [Bek99] *In our opinion these two estimators are the best known ones in connection with the shooting radiosity method.*

5.3 Discretisation of surface geometry

The surface discretisation (also known as meshing) is a conversion of the surface geometry of a 3D model into a polygonal representation. This polygonal representation is usually a set of triangle meshes. The illumination computed by a radiosity algorithm is stored in this mesh (usually in the vertices of the

triangles). The initial mesh is created in the preprocessing stage and it is dynamically refined by the radiosity algorithm in order to accurately represent the stored illumination.

Mesh generation has been the subject of many research papers, especially in relation to the finite element methods. Specific requirements of mesh generation for the purpose of radiosity computations were formulated in [BMSW91]. These specific requirements place constraints on the size of the triangles and on the topology of the input mesh. *We will show that none of these constraints are important if suitable techniques are used in the radiosity algorithm. We propose a radiosity algorithm which works well with an arbitrary input mesh*—the only requirement is, that the triangles are numerically interpreted as triangles (that means, that the numerically computed area of any triangle must not be 0). The initial size of the triangles is also not important—the larger they are, the better. Some radiosity algorithms (e.g. [Sch00]) work with triangles and quadrangles—we only allow the use of triangles in order to keep the algorithm simple (this only influences the computational time, not the quality of the radiosity solution).

5.4 Illumination storage and reconstruction

The assumption of the radiosity method is constant radiosity over a patch. However, as the diffuse illumination usually smoothly varies over a surface, the storage of a single RGB value per patch leads to unpleasant visual artifacts. These artifacts are caused by the discontinuities of the illumination on the borders between the patches. This problem can be solved by making the patches very small—however, this would increase the computational time. A usual practice is to store different RGB values in the vertices of the patches.[4] These RGB values are called *vertex radiosities*. The illumination at a point of the patch is reconstructed using the interpolation of the vertex radiosities of the patch. As patches are two-sided, the illumination must be independently stored for the front side and the back side of each patch (see Section 3.2.2).

The illumination over a patch is not always smooth. It can vary rapidly in particular on shadow boundaries. The linear interpolation is not able to capture sudden changes. This problem can also be solved by using smaller patches but it is desirable to keep the number of patches as small as possible. A commonly used compromise is using an *adaptive hierarchical subdivision*. The initial patches are as large as possible (e.g. a planar wall is represented as two triangles, regardless of its size). A patch is only subdivided when a significant discontinuity of the illumination is detected on its surface. The resulting subpatches can be further subdivided in a similar manner.

[4] An alternative is to represent the illumination over a patch as a linear combination of a finite number of base functions (the so-called Galerkin method).

A patch only needs to be subdivided at the moment when its vertex radiosities are being updated. This only happens when the patch is acting as the receiving patch during the radiosity algorithm and when its current level of subdivision is not able to capture the illumination.

The push-pull algorithm can be used to maintain the patch radiosities and patch unshot radiosities at each subdivision level. [SP94] The radiosity stored in a node of the tree is equal to the average radiosity stored in the nodes' children. This is important for algorithms which perform the energy exchange between the shooting patch and the receiving patch at different subdivision levels.

Our algorithm also uses the adaptive hierarchical subdivision. However, the energy exchange always takes place on the top level of the hierarchy—that means, the shooting patch and the receiving patch are the original (large) patches. The tree data structure provides two operations for the storage and retrieval of radiosity and unshot radiosity (these are both represented as RGB values):[5]

- `retrieve(`P`,` $side$`,` x`,` c`)` retrieves the RGB value at the point x on the side $side$ of the patch P. This retrieved value is stored into c. The retrieval procedure first traverses the hierarchy of the patch P on the side $side$ in order to find the smallest triangle (the smallest triangle is always a leaf of the subdivision tree) which contains the point x. Then the RGB values which are stored in the vertices of this triangle are interpolated in order to compute the value c.

- `store(`P`,` $side$`,` x`,` c`)` stores the RGB value c at the point x on the side $side$ of the patch P. The storing procedure first traverses the hierarchy of the patch P in order to find the smallest triangle which contains the point x. Then it updates the RGB values which are stored in the vertices of this smallest triangle and checks whether the interpolation of the new vertex values at the point x differs from c. If the difference is larger than a user-specified threshold, the triangle is subdivided into smaller non-overlapping triangles[6] and the update is recursively repeated for that of these new triangles which contains the point x. When this recursion returns, the vertex values are "pulled" from the smallest triangle up to the root of the tree.

The (point-based) `retrieve(`P`,` $side$`,` x`,` c`)` operation always retrieves the best level of detail of the illumination at the point x. The point-based `store` operation is more general than a patch-based `store` operation. Indeed, a patch-based `store(`P`,` $side$`,` c`)` operation can be simulated using the point-based operation `store(`P`,` $side$`,` x_{center}`,` c`)`, where x_{center} is the center of the patch P. The patch-based `retrieve` operation (this operation is needed for the selection of

[5] Similar operations are proposed in [Bek99] (per-ray refinement).

[6] All the smaller triangles must still numerically be interpreted as triangles (with non-zero areas). If this assertion fails then the subdivision of the current node is discarded.

the shooting patch) simply returns the average of the vertex radiosities of the top-level patch.

Remark. The illumination can be stored in any data structure which provides the `store` and `receive` operations. The data structure must additionally allow for an efficient generation of sample points on the shooting patch which is described in Section 5.5.1 (Fig. 5.4). •

5.5 Energy transfer

The transfer of energy from the shooting patch P_s to the receiving patch P_r in the basic radiosity algorithm in Fig. 5.1 is comprised in the *while* loop which follows the shooting patch selection. The actual energy transfer is preceded by the form factor computation. Our algorithm combines these two steps (and it retains the explicit enumeration of the receiving patches).

We will focus on the Monte Carlo form factor computation (see Section 5.2.1). In order to explain the proposed energy transfer mechanism, we will first consider the simple direct area form factor estimator (Equation 5.7). This method uniformly generates N_s points $x_i,\ i = 1, \ldots, N_s$ on the patch P_s and N_r points $y_j,\ j = 1, \ldots, N_r$ on the patch P_r. Equation 5.7 sums up the contributions of all point pairs $[x_i, y_j]$ to the estimated form factor $\hat{F}_{sr}$. We can merge this step with the computation of *inc* in Fig. 5.1—instead of working with the *form factor* estimator $\hat{F}_{sr}$ we can directly work with the *added radiosity* estimator

$$\widehat{inc} = \frac{\rho_r\ \Delta B_s}{N_s\ N_r\ \pi\ A_r} \sum_{i=1}^{N_s} \sum_{j=1}^{N_r} \frac{\cos\theta_{ij}\ \cos\theta'_{ij}}{r(x_i, y_j)^2}\ V(x_i, y_j) \tag{5.16}$$

Let us assume that there are N_s point light sources located at the points $x_i,\ i = 1, \ldots, N_s$. Their emittances are

$$E(x_i, \omega) = \begin{cases} \frac{\cos\theta}{r(x_i,y)^2} & \text{if } \omega \text{ lies in the half-space of the surface normal at } x_i \\ 0 & \text{otherwise} \end{cases} \tag{5.17}$$

where θ is the angle between the surface normal at the point x_i and the direction ω. The attenuation term $r(x_i, y)^2$ is artificial. It expresses that the emittance is inversely proportional to the square of the distance to the point illuminated by the light source. If the N_s point light sources are the only light sources in the model, then the sum

$$\rho_r \sum_{i=1}^{N_s} \frac{\cos\theta_{ij}\ \cos\theta'_{ij}}{r(x_i, y_j)^2}\ V(x_i, y_j) \tag{5.18}$$

is equal[7] to the direct illumination term which is computed by the ray tracing shader at the point y_j. Equation 5.18 is the inner sum of Equation 5.16 multiplied by ρ_r (if the sums in Equation 5.16 are swapped). If the ray tracing shader is applied to all N_r points on the patch P_r, then the sum of the resulting "colours" scaled by $\frac{\Delta B_s}{N_s\ N_r\ \pi\ A_r}$ is equal to the right side of Equation 5.16. This final summation can be expressed using the `store` operation (see Section 5.4). The "colours" c_j computed by the ray tracing shader for the receiving samples y_j, $j = 1, \ldots, N_r$ are scaled and added to vertex radiosities of the patch P_r: `store(`P_r`,` *side*`,` y_j`,` $c_j * \frac{\Delta B_s}{N_s\ N_r\ \pi\ A_r}$`)`. The idea behind using the ray tracing shader for the energy exchange in the radiosity algorithm is illustrated in Fig. 5.2.

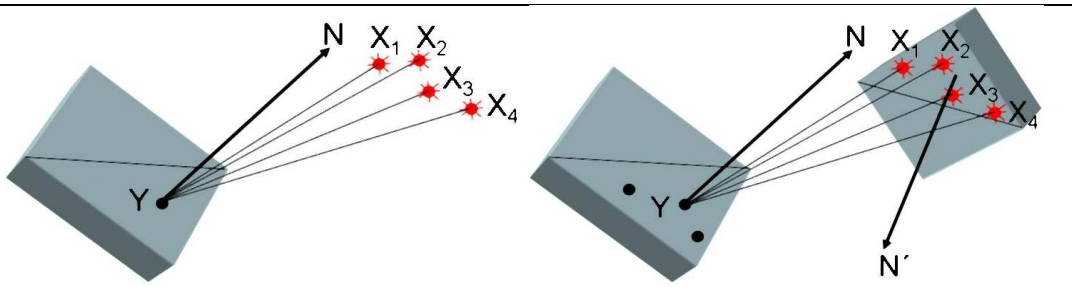

Figure 5.2: Radiosity energy exchange using the ray tracing shader. Left: shadow rays are traced by the ray tracing shader from the light sources in order to illuminate a surface point Y. Right: shadow rays are traced by the ray tracing shader in order to transfer energy from the shooting patch to the receiving patch. The temporary point light sources are randomly generated on the shooting patch

Remark. The construction above can be easily adapted to use a better form factor estimator of Section 5.2.1, for instance the weighted analytical estimator (Equation 5.14). We show in Section 5.7.1 that the weighted analytical estimator outperforms the direct area form factor estimator which we used in the construction above for explanatory purposes. •

Remark. The idea of using artificially generated light sources for the purpose of the computation of the indirect diffuse illumination can also be found in [Kel97]. Keller's method does not use any permanent illumination storage—it is "view-

[7] More precisely, Equation 5.18 is equal to the direct illumination term which is computed by the ray tracing shader only if two simplifying assumptions hold in the 3D model: 1.all patches are opaque, 2.all patches are pure diffuse reflectors described by the scalar ρ_r. Ray tracing shaders usually work with more general material descriptions. If the materials in the model do not satisfy the previous assumptions then the "colour" returned by the ray tracing shader may differ from Equation 5.18 because the shader takes into account all material properties, not only the scalar ρ_r.

dependent". The method shoots the original light sources onto randomly generated points on a patch and creates a new temporary point light source in each of these points. An image of the scene is then rendered, whereby the scene is illuminated by the temporary light sources. This process is repeated for each patch and the final image is obtained as a sum of the single images. The rendering step makes use of graphics hardware. The algorithm only computes diffuse scatterings of level one (the term $\mathcal{R}L$ of Equation 3.35). Scatterings of level two and higher (the terms $\mathcal{R}^i L,\ i \geq 2$) can be added by extending the lengths of the light paths using a direction sampling of the temporary light sources. (Rays are traced in randomly generated directions and additional temporary light sources are generated if in the surface points which are hit by the rays.) •

5.5.1 Shooting radiosity algorithm using the ray tracing shader

The above reformulation of the energy transfer in terms of the ray tracing shader leads to the algorithm in Fig. 5.3. We will discuss a few algorithmic aspects hidden in the pseudo-code:

- Patch radiosities B_i and patch unshot radiosities ΔB_i do not need to be explicitly stored in the patches (and subpatches) because they can be reconstructed from the vertex radiosities and vertex unshot radiosities.

- The basic radiosity algorithm only works with patch light sources. The function `set_vertex_unshot_radiosities` in Fig. 5.3 may include the so-called *first shot* during which all other point light types are shot onto the patches P_i. This first shot can be implemented similarly to the shooting in one Southwell iteration—points are randomly generated on the receiving patches and the ray tracing shader is called to illuminate them. (It is important that the illumination acquired during the first shot is compatible with the direct ray tracing illumination.)

- The patch selected by the function `select_shooting_patch` is always the top-level patch (not a subpatch of the subdivision tree). Note that illumination is stored independently for front and back sides of every patch. The shooting patch selection routine must search both the front and back sides of each patch.

- An appropriate choice of the sampling rates (computed in the function `choose_sampling_rates`) requires a heuristics which should depend on the amount of the transferred energy $A_s\ \Delta B_s$ and on a rough estimation of the form factor F_{sr}. A discussion on sampling rates for the Monte Carlo form factor computations can be found in [Bek99].

```
/* Input:
   patches P_1, ..., P_n
   patch emittances E_1, ..., E_n
   desired accuracy p
*/
shooting_radiosity()
{
   /* Initialisation. */
   for (i = 1; i < n; i++)
   {
      zero_vertex_radiosities(B_i);
      set_vertex_unshot_radiosities(B_i, E_i);
   }

   /* Southwell iterations. */
   while (sum_{i=1}^{n} A_i ΔB_i > p)
   {
      P_s = select_shooting_patch(); /* A_s ΔB_s = max_i A_i ΔB_i */
      for (r = 1; r < n; r++)
      {
         choose_sampling_rates(P_s, P_r, N_s, N_r);
         generate_shooter_random_samples(P_s, x_1, ..., x_{N_s});
         store_light_sources();
         set_light_sources(x_1, ..., x_{N_s});
         generate_receiver_random_samples(P_r, y_1, ..., y_{N_r});
         for (j = 1; j < N_r; j++);
         {
            ray_tracing_shader(y_j);
         }
         restore_light_sources();
      }
      zero_vertex_unshot_radiosities(B_s);
   }
}
/* Output:
   patch radiosities B_1, ..., B_n
   vertex radiosities stored in the trees of patches B_1, ..., B_n
*/
```

Figure 5.3: Shooting radiosity algorithm using the ray tracing shader

- An efficient algorithm for uniform generation of random points inside a triangle can be found in [Gla90].
- The use of stratified sampling (also known as latin square sampling or jittering, see [ES00]) in the function `generate_random_samples` may reduce the variance of the Monte Carlo integration.
- The first three function calls in the `for (r=1; r < n; r++)` loop set up the temporary point light sources on the shooting patch. These three lines of code can be moved before the `for` loop which significantly speeds up the energy transfer. The idea behind this rearrangement is the reusing of the shooting samples (more precisely, the point lights) in shootings to different receiving patches. This leads to a significant increase of the efficiency, especially if the light buffer technique is applied. It is much more efficient to compute the light buffers once per iteration than once per a receiving patch.
- The function `ray_tracing_shader` is actually called twice—once for the front side of the receiving patch P_r and once for the back side. This function computes the direct illumination c_j at the points y_j and calls the function `store` in order to store the obtained "colours" c_j in the subdivision tree (after scaling).
- The function `set_light_sources` and the `store()` call in the function `ray_tracing_shader` can be adapted to (implicitly) compute the the form factor estimate 5.14 or 5.15 instead of 5.7. (Our implementation uses the form factor estimate of Equation 5.14.)

Random generation of sampling points

A particularly interesting issue is the generation of the sampling points on the shooting patch (`generate_shooter_random_samples`) and the receiving patch (`generate_shooter_random_samples`). The computation of form factors using Equation 5.14 requires a uniform sampling of both the shooting patch and the receiving patch. However, the following example shows that *it is not desirable to generate the samples uniformly on the shooting patch from the point of view of the energy transfer (we recall that both the shooting patch and the receiving patch are top-level patches)*. Let us assume that the unshot radiosity in one part of the shooting patch is much greater than in another parts. In an extreme case the whole unshot radiosity is accumulated in one leaf of the subdivision tree, while the unshot radiosity stored in the other leaves is zero. If the sampling points are generated uniformly on the shooting patch, then it may happen that none of the sampling points lies inside the leaf with the non-zero unshot radiosity. The `retrieve` operation would return 0 for all the generated points, therefore no energy would actually be shot at the receiving patches.

There are two solutions to this problem. (Note that the problem is only related to the shooting patch—the points on the receiving patch should be generated uniformly.) The first solution involves shooting smaller patches than the top-level patches—however, this would lead to a slow convergence of the shooting algorithm. The second solution involves a weighted sampling of the shooting patch. The idea behind this weighted sampling is to control the density of the generated points on the shooting patch according to the unshot radiosity over the shooting patch. The pseudo-code in Fig. 5.4 describes the point generation process. This process first chooses a leaf of the subdivision tree (a triangle) with a probability proportional to the unshot radiosity of the leaf and then generates the random point inside this leaf.

```
generate_shooter_random_samples(P_s, x_1, ..., x_Ns)
{
  for (i = 1; i <= N_s; i++)
  {
    node = P_s;
    while (node is not a leaf)
    {
      ... select a successor child of node with probability
      proportional to the patch unshot radiosity of child;
      node = child;
    }
    x_i = (uniform) random point on the patch of the node node;
  }
}
```

Figure 5.4: Generation of sample points on the shooting patch

It may be argued that the non-uniformity of the random sampling of the shooting patch violates the assumption of the form factor computation of Equation 5.14. However, the proposed process should be looked at as shooting of the group of leaf patches of the subdivision tree. Let us return to the previous example—let us assume that only one leaf patch stores the whole unshot radiosity of the shooting patch. In this case the random points will only be generated on this one leaf patch and the implicitly computed form factor will correspond to the form factor between this one leaf patch and the receiving patch. The effect of the subsequent shooting is therefore the same as if that one leaf patch was selected for the shooting.

Practical consequences

The proposed algorithm has a number of positive practical consequences (the only drawback is the mesh representation of the surface geometry):

- The visibility function $V(x_i, y_j)$ computed by the ray tracing shader is more general than its original definition in Equation 3.52 as the ray tracing shader automatically handles (non-scattering) transparency. Hence, the visibility function is no longer a function which returns either 0 or 1—it becomes a real function instead: $V(x_i, y_j) \in \langle 0, 1 \rangle$.

- Ray tracers can usually work with complex material descriptions which include layered textures, alpha channel, bump mapping etc. As the energy transfer in the radiosity algorithm invokes the ray tracing shader, the radiosity algorithm can work with the same material description.

- Only a few parameters must be supplied by the user: 1.desired accuracy of the radiosity solution (the constant p in Fig. 5.3), 2.threshold which controls the adaptive refinement (see Section 5.4), 3.parameter which controls the sampling density in the function `choose_sampling_rates`. All these parameters are intuitive as they are expressed as percentages.

- The radiosity computation can be incorporated into any ray tracer as a preprocessing step. The combination of radiosity and ray tracing yields the so-called two-pass solution of the global illumination problem. [SP89]

5.6 Visualisation

Ray tracing is used for the visualisation of the radiosity solution. The computation of the ambient and the direct illumination terms in the ray tracing shader is replaced by the `retrieve()` function call (this function is defined in Section 5.4).

The quality of the rendered two-pass images can be further improved if the direct illumination is subtracted from the radiosity solution and computed by the ray tracing shader (in this case only the ambient term is replaced by the `retrieve()` function call). Unless the size of the leaf triangles in the radiosity mesh is after the projection onto camera comparable to the size of the camera pixels, the direct illumination computed by ray tracing is more accurate.

The computed images include the direct illumination, indirect perfect specular illumination (multiple perfect specular scatterings) and indirect perfect diffuse illumination (multiple perfect diffuse scatterings). Transparency is also accounted for. Some light phenomena, e.g. caustics, are not included in the two-pass solution. These missing phenomena are related to photon paths along which indirect specular and diffuse scatterings are mixed (or photon paths with imperfect scatterings).

5.7 Experiments

We implemented the shooting radiosity algorithm in Fig. 5.3 as a module of the Persistence of Vision Ray Tracer (POV-Ray). The radiosity computation runs as a preprocessing step which is followed by ray tracing. The scene must consist of triangle meshes (more precisely, other object types may be used as well but the radiosity algorithm only stores the illumination in triangle meshes).

The implementation is fully functional even though not yet complete. An additional programming effort would be needed in order to implement extensions such as a heuristics for the dynamical control of the number of samples on the shooting patch and the receiving patch (a fixed number of samples is currently used), the adaptive substructuring, the storing of the computed illumination in a file etc. However, all major algorithmic issues have already been addressed. The missing details only influence the efficiency of the implementation.

5.7.1 Form factors

In order to verify the correctness of the implementation we first created a special 3D model for which the radiosity solution can be calculated analytically. The model consists of an empty box (a cube with 6 square walls) and a camera which views the inside of the box. As our implementation only uses triangles, each wall is modeled as two triangles. In the following experiments we also used a refined version of the box in which each wall is modeled as eight triangles. (We implemented a simple mesher which refines the model until all triangle edges are shorter than a given threshold.) These two versions, `BOX12` and `BOX48` are equivalent in the sense that they describe the same surfaces—the only difference is the number of triangles, see Fig. 5.5.

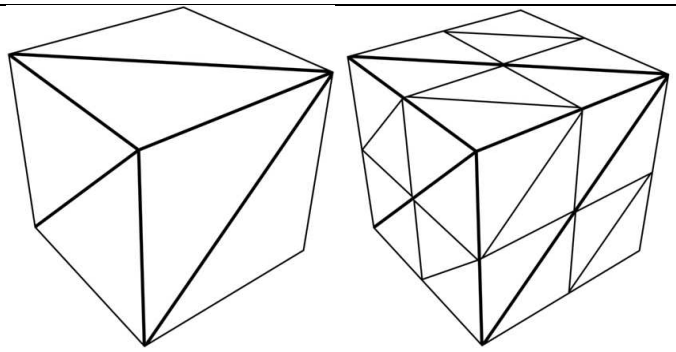

Figure 5.5: Left: `BOX12`, the box scene modeled of 12 top-level triangle patches. Right: `BOX48`, the same box scene modeled of 48 top-level triangle patches (level-one refinement)

There are two kinds of form factors in the scene: 1.the form factor between the floor and one of the vertical walls, F_{perp}; 2.the form factor between the floor

and the ceiling, F_{par}. The analytical formulas for both these form factors can be found in Howell's catalogue: [How82]

$$F_{perp} = \frac{1}{2} + \frac{\frac{1}{4}\ln\frac{3}{4} - \sqrt{2}\arctan\frac{1}{\sqrt{2}}}{\pi} \approx 0.200044 \tag{5.19}$$

$$F_{par} = \frac{4\sqrt{2}\arctan\frac{1}{\sqrt{2}} - \ln\frac{3}{4}}{\pi} - 1 \approx 0.199825 \tag{5.20}$$

Remark. F_{perp} and F_{par} are *not* equal. This is an interesting asymmetry which can rarely be observed in the nature for a symmetric configuration such as a cube. The difference between the two form factors is very small, which yields an uncommon approximation of π (if we set $F_{perp} = F_{par}$):

$$\pi \approx \frac{10\sqrt{2}}{3}\arctan\frac{1}{\sqrt{2}} - \frac{5}{6}\ln\frac{3}{4} \approx 3.141134 \tag{5.21}$$

•

It is not straightforward to verify the implementation of the form factor computation as the value of a form factor does not appear explicitly anywhere in the algorithm in Fig. 5.3. However, it is possible to modify the selections of the shooting patch and the receiving patch in Fig. 5.3 so that only all the triangles of the emitting floor will shoot energy at a particular receiving patch (the particular patch is one of the vertical walls and the ceiling, respectively). We also modified the algorithm so that it terminates after the energy of the floor has been shot. Even though we can not directly measure the value of the form factor, we can read the unshot patch radiosity values of the receiving patches after the shot. The unshot radiosity of a wall is equal to the sum of unshot radiosities of all its triangles. If we set the reflectivity ρ of the receiving patch to 1 and the unshot radiosity of the shooting patch to 1, then the unshot radiosity of the receiving patch after the shooting will be equal to the form factor between the shooting patch and the receiving patch. The side-effect of this methodology is the testing of the implementation of the function `store` and of the energy transfer.

We compare the direct area estimator of Equation 5.7 to the weighted analytical estimator of Equation 5.14. Using the procedure above we computed the form factors for the scenes `BOX12` and `BOX48` for a given number of samples per triangle in 100 independent runs (the number of samples on the shooting triangle was equal to the number of samples on the receiving triangle). We measured the minimum, maximum and average values over the 100 runs.

The results of these experiments are shown in Fig. 5.6. Note that the average values of the computed form factors match the exact values of the form factors in all graphs. However, the average is not relevant for the reliability of the implementation (unless the same shooting is repeated many times and the results are

averaged—but this is not a common practice as the visibility computation is the most expensive part of the radiosity algorithm). The variance is more relevant. Even more relevant than the variance are the minimum and the maximum values of the computed form factors over the 100 runs. (Note that a wrongly computed form factor value in an early iteration of the radiosity algorithm may invalidate the resulting radiosity solution.)

The weighted analytical estimator always outperforms the direct area estimator, especially in the computation of F_{perp}. The weighted analytical estimation of the form factor always converges to the exact value with the increasing number of samples. As we mentioned in Section 5.2.1, the increasing number of samples does not necessarily improve the precision of the form factor direct area estimator. This can clearly be observed in the F_{perp} graphs in Fig. 5.6.

The number of triangles only influences the computational time, not the quality of the radiosity solution if the weighted analytical estimator is used. (We recall that the number of samples in Fig. 5.6 is given per triangle, not per wall. This means that the number of samples per wall for the scene `BOX48` is four times higher than the number of samples per wall for the scene `BOX12`.) This is not true for the direct area estimator by the computation of F_{perp}. The reason for that is probably that the use of more triangles on the shooting and receiving patches forces the generation of samples pairs which cause an overestimation of the form factor F_{perp}.

5.7.2 Experiments with the box scene

Fig. 5.7 is a box scene which is illuminated by a single point light source which is located in the center of the box (the white spot). This scene demonstrates an effect known as colour bleeding which is missing in (eye-) ray tracing.

Fig. 5.8 is another box scene with a textured right wall. This scene is illuminated by two spot light sources which are both located in the center of the box. One spot light source illuminates the left wall, the other one illuminates the right wall. The floor is a perfect mirror. The texture on the right wall in the two-pass image is distorted because the direct lighting is reconstructed from the illumination stored in the triangle mesh. The resolution of the mesh is not as fine as the resolution of the screen, therefore the texture appear distorted. This model consists of 13000 triangles. The radiosity computation took more than 3 hours (3 samples on the shooting patch and 3 samples on the receiving patch were used for the energy transfer).

The direct illumination can be computed by the ray tracing algorithm more accurately than by the radiosity algorithm. In order not to compute the direct illumination term twice, the illumination from the first shot must be subtracted from the radiosity solution before the ray tracing phase. This technique was used for the computation of the right image in Fig. 5.9. This model consist of ca. 3000 triangles. The radiosity computation took approximately 10 minutes (4 samples

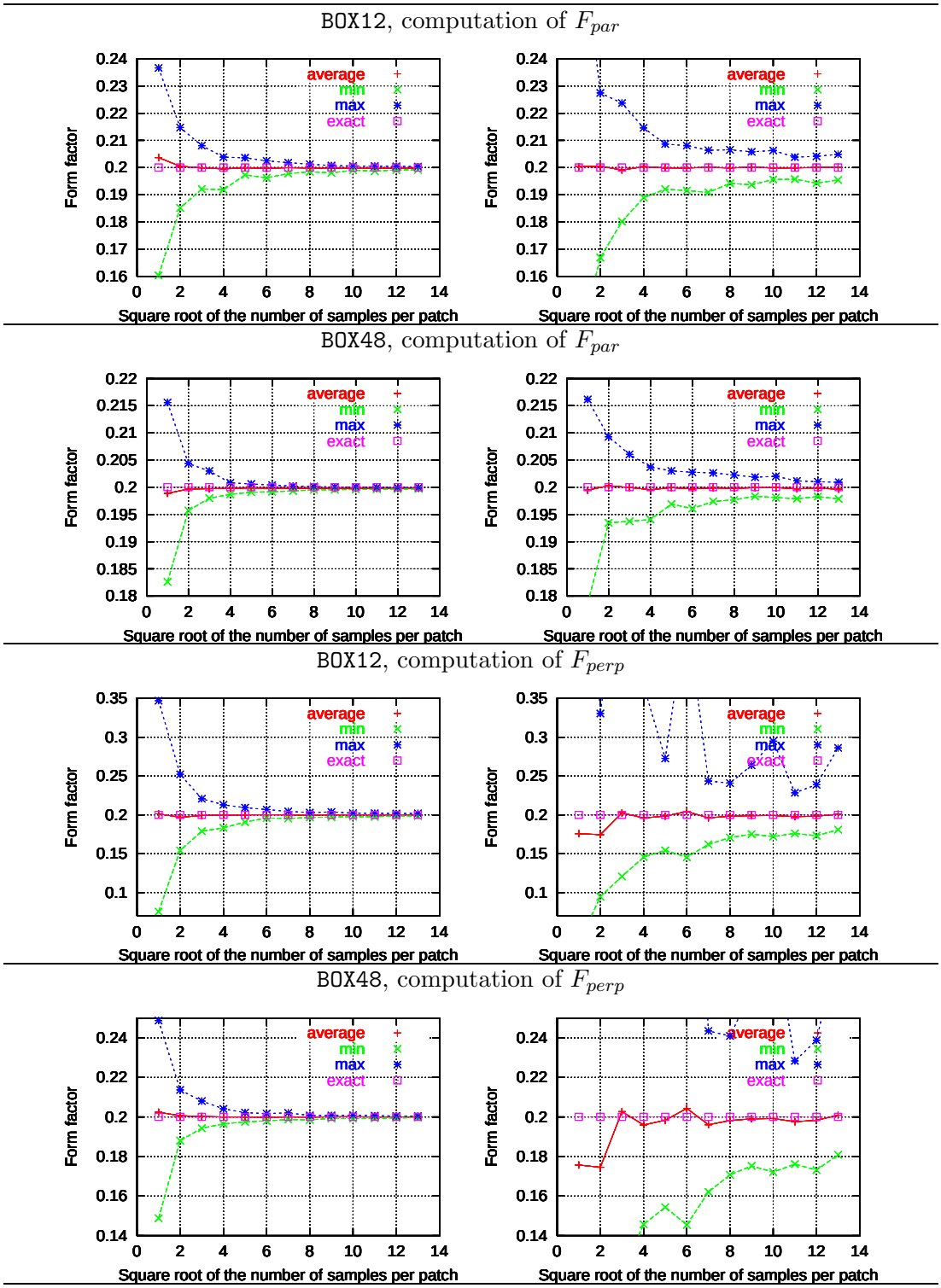

Figure 5.6: Monte-Carlo computation of the form factors F_{par} and F_{perp} for the box scenes BOX12 and BOX48. Left column: direct area estimator (Equation 5.7). Right column: weighted analytical estimator (Equation 5.14)

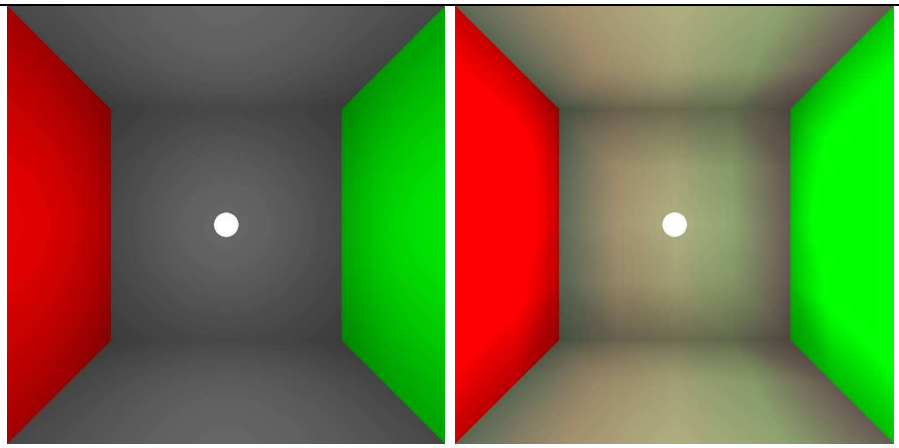

Figure 5.7: Left: ray traced box scene. Right: the radiosity solution (90% converged)

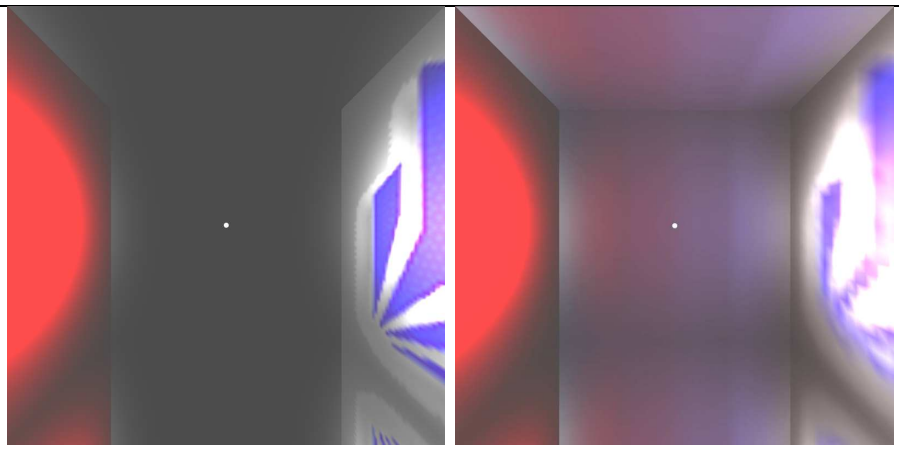

Figure 5.8: Left: ray traced box scene. Right: a two-pass solution (radiosity and ray tracing). The radiosity solution is 90% converged and it is stored in the vertices of ca. 13000 triangles. The texture in the right image is a little distorted because the direct illumination was reconstructed from the radiosity solution.

on the shooting patch and 4 samples on the receiving patch were used for the energy transfer). Note that the texture is not distorted as in Fig. 5.8 although less triangles are used for the illumination storage.

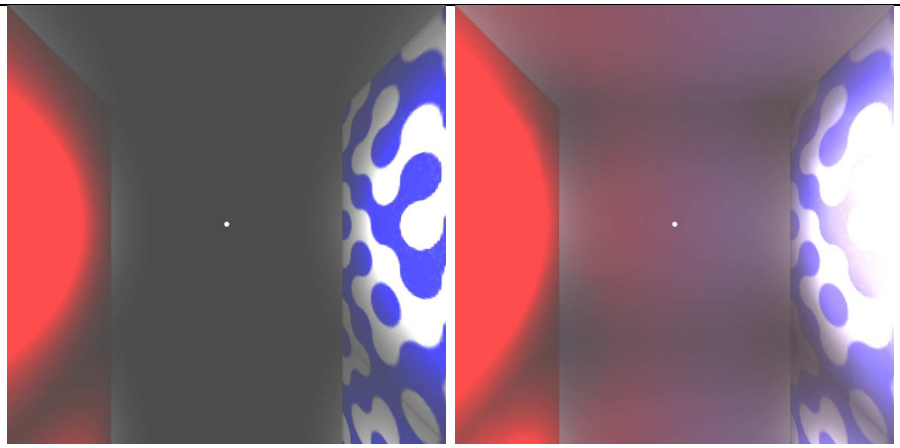

Figure 5.9: Left: ray traced box scene. Right: a two-pass solution (radiosity and ray tracing). The radiosity solution is 95% converged and it is stored in the vertices of ca. 3000 triangles. The direct illumination was subtracted from the radiosity solution and computed by the ray tracing algorithm

Fig. 5.10 shows images of the box scene which contains a blocking object. This scene is illuminated by one spot light source which is directed towards the textured wall. The blocking object is only illuminated indirectly—it is therefore invisible in the ray traced image.

5.7.3 Experiments with large scenes

The main goal of the following experiments was to test the stability of the implementation of radiosity on large scenes rather than to compute converged radiosity solutions (which would require more programming work). The first problem we encountered was the selection of large scenes suitable for the radiosity (two-pass) computation. Our radiosity implementation is based on POV-Ray. However, the implementation requires the objects to be modeled as triangle meshes. Most of the POV-Ray scenes do not use meshes—they use CSG objects instead. We chose two scenes in order to test the stability of the implementation, `HOUSE` and `JAGDSCHLOSS`. Both these scenes were created during the project HiQoS. [ACH+99], [PSA01], [ABB+01] The `HOUSE` scene was modeled in Arcon by M3B and converted into POV-Ray using a special-purpose converter which was developed during the project HiQoS. The original `HOUSE` scene contains ca. 86000 triangles and 27 point light sources. The `JAGDSCHLOSS` scene was modeled in 3DS Max by Kinetix. The model was created by Upstart! and converted into POV-Ray using the 3DWin converter by TB Software. The original scene contains ca.

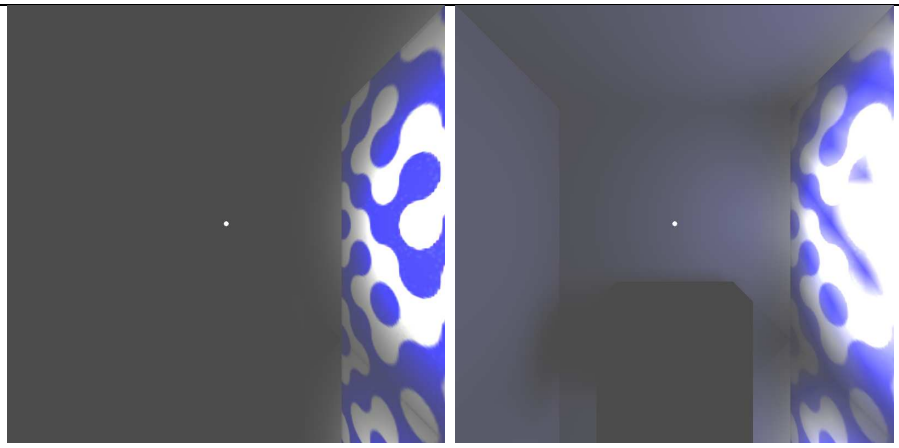

Figure 5.10: Left: ray traced box scene with a blocker. Right: a two-pass solution (radiosity and ray tracing). The radiosity solution is 95% converged and it is stored in the vertices of ca. 3000 triangles. The direct illumination was subtracted from the radiosity solution and computed by the ray tracing algorithm. Note the soft shadow, which is typical for secondary diffuse reflections

190000 triangles and 25 point light sources.

We adjusted the mesh resolution[8] and observed the absolute times during the first few runs. Another problem which we had to solve was that the light sources in both the models also illuminated the outside of the buildings, while the camera viewed the inside of the buildings. This enormously increases the computational time. Even though the outside patches do not usually illuminate any patches of interest, they will be selected for the shooting in the early iterations. In order to save this work, we manually deleted the light sources which were positioned outside of the rooms of interest.

Jagdschloss

The adjusted `JAGDSCHLOSS` scene contains ca. 350000 triangles and 25 light sources. It took ca. 11 hours to compute the first 200 shooting iterations for the scene `JAGDSCHLOSS` (the ray tracing phase took additional 25 minutes). 3 samples on the shooting patch and 3 samples on the receiving patch were used for the energy transfer. The radiosity solution which is shown in Fig. 5.11 is only ca. 3% converged. An additional modeling work is needed to make the materials and the lighting more realistic.

[8]This initial mesh refinement is not necessary when the adaptive substructuring is implemented.

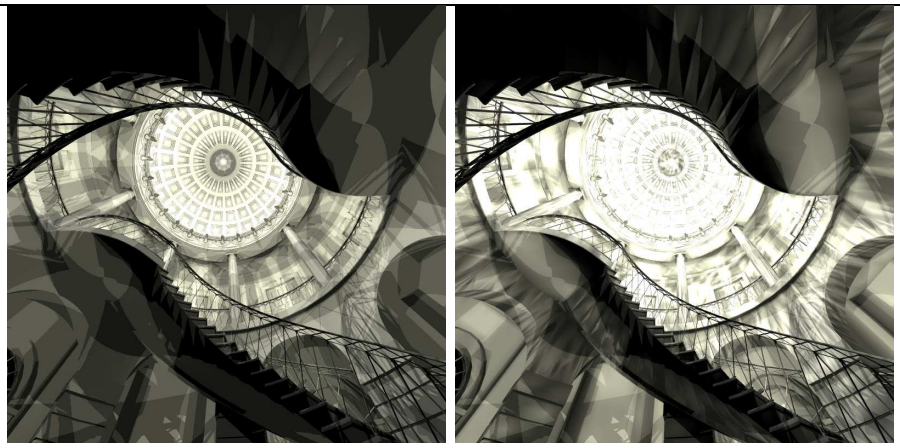

Figure 5.11: Left: ray traced scene JAGDSCHLOSS. Right: the radiosity (two-pass) solution. The radiosity solution is 3% converged and it is stored in the vertices of ca. 350000 triangles.

House

The adjusted HOUSE scene contains ca. 170000 triangles and 6 point light sources. It took ca. 35 hours to compute the first 100 shooting iterations for the scene HOUSE. 3 samples on the shooting patch and 3 samples on the receiving patch were used for the energy transfer. The radiosity solution which is shown in Fig. 5.12 is only ca. 4% converged.

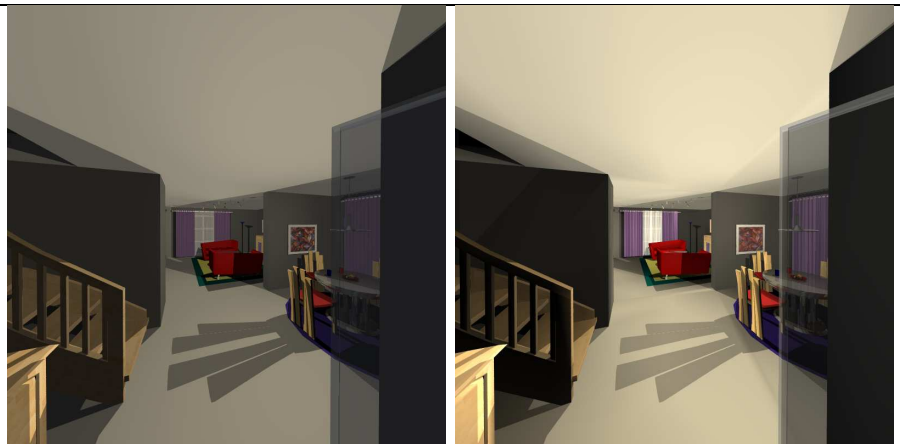

Figure 5.12: Left: ray traced scene HOUSE. Right: the radiosity (two-pass) solution. The radiosity solution is 4% converged and it is stored in the vertices of ca. 170000 triangles.

5.8 Conclusions

We presented a shooting radiosity algorithm which combines the Monte-Carlo form factor computation with the energy exchange using the ray tracing shader. The main advantages of the algorithm are its simplicity and flexibility. No constraints are placed on the topology or the resolution of the input mesh, arbitrary materials are correctly dealt with. The algorithm can be incorporated into any ray tracer as a preprocessing step and ray tracing acceleration techniques such as light buffers and hierarchy of bounding boxes can be reused in the radiosity computations. Our implementation is based on the state of the art Persistence of Vision Ray Tracer (POV-Ray).

We experimentally verified the correctness of the implementation on a specially constructed scene for which the exact radiosity solution is known. We also compared two Monte-Carlo estimators during these experiments: the direct area estimator and the weighted analytical estimator. The weighted analytical estimator clearly outperformed the direct area estimator and was used in all the following experiments.

Another set of experiments included radiosity computations for variants of the box scene and two larger scenes. The implementation needs further optimisations in order to be suitable for large scenes—however, it is reliable. We did not observe any problem although some computations took more than one day.

An inherent disadvantage of the proposed algorithm is that it requires a mesh representation of the model. The efficiency depends on the resolution of the input mesh. An interesting possibility may be a decoupling of the geometry and the illumination storage. For example, the energy exchange may only be carried out between object bounding boxes or between the nodes of the hierarchy of bounding boxes (bounding slabs). An obvious disadvantage of this approach is that the computed illumination is only a rough approximation of the actual illumination. On the other hand, this approach allows for arbitrary object types, not only triangles meshes. Moreover, some of the tricks which are used in order to reduce the time complexity of the shooting radiosity algorithm lead to approximations of the actual illumination anyway. The use of bounding boxes for the energy exchange may be a viable alternative.

Chapter 6

Summary

Parallel photorealistic image synthesis is a challenging problem. Parallel rendering systems which compute photorealistic images are very rare. This thesis identifies some of the obstacles which hamper the development of such systems.

The target architecture for the deployment of parallel rendering systems are distributed-memory systems such as computing clusters. The contemporary middleware for the development of parallel applications for these systems are message passing standards such as PVM and MPI. An obvious problem of some of the implementations of PVM and MPI is a lack of thread-safety. This forces a large class of parallel applications to use polling. This class can be characterised and includes for instance all applications which compute on distributed data. We refer to applications which form this class as non-trivial. Parallel ray tracing using a distributed object database is a non-trivial application.

Polling generally diminishes performance and causes non-deterministic effects in applications. We identified several sources of polling in message passing programs which build on PVM or MPI. Apart from issues such as the lack of thread-safety or the violation of an independent message progress in the implementation of the standards, the specification of the MPI standard differs from the specification of well-accepted abstract message passing models. This difference does not mean an improvement of the abstract models, it is another source of problems in parallel programs which require asynchronous message passing.

We proposed a formal framework which adheres to existing formal message passing models and which addresses practical issues at the same time. There is a strong similarity between our framework and the framework for database systems which is accepted as a standard by both the developers and the users of database systems. We developed a message passing library, TPL, which is a direct implementation of our framework. TPL is thread-safe, does not internally use polling and—unlike PVM or MPI—it defines asynchronous message passing. We showed that TPL can be implemented on the top of slightly extended PVM or MPI implementations. These extensions include an introduction of an interrupt mechanism which is invoked on demand. The event-driven TPL library

outperformed both PVM 3.4 and MPICH 1.2.4 by two orders of magnitude on a simple threaded pingpong benchmark running on a standard computing cluster hardware. This benchmark is an abstraction of a non-trivial application and it must use polling when the communication library is not thread-safe. We do not claim that TPL is the best possible implementation of message passing—on the contrary, there is enough space for improvements in its current implementation. Interestingly, some of the optimisations were addressed in the hardware of INMOS Transputers.

We defined the global illumination problem and gave a brief overview of methods which solve the problem. Direct solution methods are recently getting attention of the computer graphics community but approximation methods such as ray tracing and radiosity are still widely used in applications. We focused on ray tracing because the techniques of the basic ray tracing algorithm can be used in direct methods which solve the global illumination problem without approximations.

We described a parallelisation of ray tracing which builds on the ideas of Green and Paddon who developed a parallel ray tracer on Transputers more than 10 years ago. The parallelisation is relatively simple and robust and does not place constraints on the size of the 3D model, as the model can be distributed in the memories of the processes. We presented a load balancing algorithm for parallel ray tracing which uses demand-driven screen space subdivision. Our algorithm is perfect in the sense that if its parameters are optimally set, then it guarantees a perfect balance of load (hence, the shortest parallel time) and it minimises the communication at the same time. The optimal setting of the parameters is unknown but the parameters are intuitive and can be automatically tuned in the run-time. We suggested a tuning procedure and demonstrated it on a set of experiments. In these experiments, we compared two parallel ray tracing programs which are identical in all respects but one: the first program uses the interrupt mechanism inside the communication library, whereas the second program uses polling inside the communication library. The first program outperformed the second one in all experiments (for all problem instances and in all runs).

We introduced a practical shooting radiosity algorithm which can be incorporated into any ray tracer. The algorithm uses Monte Carlo form factor computation which is combined with energy transfer using the ray tracing shader. This combined step keeps the radiosity algorithm simple and general at the same time. We devoted a special attention to the choice of the Monte Carlo form factor estimator which is essential for the reliability of the radiosity algorithm.

The state of the art of global illumination algorithms is in our opinion much more advanced than the state of the art of 3D standards which define the form in which 3D models are stored. Rendering algorithms, as well as human 3D artists, are often either forced to work with insufficient or incorrect data or to develop their own standards. The following short section identifies a source of this problem.

6.1 Towards portable 3D standards

Computer graphics has been evolving very fast in the past few decades. It has found applications in computer games, films, architecture, etc. However, the process of producing photorealistic images is far from being automatic. The human factor is required in order to achieve the desired level of perfection. The 3D artist must sometimes "help" the rendering system to make the image look realistic. This is not acceptable in certain applications. An example of such an application is the conservation of cultural heritage. We would like to store models of real 3D objects such as buildings, cars, furnitures, vases, statues etc. in a form from which very realistic images can be reconstructed later. This form must not be tied to any particular modeling or rendering software product or to a particular rendering algorithm. Future improvements of the rendering systems should increase the level of photorealism. Several hundreds of 3D formats exist but none provides this level of portability. We will return to the modeling of surface geometry in order to explain what is missing. However, the following reasoning also applies to the modeling of materials, light sources etc. (The introduction of procedural shaders, IES luminaire format and MGF format indicates a movement toward the standardisation of materials and light sources.)

Almost all modeling programs can internally work with spheres. Why cannot a sphere be passed to some other modeling or rendering program as a sphere? Why must an object as simple as a sphere be converted into a triangle mesh instead? The first reason is the "almost all"—not all modeling or rendering programs work with spheres. Another reason are differences in representations of spheres. One program may work with the representation $\langle center, radius \rangle$ whereas another program may work the representation $\langle x_1, x_2, x_3, x_4 \rangle$ (where x_1, x_2, x_3, x_4 are points in 3D space). The conversion between these two representations is simple in this particular case, but it may be difficult or impossible for representations of other geometric primitives.

Let us assume that a standard 3D format exists which can store spheres (in some representation). Which other geometric primitives must the format support? Cones? Tori? Cylinders? The current state of the art reduces the set of geometric primitives to a triangle mesh because triangle meshes are currently supported in practically all software and hardware systems. We claim that a portable 3D standard must support *all* geometric primitives. However, the set of all geometric primitives is infinite and there is no unique representation for all of them.

A portable 3D standard should not attempt to define the set of representations which must be supported by modeling and rendering programs—instead of that it must define *methods* (operations on the geometric primitives) on which the programs can rely. These methods constitute the interface between the representations of geometric primitives and algorithms which work with the primitives. Good candidates for such methods are finding all intersections of a ray and a

primitive and computation of the surface normal at an intersection point. The set of methods must be chosen carefully so that they can also be applied on compound objects which are created using the CSG operations (see Section 3.2.2). *The implementation of the methods for a geometric primitive (a program code) must be stored in a file together with the primitive.* The storage of program code in a platform-independent form was a problem in the past but this has changed. At the time being, Java code is portable across practically all existing platforms. [Sunb] The methods can therefore be implemented in Java.

The hiding of the implementation of the methods allows for an insertion of a new geometric primitive at any time without the need of reimplementation of existing modeling or rendering software. This is actually the idea behind the Java technology. Java3D is in our opinion a step backwards (it is based on the mesh-only technology) and we believe that the introduction of Java3D by the original Java developers (Sun Microsystems) is rather a tactical than a strategical decision. [Suna]

Triangle meshes have been around for several decades and will certainly remain being around for a long time. However, the concept which we sketch does not exclude using triangle meshes further—it only allows for using also other geometric primitives. Rendering algorithms are prepared for the introduction of such a concept and so are 3D artists. A great part of contemporary computer graphics is a collection of one-purpose tricks. A unification concept would help to distinguish one-purpose tricks from techniques which apply generally.

Appendix A

MPI progress rule tester

The MPI program below verifies whether the implementation of the MPI library violates the progress rule defined in the MPI standard or not. The `sleep(5)` in the process 0 should ensure that the process 1 performs its actions prior to the `MPI_Ssend()` call in the process 0. This program, linked with an MPI library which obeys the progress rule, eventually (usually immediately after 5 seconds have passed) prints out the following:

```
0: Sleeping for 5 seconds...
1: Posting Irecv
1: Irecv posted, blocking
0: Ssend...
0: Ssend completed (test passed)
```

An incorrect MPI implementation (an implementation which violates the progress rule) does not print out the last line. None of the following MPI implementations passed this test:

- MPICH 1.2.4 (Intel Pentium, Fast-Ethernet/TCP)
- MPICH 1.2.4 (Intel Itanium, Myrinet/GM)
- ScaMPI 1.13.7 (Intel Pentium, Dolphin PCI/SCI)

(We are aware of no MPI implementation which passes this test.)

```
/* mpi_progress.c */

#include <stdio.h>
#include <stdlib.h>

#include "mpi.h"
```

```
int main(int argc, char *argv[])
{
  int rank;
  MPI_Request req;
  int tmp_int;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0)
  {
    printf("0: Sleeping for 5 seconds...\n"); fflush(stdout);
    sleep(5);
    printf("0: Ssend...\n"); fflush(stdout);
    MPI_Ssend(&tmp_int, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0: Ssend completed (test passed)\n"); fflush(stdout);
  }
  else
  {
    printf("1: posting Irecv\n"); fflush(stdout);
    MPI_Irecv(&tmp_int, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &req);
    printf("1: Irecv posted, blocking\n"); fflush(stdout);
    for (;;)
      ;
  }
  MPI_Finalize();
  return(0);
}
```

Appendix B

Threaded pingpong benchmark

The three programs below implement the *SYMMETRICAL THREADED PING-PONG* benchmark which is described in Section 2.8.2. The TPL 2.0 program is event-driven, whereas the PVM 3.4 and MPI programs use polling.

B.1 TPL 2.0

```
/* pingpong_tpl.c */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

#include "tpl.h"

#define RANK_PING 0
#define RANK_PONG 1

enum
{
  MSG_INIT = TPL_MSGTAG_LAST,
  MSG_PING,
  MSG_PONG,
  MSG_QUIT
};

static int nr_msgs;
static int msg_size;

static char *databuf;
```

```
static int rank_ping = RANK_PING;
static int rank_pong = RANK_PONG;
static int my_rank;

static struct timeval tv1, tv2;
static struct timezone tz1, tz2;
static float elapsed_time, mbytes;

static int match_pong(int sender, int tag, void *message);
static void *send_thread(void *arg);
static void *recv_thread(void *arg);
static TPL_ACTION message_handler(int sender, int tag, void *message);

static int match_pong(int sender, int tag, void *message)
{
  return(tag == MSG_PONG);
}

static void *send_thread(void *arg)
{
  int counter;
  void *sendbuf;

  /* To avoid startup bias. */
  sleep(3);

  /* Start timer. */
  gettimeofday(&tv1, &tz1);

  for (counter = 0; counter < nr_msgs; counter++)
  {
    tpl_begin_send(&sendbuf);
    tpl_pkchar(sendbuf, databuf, msg_size);
    tpl_send(&rank_pong, 1, MSG_PING, sendbuf);
    tpl_end_send(sendbuf);
  }

  return NULL;
}

static void *recv_thread(void *arg)
{
  int counter;
  void *message;
```

```
  int tag;
  int sender;

  char *buf;

  if (msg_size > 0)
    buf = (char *) tpl_malloc(msg_size * sizeof(char));

  for (counter = 0; counter < nr_msgs; counter++)
  {
    tpl_begin_recv();
    tpl_recv(match_pong, &sender, &tag, &message);
    tpl_upkchar(message, buf, msg_size);
    tpl_end_recv(message);
  }

  /* Stop timer. */
  gettimeofday(&tv2, &tz2);

  tpl_begin_send(&message);
  tpl_send(&rank_pong, 1, MSG_QUIT, message);
  tpl_end_send(message);
  tpl_begin_send(&message);
  tpl_send(&rank_ping, 1, MSG_QUIT, message);
  tpl_end_send(message);

  elapsed_time = tv1.tv_usec < tv2.tv_usec ?
   tv2.tv_sec - tv1.tv_sec + (tv2.tv_usec - tv1.tv_usec) / 1e6 :
   tv2.tv_sec - tv1.tv_sec - 1 + (tv1.tv_usec - tv2.tv_usec) / 1e6;
  mbytes = 2.0 * (float) nr_msgs * (float) msg_size / 1048576.0;
  printf("%d x %d Bytes x 2 = %f MBytes, %f s, %f MB/s, %f msg/s,
   %f s/msg, %d / %d sleeps\n",
   nr_msgs, msg_size, mbytes, elapsed_time, mbytes / elapsed_time,
   nr_msgs / elapsed_time, elapsed_time / nr_msgs, 0, 0);

  return NULL;
}

static TPL_ACTION message_handler(int sender, int tag, void *message)
{
  void *sendbuf = NULL;

  switch (tag)
  {
    case MSG_PING: /* Only for PONG */
```

```
        tpl_begin_send(&sendbuf);
        tpl_pkchar(sendbuf, databuf, msg_size);
        tpl_send(&rank_ping, 1, MSG_PONG, sendbuf);
        tpl_end_send(sendbuf);
        return(TPL_ACTION_DROP);
        break;

      case MSG_PONG: /* Only for PING */
        return(TPL_ACTION_ENQUEUE);
        break;

      case MSG_QUIT: /* For both PING and PONG */
        return(TPL_ACTION_EXIT);
        break;

      default:
        printf("%d:  ", my_rank);
        tpl_error("Unknown message\n");
        break;
    }

    return(FALSE);
}

int main(int argc, char **argv)
{
    pthread_t send_thread_id, recv_thread_id;

    int nr_tasks;

    tpl_initialize(&argc, &argv);

    /* Spawn processes. */

    nr_tasks = 2;
    tpl_spawn(&nr_tasks, &my_rank, &argc, &argv);

    if (argc != 3)
    {
      tpl_error("Usage:  ping <nr_msgs> <msg_size>\n");
    }

    nr_msgs = atoi(argv[1]);
    msg_size = atoi(argv[2]);
```

```
  if (msg_size > 0)
    databuf = (char *) tpl_malloc(msg_size * sizeof(char));

  switch(my_rank)
  {
    case RANK_PING:
      /* Start sending thread. */
      if (pthread_create(&send_thread_id, NULL, send_thread,
       NULL) != 0)
      {
        tpl_error("Could not start sending thread\n");
      }

      /* Start receiving thread. */
      if (pthread_create(&recv_thread_id, NULL, recv_thread,
       NULL) != 0)
      {
      tpl_error("Could not start sending thread\n");
      }

      tpl_handle_messages(message_handler);

      pthread_join(send_thread_id, NULL);
      pthread_join(recv_thread_id, NULL);

      break;

    case RANK_PONG:
      tpl_handle_messages(message_handler);
      break;

    default:
      tpl_error("Unknown role\n");
      break;
  }

  /* Terminate the program. */
  tpl_deinitialize();

  if (msg_size > 0)
    tpl_free(databuf);

  return(0);
}
```

B.2 PVM 3.4

```
/* ping_pvm.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#include "pvm3.h"

#define ERROR(a) {printf(a); fflush(stdout); exit(1);}
#define PONG_LOCATION "pvm_pong";
#define MAX_MSG_LENGTH 2000000

enum
{
  MSG_INIT = 0,
  MSG_PING,
  MSG_PONG,
  MSG_QUIT
};

static int pong_tid;
static int nr_msgs;
static int msg_size;

static struct timeval tv1, tv2;
static struct timezone tz1, tz2;
static float elapsed_time, mbytes;

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

static void *pong_send(void *arg);

static void *pong_send(arg)
void *arg;
{
  int counter;
  char msg[MAX_MSG_LENGTH];

  /* To avoid startup bias. */
  sleep(3);
```

```
  /* Start timer. */
  gettimeofday(&tv1, &tz1);

  for (counter = 0; counter < nr_msgs; counter++)
  {
    pthread_mutex_lock(&mutex);
    if (pvm_initsend(PvmDataDefault) < 0)
      ERROR("error pvm_initsend\n");
    pvm_pkbyte(msg, msg_size, 1);
    if (pvm_send(pong_tid, MSG_PING) < 0)
      ERROR("error pvm_send\n");
    pthread_mutex_unlock(&mutex);
  }
  return(NULL);
}

int main(argc, argv)
int argc;
char *argv[];
{
  int counter;
  char msg[MAX_MSG_LENGTH];
  pthread_t pth_id;
  int bufid;
  struct timespec pause, rem;
  struct pvmhostinfo *pvm_hosts;
  int pvm_nr_hosts;
  int pvm_nr_archs;

  int mytid = pvm_mytid();
  int nr_sleep_calls = 0;
  int nr_sleep_calls2 = 0;

  if (argc != 3)
  {
    printf("Usage:  ping <nr_msgs> <msg_size>\n");
    exit(1);
  }

  nr_msgs = atoi(argv[1]);
  msg_size = atoi(argv[2]);

  pvm_setopt(PvmRoute, PvmRouteDirect);
  pvm_config(&pvm_nr_hosts, &pvm_nr_archs, &pvm_hosts);
```

```
if (pvm_spawn(PONG_LOCATION, NULL, PvmTaskHost,
 pvm_hosts[1].hi_name, 1, &pong_tid) < 1)
  ERROR("Could not spawn pong\n");

if (pvm_initsend(PvmDataDefault) < 0)
  ERROR("error pvm_initsend\n");
pvm_pkint(&mytid, 1, 1);
pvm_pkint(&nr_msgs, 1, 1);
pvm_pkint(&msg_size, 1, 1);
if (pvm_send(pong_tid, MSG_INIT) < 0)
  ERROR("error pvm_send\n");
pvm_recv(-1, MSG_INIT);

pthread_create(&pth_id, NULL, pong_send, NULL);

for (counter = 0; counter < nr_msgs;)
{
  pthread_mutex_lock(&mutex);
  while ((bufid = pvm_probe(-1, MSG_PONG)) > 0)
  {
    pvm_recv(-1, MSG_PONG);
    pvm_upkbyte(msg, msg_size, 1);
    pvm_freebuf(bufid);
    counter++;
  }
  pthread_mutex_unlock(&mutex);
  if (counter)
    nr_sleep_calls++;
  if (counter < nr_msgs)
  {
    pause.tv_sec = 0;
    pause.tv_nsec = 50000000;
    while (nanosleep(&pause, &rem) == -1)
      nanosleep(&rem, &rem);
  }
}

pvm_recv(-1, MSG_QUIT);
pvm_upkint(&nr_sleep_calls2, 1, 1);

/* Stop timer. */
gettimeofday(&tv2, &tz2);

pthread_join(pth_id, NULL);
```

```
  elapsed_time = tv1.tv_usec < tv2.tv_usec ?
   tv2.tv_sec - tv1.tv_sec + (tv2.tv_usec - tv1.tv_usec) / 1e6 :
   tv2.tv_sec - tv1.tv_sec - 1 + (tv1.tv_usec - tv2.tv_usec) / 1e6;
  mbytes = 2.0 * (float) nr_msgs * (float) msg_size / 1048576.0;
  printf("%d x %d Bytes x 2 = %f MBytes, %f s, %f MB/s, %f msg/s,
   %f s/msg, %d / %d sleeps\n",
   nr_msgs, msg_size, mbytes, elapsed_time, mbytes / elapsed_time,
   nr_msgs / elapsed_time, elapsed_time / nr_msgs,
   nr_sleep_calls, nr_sleep_calls2);

  pvm_exit();

  return(0);
}
```

```
/* pong_pvm.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#include "pvm3.h"

#define MAX_MSG_LENGTH 2000000

enum
{
  MSG_INIT = 0,
  MSG_PING,
  MSG_PONG,
  MSG_QUIT
};

static int ping_tid;
static int nr_msgs;
static int msg_size;

static char sendbuf[MAX_MSG_LENGTH];

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main()
```

```
{
  int counter;
  int bufid;
  struct timespec pause, rem;
  int nr_sleep_calls = 0;

  pvm_mytid();
  pvm_setopt(PvmRoute, PvmRouteDirect);
  bufid = pvm_recv(-1, MSG_INIT);

  pvm_upkint(&ping_tid, 1, 1);
  pvm_upkint(&nr_msgs, 1, 1);
  pvm_upkint(&msg_size, 1, 1);

  pvm_initsend(PvmDataDefault);
  pvm_send(ping_tid, MSG_INIT);

  for (counter = 0; counter < nr_msgs;)
  {
    pthread_mutex_lock(&mutex);
    while ((bufid = pvm_probe(-1, MSG_PING)) > 0)
    {
      pvm_recv(-1, MSG_PING);
      pvm_upkbyte(sendbuf, msg_size, 1);
      pvm_freebuf(bufid);
      pvm_initsend(PvmDataDefault);
      pvm_pkbyte(sendbuf, msg_size, 1);
      pvm_send(ping_tid, MSG_PONG);

      counter++;
    }
    pthread_mutex_unlock(&mutex);
    if (counter)
      nr_sleep_calls++;
    if (counter < nr_msgs)
    {
      pause.tv_sec = 0;
      pause.tv_nsec = 50000000;
      while (nanosleep(&pause, &rem) == -1)
        nanosleep(&rem, &rem);
    }
  }
  pvm_initsend(PvmDataDefault);
  pvm_pkint(&nr_sleep_calls, 1, 1);
  pvm_send(ping_tid, MSG_QUIT);
```

```
  pvm_exit();

  return(0);
}
```

B.3 MPI (MPI 1, MPI 2)

```
/* pingpong_mpi.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#include "mpi.h"

#define MAX_MSG_LENGTH 2000000
#define ERROR(a) {printf(a); fflush(stdout); exit(1);}

enum
{
  MSG_INIT = 0,
  MSG_PING,
  MSG_PONG,
  MSG_QUIT
};

static int ping_tid = 0;
static int pong_tid = 1;
static int nr_msgs;
static int msg_size;

static struct timeval tv1, tv2;
static struct timezone tz1, tz2;
static float elapsed_time, mbytes;

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

static void *pong_send(void *arg);

static void *pong_send(arg)
```

```
void *arg;
{
   int counter;
   char msg[MAX_MSG_LENGTH];
   char msg2[MAX_MSG_LENGTH];
   int offset;
   MPI_Request send_req;
   MPI_Status sts;
   int send_finished;
   struct timespec pause, rem;

   /* To avoid startup bias. */
   sleep(3);

   /* Start timer. */
   gettimeofday(&tv1, &tz1);

   pthread_mutex_lock(&mutex);
   for (counter = 0; counter < nr_msgs; counter++)
   {
      offset = 0;
      MPI_Pack(msg2, msg_size, MPI_BYTE, msg, MAX_MSG_LENGTH, &offset,
       MPI_COMM_WORLD);
      /* We must not use synchronous send here, otherwise the data flow
      mechanism will block if this thread gets to sending
      many times in a row. */
      MPI_Isend(msg, offset, MPI_PACKED, pong_tid, MSG_PING,
       MPI_COMM_WORLD, &send_req);
      /* We must make sure that the send finishes before we attempt to
      start another one.  But at the same time we must allow the recv
      thread to proceed. */
      do
      {
         send_finished = 0;
         MPI_Test(&send_req, &send_finished, &sts);
         if (! send_finished)
         {
            pthread_mutex_unlock(&mutex);
            pause.tv_sec = 0;
            pause.tv_nsec = 50000000;
            while (nanosleep(&pause, &rem) == -1)
               nanosleep(&rem, &rem);
            pthread_mutex_lock(&mutex);
         }
      } while (! send_finished);
```

```
    }
    pthread_mutex_unlock(&mutex);
    return(NULL);
}

int main(argc, argv)
int argc;
char *argv[];
{
    int counter;
    char msg[MAX_MSG_LENGTH];
    char msg2[MAX_MSG_LENGTH];
    pthread_t pth_id;
    int offset;
    struct timespec pause, rem;
    int rank;
    MPI_Status sts;
    int flag;
    int nr_sleep_calls = 0;
    int nr_sleep_calls2 = 0;

    MPI_Init(&argc, &argv);

    if (argc != 3)
    {
        printf("Usage:  ping <nr_msgs> <msg_size>\n");
        exit(1);
    }

    nr_msgs = atoi(argv[1]);
    msg_size = atoi(argv[2]);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == ping_tid)
    {
        offset = 0;
        MPI_Pack(&pong_tid, 1, MPI_INT, msg, MAX_MSG_LENGTH, &offset,
         MPI_COMM_WORLD);
        MPI_Pack(&nr_msgs, 1, MPI_INT, msg, MAX_MSG_LENGTH, &offset,
         MPI_COMM_WORLD);
        MPI_Pack(&msg_size, 1, MPI_INT, msg, MAX_MSG_LENGTH, &offset,
         MPI_COMM_WORLD);
        MPI_Send(msg, offset, MPI_PACKED, pong_tid, MSG_INIT,
         MPI_COMM_WORLD);
        MPI_Recv((void *) msg, MAX_MSG_LENGTH, MPI_PACKED, pong_tid,
```

```
 MSG_INIT, MPI_COMM_WORLD, &sts);

pthread_create(&pth_id, NULL, pong_send, NULL);

pthread_mutex_lock(&mutex);
for (counter = 0; counter < nr_msgs;)
{
  do
  {
    flag = 0;
    MPI_Iprobe(pong_tid, MSG_PONG, MPI_COMM_WORLD, &flag, &sts);
    if (flag)
    {
      MPI_Recv((void *) msg, MAX_MSG_LENGTH, MPI_PACKED,
       pong_tid, MSG_PONG, MPI_COMM_WORLD, &sts);
      offset = 0;
      MPI_Unpack((void *) msg, MAX_MSG_LENGTH, &offset,
       msg2, msg_size, MPI_BYTE, MPI_COMM_WORLD);
      counter++;
    }
  }
  while (flag);

  if (counter)
    nr_sleep_calls++;
  if (counter < nr_msgs)
  {
    pthread_mutex_unlock(&mutex);
    pause.tv_sec = 0;
    pause.tv_nsec = 50000000;
    while (nanosleep(&pause, &rem) == -1)
      nanosleep(&rem, &rem);
    pthread_mutex_lock(&mutex);
  }
}
pthread_mutex_unlock(&mutex);

/* Stop timer. */
gettimeofday(&tv2, &tz2);

MPI_Recv((void *) msg, MAX_MSG_LENGTH, MPI_PACKED,
 pong_tid, MSG_QUIT, MPI_COMM_WORLD, &sts);

offset = 0;
MPI_Unpack((void *) msg, sizeof(int), &offset,
```

```
     &nr_sleep_calls2, 1, MPI_INT, MPI_COMM_WORLD);

   pthread_join(pth_id, NULL);

   elapsed_time = tv1.tv_usec < tv2.tv_usec ?
    tv2.tv_sec - tv1.tv_sec + (tv2.tv_usec - tv1.tv_usec) / 1e6 :
    tv2.tv_sec - tv1.tv_sec - 1 + (tv1.tv_usec - tv2.tv_usec) / 1e6;
   mbytes = 2.0 * (float) nr_msgs * (float) msg_size / 1048576.0;
   printf("%d x %d Bytes x 2 = %f MBytes, %f s, %f MB/s, %f msg/s,
    %f s/msg, %d / %d sleeps\n",
    nr_msgs, msg_size, mbytes, elapsed_time, mbytes / elapsed_time,
    nr_msgs / elapsed_time, elapsed_time / nr_msgs,
    nr_sleep_calls, nr_sleep_calls2);
}
else
{
   /* Pong process. */
   MPI_Recv((void *) msg, MAX_MSG_LENGTH, MPI_PACKED, ping_tid,
    MSG_INIT, MPI_COMM_WORLD, &sts);
   offset = 0;
   MPI_Unpack((void *) msg, MAX_MSG_LENGTH, &offset, &pong_tid, 1,
    MPI_INT, MPI_COMM_WORLD);
   MPI_Unpack((void *) msg, MAX_MSG_LENGTH, &offset, &nr_msgs, 1,
    MPI_INT, MPI_COMM_WORLD);
   MPI_Unpack((void *) msg, MAX_MSG_LENGTH, &offset, &msg_size, 1,
    MPI_INT, MPI_COMM_WORLD);
   MPI_Send(msg, offset, MPI_PACKED, ping_tid, MSG_INIT,
    MPI_COMM_WORLD);

   pthread_mutex_lock(&mutex);
   for (counter = 0; counter < nr_msgs;)
   {
      do
      {
         MPI_Iprobe(ping_tid, MSG_PING, MPI_COMM_WORLD, &flag, &sts);
         if (flag)
         {
            MPI_Recv((void *) msg, MAX_MSG_LENGTH, MPI_PACKED,
             ping_tid, MSG_PING, MPI_COMM_WORLD, &sts);
            MPI_Unpack((void *) msg, MAX_MSG_LENGTH, &offset,
             msg2, msg_size, MPI_BYTE, MPI_COMM_WORLD);
            counter++;
            offset = 0;
            MPI_Pack(msg2, msg_size, MPI_BYTE, msg, MAX_MSG_LENGTH,
             &offset, MPI_COMM_WORLD);
```

```
            MPI_Send(msg, offset, MPI_PACKED, ping_tid, MSG_PONG,
             MPI_COMM_WORLD);
          }
        }
        while (flag);

        if (counter)
          nr_sleep_calls++;
        if (counter < nr_msgs)
        {
          pthread_mutex_unlock(&mutex);
          pause.tv_sec = 0;
          pause.tv_nsec = 50000000;
          while (nanosleep(&pause, &rem) == -1)
          nanosleep(&rem, &rem);
          pthread_mutex_lock(&mutex);
        }
      }
      pthread_mutex_unlock(&mutex);
      offset = 0;
      MPI_Pack(&nr_sleep_calls, 1, MPI_INT, msg, MAX_MSG_LENGTH, &offset,
       MPI_COMM_WORLD);
      MPI_Send(msg, offset, MPI_PACKED, ping_tid, MSG_QUIT,
       MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return(0);
}
```

List of Figures

Bibliography

[ABB+01] P. Altenbernd, A. Bartels, S. Bicskey, L. O. Burchard, M. Holch, J. Jensch, M. Oestreicher, I. Neumann, T. Plachetka, T. Prill, J. Seiler, and A. Schmitt. HiQoS: High Performance Multimedia-Dienste mit Quality-of-Service-Garantien, 2001. Projekt HiQoS, Abschlussbericht.

[ACH+99] P. Altenbernd, F. Cortes, M. Holch, J. Jensch, O. Michel, C. Moar, T. Prill, R. Lüling, K. Morisse, I. Neumann, T. Plachetka, M. Reith, O. Schmidt, A. Schmitt, and A. Wabro. BMBF-Projekt HiQoS: High-Performance-Multimedia-Dienste mit Quality-of-Service-Garantien. In R. Krahl, editor, *Statustagung des BMBF, HPSC '99, Höchstleistungsrechnen in der Bundesrepublik Deutschland*, pages 29–32. BMBF, Bundesministerium für Bildung und Forschung, 1999.

[And91] G. A. Andrews. *Concurrent Programming, Principles and Practice.* Benjamin/Cummings Publishing Company, 1991.

[App68] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of AFIPS 1968 Joint Computer Conference*, volume 32, pages 37–45, 1968.

[Arv95] J. Arvo. Stratified sampling of spherical triangles. *Computer Graphics*, pages 437–438, 1995.

[Ash01] I. Ashdown. Eigenvector radiosity. Diploma thesis, Department of Computer Science, Faculty of Graduate Studies, University of British Columbia, 2001.

[Atk76] K. E. Atkinson. *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind.* Society for Industrial Mathematics (SIAM), 1976.

[Bac98] J. Bacon. *Concurrent Systems (Operating Systems, Database and Distributed Systems: An Integrated Approach).* Addison-Wesley-Longman, 1998.

[BBP94] D. Badouel, K. Bouatouch, and T. Priol. Distributing data and control for ray tracing in parallel. *IEEE Computer Graphics and Applications*, 14(4):69–77, 1994.

[Bek99] P. Bekaert. *Hierarchical and Stochastic Algorithms for Radiosity*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1999.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BL93] A. J. Bernstein and P. M. Lewis. *Concurrency in Programming and Database Systems*. Jones and Bartlett Publishers, 1993.

[Bli77] J. F. Blinn. Models of light reflection for computer synthesized pictures. *Computer Graphics*, pages 192–198, 1977.

[Bli78] J. F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics*, 12:296–292, 1978.

[BMR02] R. Brightwell, A. B. Maccabe, and R. Riesen. Design and implementation of MPI on Portals 3.0. In D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *Proc. of the 9th EuroPVM/MPI User's Group Conference (Recent Advances in Parallel Virtual Machine and Message Passing Interface)*, volume 2474 of *Lecture Notes in Computer Science*, pages 331–340. Springer-Verlag, 2002.

[BMSW91] D. R. Baum, S. Mann, K. P. Smith, and J. M. Winget. Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. *Computer Graphics*, 25(4):51–60, 1991.

[BP88] K. Bouatouch and T. Priol. Parallel space tracing: An experience on an iPSC hypercube. In *Proc. of Computer Graphics International'88 (New Trends in Computer Graphics)*, pages 170–188. Computer Graphics Society, 1988.

[BRW89] D. R. Baum, H. E. Rushmeier, and J. M. Winget. Improving radiosity solutions through the use of analytically determined form factors. *Computer Graphics*, 23:325–334, 1989.

[BW95] P. Bekaert and Y. D. Willems. Importance-driven progressive refinement radiosity. In P. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95, Proceedings of the Eurographics Workshop on Rendering*. Springer, 1995.

[CC02] A. Chalmers and K. Cater. Realistic rendering in real-time. In B. Monien and R. Feldman, editors, *Proceedings of Euro-Par 2002 (Parallel Processing)*, volume 2400, pages 21–28. Springer, 2002.

[CCWG88] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, 22:75–84, 1988.

[CDP95] F. Cazals, G. Drettakis, and C. Puech. Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. *Computer Graphics Forum*, 14(3):371–382, 1995.

[CDR02] A. Chalmers, T. Davis, and E. Reinhard. *Practical Parallel Rendering*. A K Peters, 2002.

[CG85] M. F. Cohen and D. P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics*, 19:31–30, 1985.

[CPC84] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, 1984.

[CSD94] A. K. Chowdappa, A. Skjellum, and N. E. Doss. Thread-safe message passing with P4 and MPI. Technical report, Mississippi State University, Dept. of Computer Science, 1994.

[CT96] A. Chalmers and J. Tidmus. *Practical Parallel Processing*. Interanational Thomson Publishing, 1996.

[CW93] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, 1993.

[CWBV85] J. G. Clearly, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, 5:3–12, 1985.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[Dim01] R. P. Dimitrov. *Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations*. PhD thesis, Mississippi State University, 2001.

[DLW93] P. Dutre, E. P. Lafortune, and Y. D. Willems. Monte Carlo light tracing with direct computation of pixel intensities. In *Proceedings of Compugraphics*, pages 128–137. Alvor, 1993.

[DS84] M. Dippé and J. Swenssen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics*, 18(3), 1984.

[DS02] R. Dimitrov and A. Skjellum. *Software Architecture and Performance Comparison of MPI/Pro and MPICH*. MPI Software Technology, Inc., 2002. (White paper).

[EBe92] M. C. Escher, F. Bool, and J. L. Locher (editor). *M. C. Escher: His Life and Complete Graphic Work*. Abradale Press, 1992.

[Ent93] N. E. Things Enterprises. *Magic Eye I: A New Way of Looking at the World*. Andrews and McMeel, 1993.

[ES00] M. Evans and T. Swartz. *Approximating Integrals via Monte Carlo and Deterministic Methods*. Oxford University Press, 2000.

[Fer98] A. Ferrari. JPVM: Network parallel computing in Java. *Concurrency: Practice and Experience*, 10(11–13):985–992, 1998.

[Fey88] R. Feynman. *QED: The Strange Theory of Light and Matter*. Princeton University, 1988.

[FFB99] A. Fava, E. Fava, and M. Bertozzi. MPIPOV: A parallel implementation of POV-Ray based on MPI. In J. Dongarra, E. Luque, and T. Margalef, editors, *Proc. of the 6th EuroPVM/MPI User's Group Conference (Recent Advances in Parallel Virtual Machine and Message Passing Interface)*, volume 1697 of *Lecture Notes in Computer Science*, pages 426–433. Springer-Verlag, 1999.

[FHBWW95] S. Flynn-Hummel, I. Banicescu, C. T. Wang, and J. Wein. Load balancing and data locality via fractling: An experimental study. In B. K. Szymanski and B. Sinharoy, editors, *Proc. of the 3rd Workshop on Languages, Compilers, and RunTime Systems for Scalable Computers*, pages 85–89. Kluwer Academic Publishers, 1995.

[FHK97] B. Freisleben, D. Hartmann, and T. Kielmann. Parallel raytracing: A case study on partitioning and scheduling on workstation clusters. In *Proceedings of Hawaii International Conference on System Sciences (HICSS-30)*, volume 1, pages 596–605. IEEE Computer Society Press, 1997.

[FHSF91] S. Flynn-Hummel, E. Schonberg, and L. E. Flynn. Factoring: A practical and robust method for scheduling parallel loops. In *Proc. of Supercomputing '91*, pages 610–619. IEEE Computer Society / ACM, 1991.

[FHSF92] S. Flynn-Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM (CACM)*, 35(8):90–101, 1992.

[FHSUW96] S. Flynn-Hummel, J. P. Schmidt, R. N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proc. of the 8th Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pages 318–328. ACM Press, 1996.

[FLS70] R. P. Feynman, R. B. Leighton, and M. Sands. *The Feynman Lectures on Physics*. Addison Wesley Longman, 1970.

[FS98] A. Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. *Concurrency—Practice and Experience*, 10(3):199–228, 1998.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (San Diego, CA)*, pages 114–118. ACM Press, 1978.

[Gal96] J. Galletly. *OCCAM 2. Including OCCAM 2.1.* UCL (University College London) Press Ltd, second edition, 1996.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine (A Users's Guide and Tutorial for Networked Parallel Computing)*. The MIT Press, 1994.

[GCC99] F. García, A. Calderón, and J. Carretero. MiMPI: A multithread-safe implementation of MPI. In J. Dongarra, E. Luque, and T. Margalef, editors, *Proceedings of EuroPVM/MPI 99, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes on Computer Science*, pages 207–214. Springer Verlag, 1999.

[GK90] I. Graham and T. King. *The Transputer Handbook*. Prentice Hall, 1990.

[GKPS97] A. Geist, J. A. Kohl, P. M. Papadopoulos, and S. L. Scott. Beyond PVM 3.4: What we've learned, what's next and why. In *Proceedings of EuroPVM/MPI 97, 4th European PVM/MPI Users' Group Meeting*, pages 3–5. Springer-Verlag, 1997.

[GL96] W. Gropp and E. Lusk. MPICH working note: The second-generation ADI for the MPICH implementation of MPI. Technical report, Argonne National Laboratory, USA, 1996.

[Gla84] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[Gla89] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.

[Gla90] A. S. Glassner. *Graphics Gems I*. Academic Press, 1990.

[GLS95] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation Series. The MIT Press, 1995.

[GN71] R. A. Goldstein and R. Nagel. 3D visual simulation. *Simulation*, 16(1):25–31, 1971.

[Gou71] H. Gouraud. Computer display of curved surfaces. *IEEE Transactions on Computers*, 20(6):623–629, 1971.

[GP89] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, 1989.

[Gra98] P. Gramblička. Cache techniky pre paralelný raytracing. Diploma thesis, Department of Informatics, Faculty of Mathematics and Physics, Comenius University, Bratislava, Slovakia, 1998.

[Gre91] S. Green. *Parallel Processing for Computer Graphics*. Research Monographs in Parallel and Distributed Computing. Pitman Publishing, 1991.

[Gro98] Object Management Group. *The Common Object Request Broker: Architecture and Specification (Revision 2.3)*, 1998.

[Gro02] W. Gropp. MPICH2: A new start for MPI implementations. In D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *Proc. of the 9th EuroPVM/MPI User's Group Conference (Recent Advances in Parallel Virtual Machine and Message Passing Interface)*, volume 2474 of *Lecture Notes in Computer Science*, pages 7–7. Springer-Verlag, 2002.

[GRS97] J. C. Gomez, V. Rego, and V. S. Sunderam. Efficient multithreaded user-space transport for network computing: Design and test of the TRAP protocol. *Journal of Parallel and Distributed Computing*, 40(1):103–117, 1997.

[GTGB84] C. M. Goral, D. E. Torrance, D. P. Greenberg, and G. Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, 18:213–222, 1984.

[GU77] G. H. Golub and R. Underwood. The Block Lanczos method for computing eigenvalues. In J. R. Rice, editor, *Mathematical Software III*, pages 361–377. Academic Press, 1977.

[HA98] A. Heirich and J. Arvo. A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing*, 12(1/2):57–68, 1998.

[HG86] E. A. Haines and D. P. Greenberg. The light buffer: a shadow testing accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, 1986.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HOB+99] L. P. Huse, K. Omang, H. Bugge, H. Ry, A. T. Haugsdal, and E. Rustad. ScaMPI—design and implementation. In H. Hellwagner and A. Reinefeld, editors, *SCI: Scalable Coherent Interface*, volume 11734 of *Lecture Notes in Computer Science*, pages 249–261. Springer-Verlag, 1999.

[How82] J. R. Howell. *A Catalog of Radiation Configuration Factors*. McGraw-Hill, 1982.

[HR03a] J. Hippold and G. Rünger. A communication API for implementing irregular algorithms on SMP clusters. In *Proc. of the 10th EuroPVM/MPI User's Group Conference (Recent Advances in Parallel Virtual Machine and Message Passing Interface)*, Lecture Notes in Computer Science. Springer-Verlag, 2003. (To appear).

[HR03b] J. Hippold and G. Rünger. Task pool teams for implementing irregular algorithms on clusters of smps. In *Proceedings of 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, pages 54–54, 2003.

[HS67] H. C. Hottel and A. F. Sarofin. *Radiative Transfer*. McGraw-Hill, 1967.

[HSS+98] L. S. Hebert, W. G. Seefeld, A. Skjellum, C. D. Taylor, and R. Dimitrov. MPI for Windows NT: Two generations of implementations and experience with the message passing interface for clusters and SMP environments. In H. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 309–316. CSREA Press, 1998.

[HV99] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley Longman, 1999.

[IRT] Internet Ray Tracing Competition. http://www.irtc.org.

[ISO90] ISO/IEC 9945-1:1990 Information Technology. *Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, 1990.

[ISO97] ISO/IEC 14772-1:1997 Information Technology. *Computer graphics and image processing—The Virtual Reality Modeling Language (VRML)*, 1997.

[Jan86] F. Jansen. Data structures for ray tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, editors, *Proc. of the Eurographics Seminar*, pages 57–73. Springer-Verlag, 1986.

[Jev89] D. A.J. Jevans. Optimistic multi-processor ray tracing. In *Proc. of Computer Graphics International'89 (New Trends in Computer Graphics)*, pages 507–522. Computer Graphics Society, 1989.

[Kaj86] J. T. Kajiya. The rendering equation. *Computer Graphics*, 20(4):143–150, 1986.

[Kap85] M. R. Kaplan. Space tracing: A constant time ray tracer. SIGGRAPH '85 Tutorial, 1985.

[Kel94] A. Keller. A quasi-Monte Carlo algorithm for the global illumination problem in the radiosity setting. Technical Report 260/94, Universität Kaiserslautern, Fachbereich Informatik, 1994.

[Kel96] A. Keller. Quasi-Monte Carlo radiosity. Technical Report 279/96, Universität Kaiserslautern, Fachbereich Informatik, 1996.

[Kel97] A. Keller. Instant radiosity. *Computer Graphics*, pages 59–56, 1997.

[KH95] M. J. Keates and R. J. Hubbold. Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum*, 14(4):189–202, 1995.

[KHS96] O. Krone, B. Hirsbrunner, and V. S. Sunderam. PT-PVM+: A portable platform for multithreaded coordination languages. *Calculateurs Parallèles*, 8(2):167–182, 1996.

[KNK+88] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 4:197–209, 1988.

[KNS87] H. Kobayashi, T. Nakamura, and Y. Shigei. Parallel processing of an object space for image synthesis using ray tracing. *The Visual Computer*, 3(1):13–22, 1987.

[KR98] A. Keller and A. Reinefeld. CCS resource management in networked HPC systems. In *Proc. Heterogenous Computing Workshop HCW'98 at IPPS*, pages 44–56, Orlando, Florida, 1998. IEEE Computer Society Press.

[KRR94] K. Kremer, Thomas Römke, and Friedhelm Ramme. A distributed computing center software for the efficient use of parallel computer systems. In *HPCN 94*, volume 797-II of *Lecture Notes in Computer Science*, pages 129–136. Springer-Verlag, 1994.

[KS02] D. Kranzlmüller and M. Schulz. Notes on nondeterminism in message passing programs. In D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *Proc. of the 9th EuroPVM/MPI User's Group Conference (Recent Advances in Parallel Virtual Machine and Message Passing Interface)*, volume 2474 of *Lecture Notes in Computer Science*, pages 357–367. Springer-Verlag, 2002.

[KW85] C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[LAB93] P. Liu, W. Aiello, and S. Bhatt. An atomic model for message-passing. In *The 5th Annual ACM Symposium on Parallel Architectures and Algorithms (SPAA'93)*, pages 154–163. ACM Press, 1993.

[LM-02] LM-63-1986 Illuminating Engineering Society of North America (IESNA) and American National Standards Institute (ANSI). *IES*

Recommended Standard File Format for Electronic Transfer of Photometric Data, 2002.

[LRBB96] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating polling, interrupts, and thread management. In *Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pages 13–22. IEEE, 1996.

[Lüc03] S. Lücking. Berechnung von Caustics in 3D-Rendering-Programmen. Studienarbeit, University of Paderborn, Department of Computer Science, 2003.

[LW93] E. P. Lafortune and Y. D. Willems. Bi-directional path-tracing. In *Proceedings of Compugraphics*, pages 143–153. Alvor, 1993.

[LW94] E. P. Lafortune and Y. D. Willems. Using the modified Phong reflectance model for physically based rendering. Technical Report CW 197, Department of Computer Science, Katholieke Universiteit Leuven, 1994.

[Lyn93] A. Lyne. Indecent proposal. (Film), 1993.

[McN00] A. McNamara. *Comparing Real and Synthetic Scenes using Human Judgements of Lightness*. PhD thesis, University of Bristol, 2000.

[MPI94] MPI Forum. *MPI: Message Passing Interface*, 1994. Version 1.0.

[MPI97] MPI Forum. *MPI-2: Extensions to the Message Passing Interface*, 1997. Version 2.0.

[MPI98] MPI Forum. *MPI: Message Passing Interface*, 1998. Version 1.1.

[MS87] D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[New52] I. Newton. *Opticks: Or a Treatise of the Reflections, Inflections and Colours of Light*. Dover Pubns, 1952. Preface by B. Cohen.

[NL96] M. L. Netto and B. Lange. Exploiting multiple partitioning strategies for an evolutionary ray tracer supported by DOMAIN. In *First Eurographics Workshop on Parallel Graphics and Visualisation*, 1996.

[Nus28] W. Nusselt. Graphische Bestimmung des Winkelverhaltnisses bei der Wärmestrahlung. *Zeitschrift des Vereines Deutscher Ingeneure*, 19(3):72–673, 1928.

[OPR96] R. Otte, P. Patrick, and M. Roy. *Understanding CORBA: the common object request broker architecture*. Prentice Hall, 1996.

[Par94] Parsytec GmbH. *Parix V1.3 PowerPC Software Documentation*, 1994.

[PB89] T. Priol and K. Bouatouch. Static loadbalancing for parallel ray tracing on MIMD hypercube. *The Visual Computer*, 5:109–119, 1989.

[PBMH02] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *Computer Graphics*, 21(3):703–712, 2002.

[Per85] Y. Perelman. *Fun With Maths and Physics*. Firebird Publications, Inc, 1985.

[Pho75] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[Pie93] G. Pietrek. Fast calculation of accurate formfactors. In *Proceedings of the 4th Eurographics Workshop on Rendering*, pages 201–220, 1993.

[Pit93] P. Pitot. The Voxar project. *IEEE Computer Graphics and Applications*, pages 27–33, 1993.

[PKGH02] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics*, 21(3):703–712, 2002.

[Pla98] T. Plachetka. POV||Ray: Persistence of Vision parallel raytracer. In L. Szirmay-Kalos, editor, *Proceedings of Spring Conference on Computer Graphics*. Comenius University, Bratislava, 1998.

[Pla02a] T. Plachetka. Perfect load balancing for demand-driven parallel ray tracing. In B. Monien and R. Feldman, editors, *Proceedings of Euro-Par 2002 (Parallel Processing)*, volume 2400, pages 410–419. Springer, 2002.

[Pla02b] T. Plachetka. (Quasi-) thread-safe PVM and (Quasi-) thread-safe MPI without active polling. In D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *Proc. of the 9th EuroPVM/MPI User's Group Conference (Recent Advances in Parallel Virtual Machine and Message Passing Interface)*, volume 2474 of *Lecture Notes in Computer Science*, pages 296–305. Springer-Verlag, 2002.

[PMTR95] I. S. Pandzic, N. Magnenat-Thalmann, and M. Roethlisberger. Parallel raytracing on the IBM SP2 and T3D. In *EPFL Supercomputing Review (Proceedings of First European T3D Workshop in Lausanne)*, volume 7, 1995.

[PS98] B. V. Protopopov and A. Skjellum. A multi-threaded message passing interface (MPI) architecture: performance and program issues. Technical report, Computer Science Department, Mississippi State University, 1998.

[PSA01] T. Plachetka, O. Schmidt, and F. Albracht. The HiQoS rendering system. In L. Pacholski and P. Ružička, editors, *Proc. of the 28th Annual Conf. on Current Trends in Theory and Practice of Informatics (SOFSEM 2001: Theory and Practice of Informatics)*, volume 2234 of *Lecture Notes in Computer Science*, pages 304–315. Springer-Verlag, 2001.

[PT] POV-Team. Persistence of Vision Ray Tracer (POV-Ray). http://www.povray.org.

[Ray88] M. Raynal. *Distributed Algorithms and Protocols*. J. Wiley&Sons, 1988.

[RC97] E. Reinhard and A. Chalmers. Message handling in parallel RADIANCE. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 486–493. Springer-Verlag, 1997.

[RCJ98] E. Reinhard, A. G Chalmers, and F. W. Jansen. Overview of parallel photo-realistic graphics. In A. de Sousa and B. Hopgood, editors, *State-of-the-Art reports, Eurographics '98 Conference*, pages 1–25. Springer, 1998.

[RCJ99] E. Reinhard, A. Chalmers, and F. W. Jansen. Hybrid scheduling for parallel rendering using coherent ray tasks. In *IEEE Parallel Visualisation and Graphics Symposium*, pages 21–28, 1999.

[Ree97] L. Reeker. *GOLEM Dokumentation*. University of Paderborn, 1997.

[Rei96] E. Reinhard. A parallelisation of ray tracing with diffuse interreflection. In *Advanced School for Computing and Imaging (ASCI '96)*, pages 367–372, 1996.

[RKC98] E. Reinhard, A. J. F. Kok, and A. Chalmers. Cost distribution prediction for parallel ray tracing. In K. Bouatouch, A. Chalmers, and T. Priol, editors, *Rendering Techniques '96, Proceedings of*

the Eurographics Workshop on Parallel Graphics and Visualisation, pages 77–90. Springer, 1998.

[RKJ98] E. Reinhard, A. J. F. Kok, and F. W. Jansen. Cost distribution prediction for ray tracing. In X. Pueyo and P. Schröder, editors, *Rendering Techniques '98, Proceedings of the Eurographics Workshop on Rendering*, pages 41–50. Springer, 1998.

[RW80] S. Rubin and T. Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, 1980.

[SAS92] B. E. Smits, J. R. Arvo, and D. H. Salesin. An importance-driven radiosity algorithm. *Computer Graphics*, 26(2):273–282, 1992.

[SC88] I. D. Scherson and E. Caspary. Multiprocessing for ray tracing: A hierarchical self-balancing approach. *Visual Computer*, (4), 1988.

[Sch93] C. Schlick. A customizable reflectance model for everyday rendering. In *Proceedings of the 4th Eurographics Workshop on Rendering*, pages 73–84, 1993.

[Sch00] O. Schmidt. *Parallele Simulation der globalen Beleuchtung in komplexen Architekturmodellen*. PhD thesis, Department of Mathematics and Informatics, University of Paderborn, 2000.

[SH93] P. Schröder and P. Hanrahan. On the form factor between two polygons. *Computer Graphics*, pages 163–164, 1993.

[SK99a] L. Szirmay-Kalos. Monte-Carlo global illumination methods—state of the art and new developments. In J. Žára, editor, *Proceedings of Spring Conference on Computer Graphics*, pages 3–21. Comenius University, Bratislava, 1999.

[SK99b] L. Szirmay-Kalos. *Monte-Carlo Methods in Global Illumination*. Institute of Computer Graphics, Vienna University of Technology, 1999.

[SKH96] K. R. Subramaniam, S. C. Kothari, and D. E. Heller. A communication library using active messages to improve performance of PVM. *Journal of Parallel and Distributed Computing*, 39(2):146–152, 1996.

[SOHL+95] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[SP89] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. *Computer Graphics*, 23(3), 1989.

[SP94] F. Sillion and C. Puech. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers, 1994.

[Ste94] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Professional Computing Series. Addison-Wesley, 1994.

[Suna] Sun Microsystems. Java 3D. http://java.sun.com/products/java-media/3D/index.html.

[Sunb] Sun Microsystems. The source for Java technology. http://java.sun.com.

[Tre97] R. Treumann. Experiences in the implementation of a thread safe, threads based MPI for the IBM RS/6000 SP. Technical report, IBM, T. J. Watson Research Center, 1997.

[TS67] K. E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. *Journal of Optical Society of America*, 57(9):1105–1114, 1967.

[Vea97] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997.

[VG94] E. Veach and L. Guibas. Bidirectional estimators for light transport. In *Proceedings of the Eurographics Workshop on Rendering*, pages 147–162, 1994. Also in G. Sakas, P. Shirley and S. Müller (editors), *Photorealistic Rendering Techniques*, Springer-Verlag, 1995.

[vNe66] J. von Neumann and A. W. Burks (editor). *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[War94] G. J. Ward. The RADIANCE lighting simulation and rendering system. *Computer Graphics*, pages 459–472, 1994.

[WEH89] J. R. Wallace, K. A. Elmquist, and E. A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, 23:315–324, 1989.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[Woo84] J. R. Woodwark. A multiprocessor architecture for viewing solid models. *Display Technology and Applications*, 5(2), 1984.

[WRC88] G. J. Ward, F. Rubinstein, and R. Clear. A ray tracing solution for diffuse interreflection. *Computer Graphics*, 22(4):85–92, 1988.

[WS95] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Professional Computing Series. Addison-Wesley, 1995.

[ZG98] H. Zhou and A. Geist. LPVM: A step towards multithread PVM. *Concurrency—Practice and Experience*, 10(5):407–416, 1998.