Unifying Framework for Message Passing

Tomas Plachetka
Comenius University, Bratislava

Unifying Framework for Message Passing
or
Why NOT (e.g.) MPI
Overview

• Novel formal universal framework for communication systems, independent on hardware, programming language etc. (similar to the framework for transactional database systems)
• Theorem: Our restricted model can simulate asynchronous channel model and vice versa
• Theorem: MPI (Message Passing Interface) model cannot (reasonably) simulate our restricted model
• Corollary: Asynchronous channel model is stronger than MPI (MPI lacks asynchronous communication)
• Conclusions
Channel message passing

Each process can only access its own memory
Each process is assigned a unique identifier (0, 1, ..., N)
Processes exchange data via messages
A message is passed between a process and a channel
Processes use non-blocking PUT(ch, m) and blocking GET(CH, m)
Processes communicate via **unbounded** channels. A channel is a FIFO (first-in-first-out).

![Diagram showing processes communicating via a channel](image)
Processes communicate via unbounded channels. A channel is a FIFO (first-in-first-out).
Processes communicate via **unbounded** channels. A channel is a FIFO (first-in-first-out).
Processes communicate via **unbounded** channels. A channel is a FIFO (first-in-first-out).

![Diagram showing processes communicating via a channel](image-url)
Synchronous (blocking) \texttt{SYNC\_PUT(ch, m)} can be simulated using asynchronous (non-blocking) \texttt{PUT(ch, m)}:
Synchronous (blocking) \texttt{SYNC\_PUT(ch, m)} can be simulated using asynchronous (non-blocking) \texttt{PUT(ch, m)}:
Synchronous (blocking) `SYNC_PUT(ch, m)` can be simulated using asynchronous (non-blocking) `PUT(ch, m)`:

![Diagram showing simulation process](image)
Synchronous (blocking) \texttt{SYNC\_PUT(ch, m)} can be simulated using asynchronous (non-blocking) \texttt{PUT(ch, m)}:
Synchronous (blocking) `SYNC_PUT(ch, m)` can be simulated using asynchronous (non-blocking) `PUT(ch, m)`: 
Synchronous (blocking) \texttt{SYNC\_PUT(ch, m)} can be simulated using asynchronous (non-blocking) \texttt{PUT(ch, m)}:
Synchronous (blocking) `SYNC_PUT(ch, m)` can be simulated using asynchronous (non-blocking) `PUT(ch, m)`:

![Diagram showing processes and channels]
Channel message passing

Is this allowed? (Two processes simultaneously receiving on the same channel.)

**YES**

Which process receives the message $M$?

Either $R1$ or $R2$. (Not both $R1$ and $R2$.)
Channel message passing

Is this allowed? (Two processes simultaneously sending to the same channel.)

**YES**

Which message will be received by the process $R$?

Either $M1$ or $M2$ (some of these).
Channel message passing

Is this allowed? (Same process simultaneously sending and receiving on the same channel.)

**YES** (Think of *P1* as of your department. People of the department communicate using a shared departmental mailbox.)

![Diagram showing process P1 and channel C1](image.png)
Each process can only access its own memory.

Each process is assigned a unique **identifier** \((0, 1, \ldots, N)\).

Processes exchange data via messages.

A message is passed between two processes (point-to-point).

Processes use **non-blocking send** and **blocking recv**.

A message can be sent from any process to any other one.
Point-to-point message passing

Message passing implemented as a library

send and recv are function calls; the communication library hides the implementation of these functions from the programmer.

The same application can run on a distributed-memory cluster as well as on a shared-memory multiprocessor without a change in the application.
Point-to-point message passing

Message passing implemented as a library

send and recv are function calls; the communication library hides the implementation of these functions from the programmer.

The same application can run on a distributed-memory cluster as well as on a shared-memory multiprocessor without a change in the application.
Point-to-point message passing (MPI)

**MPI_Isend** and **MPI_Recv** are function calls; the communication library hides the implementation of these functions from the programmer.

### Context of a non-blocking send
1. Allocate a send buffer
2. Pack data into the send buffer
3. MPI_Isend(recipient_id, buf, &req)
4. Continue working
5. MPI_Wait(req) or MPI_Test(req)
6. Free the send buffer

### Context of a blocking receive
1. Allocate a receive buffer
2. MPI_Recv(sender_id, buf)
3. Unpack data from buffer
4. Free the send buffer
Sending a message

1. Allocate a buffer
   
   new(s);
   semaphore_init(s, 0);
   [CREATE, sender, NULL, m, NULL, s, t];
   semaphore_wait(s);
   delete(s);

2. Put data into the buffer

3. Send the buffer to the receiver
   
   [SEND, sender, receiver, m, NULL, NULL, t];
Sending a message

1. Allocate a buffer
   new(s);
   semaphore_init(s, 0);
   [CREATE, sender, NULL, m, NULL, s, t];
   semaphore_wait(s);
   delete(s);

2. Put data into the buffer

3. Send the buffer to the receiver
   [SEND, sender, receiver, m, NULL, NULL, t];
Sending a message

1. Allocate a buffer
   ```
   new(s);
   semaphore_init(s, 0);
   [CREATE, sender, NULL, m, NULL, s, t];
   semaphore_wait(s);
   delete(s);
   ```

2. Put data into the buffer

3. Send the buffer to the receiver
   ```
   [SEND, sender, receiver, m, NULL, NULL, t];
   ```

Questions:

Who should decide how large a send buffer to allocate? **SENDER**

Who should free the send buffer? **SYSTEM** NOT IN MPI!
Receiving a message

1. Receive a message to a buffer

```c
new(s);
semaphore_init(s, 0);
RECV, receiver, sender, m, accept_all, s, t];
semaphore_wait(s);
delete(s);
```

2. Read data from the buffer

3. Free the buffer

```c
DESTROY, receiver, NULL, m, NULL, NULL, NULL, t];
```
Receivers's view (TP)

Receiving a message
1. Receive a message to a buffer
   new(s);
   semaphore_init(s, 0);
   [RECV, receiver, sender, m, accept_all, s, t];
   semaphore_wait(s);
   delete(s);
2. Read data from the buffer
3. Free the buffer
   [DESTROY, receiver, NULL, m, NULL, NULL, NULL, t];
Receiving a message

1. Receive a message to a buffer
   new(s);
   semaphore_init(s, 0);
   [RECV, receiver, sender, m, accept_all, s, t];
   semaphore_wait(s);
   delete(s);

2. Read data from the buffer

3. Free the buffer
   [DESTROY, receiver, NULL, m, NULL, NULL, t];

Questions:

Who should decide how large a receive buffer to allocate? SYSTEM NOT IN MPI!

Who should free the receive buffer? RECEIVER
The system shares a part of memory with each process. This memory is called scope of a process \(\text{SC}(x)\) denotes scope of process \(x\), \(\text{SC}(\ast)\) denotes union of scopes of all processes). Scopes store messages, semaphores and yet not executed basic operations.

The system reads streams of basic operations from processes and executes them (both the reading and the execution may be postponed).

Operations are tuples \([\text{op}, x, Y, m, f, s, t]\), where:

- \(\text{op} \in \{\text{CREATE, DESTROY, SEND, RECV}\}\)
- \(x\) is the identifier of process submitting this operation
- \(Y\) is a set of process identifiers
- \(m\) is a message
- \(f\) is boolean function defined on messages (a filter)
- \(s\) is a semaphore
- \(t\) is the timestamp of submission of this operation
Execution of \([\text{CREATE}, \ x, \ , \ m, \ , \ s, \ , \] \) (TP)

1. Create new message \(m\) in \(SC(x)\)
2. If \(s \neq \text{NULL}\) then \(\text{semaphore\_signal}(s)\)
3. Remove this operation from \(SC(x)\)
1. Remove message m from SC(x)
2. If s ≠ NULL then semaphore_signal(s)
3. Remove this operation from SC(x)
Execution of \([\text{RECV} / \text{SEND}, x, Y, m, f, s, t]\) (TP)

\[\text{BR} = [\text{RECV}, x, Y, m, f, s, t] \quad \text{BS} = [\text{SEND}, x', Y', m', f', s', t']\]

BR and BS are a **matching operation pair** iff

\[x \in Y' \quad \text{and} \quad x' \in Y \quad \text{and} \quad f(m')\]

plus some time-stamp properties must hold if ordering of messages is important (only the oldest such operations match)

Matching BR and BS are executed simultaneously:

1. Create new message \(m\) in \(\text{SC}(x)\)
2. Copy contents of \(m'\) into \(m\)
3. Remove \(m'\) from \(\text{SC}(x')\)
4. If \(s \neq \text{NULL}\) then \(\text{semaphore\_signal}(s)\)
5. If \(s' \neq \text{NULL}\) then \(\text{semaphore\_signal}(s')\)
6. Remove \(\text{BR}\) from \(\text{SC}(x)\)
7. Remove \(\text{BS}\) from \(\text{SC}(x')\)

**Progress guarantee:** If a matching pair \(\text{BR}\) and \(\text{BS}\) exists then at least one of \(\text{BR}\) and \(\text{BS}\) will be eventually executed.
What cannot be done with MPI and can be done with TP

p0(FILE *inp0)
{
    while(! feof(inp0))
    {
        new(m); /* [create...] */
        m = fgetc(inp0);
        async_send(p1, m);
        printf("sent");
    }
}

p1(FILE *inp1)
{
    while(! feof(inp1))
    {
        sync_recv(p0, m);
        printf("received %c", m);
        delete(m); /* [destroy...] */
        fgetc(inp1);
    }
}

Equivalent program cannot be written using MPI functions without breaching the bounds of the invariance thesis:

"'Reasonable' machines can simulate each other with a constant factor overhead in space and a polynomial factor overhead in time." [van Emde Boas]
Top 10 reasons why to use MPI

1. MPI has more than one freely available, *quality implementation*.

2. MPI defines a 3rd party profiling mechanism.

3. MPI has *full asynchronous communication*.

4. MPI groups are solid, efficient, and deterministic.

5. MPI *efficiently manages message buffers*.

6. MPI synchronization protects 3rd party software.

7. MPI can efficiently program MPP and clusters.

8. MPI is totally portable.

9. MPI *is formally specified*.

10. MPI *is a standard*. 

http://www.lam-mpi.org/mpi/mpi_top10.php
Message Passing Framework (TP)

Application process
(arbitrary entity with unique identifier)

Language binding

Basic msg passing operations
recv, send, create, destroy
(four basic operations with formally defined semantics)

Architecture binding

Message passing system
(implementation of basic operations for a specific architecture)

Transaction
(arbitrary entity with unique identifier)

Language binding

Basic database operations
read, write, insert, delete
(four basic operations with formally defined semantics)

Architecture binding

Database system
(implementation of basic operations for a specific architecture)
Conclusions

- MPI has no asynchronous communication (shame!) although MPI developers say otherwise (SHAME!)
- MPI is unjustly presented as industrial standard (shame!) and often also as academic standard (SHAME!)
- Want a provably better standard? You have just seen one:
  - Our restricted model is as powerful as asynchronous channel model (and other theoretical models); **our unrestricted model is at least as powerful**
  - Our framework can help in building practical message passing systems. It defines **formal semantics of four basic operations**, of which more complex operations consist
  - **Our framework can be efficiently implemented for variety of architectures, ranging from Transputer-based systems to practically all modern systems**