Implementácia variantu systému Program composition notation

DIPLOMOVÁ PRÁCA

Jozef Vavro

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Informatika

Vedúci diplomovej práce
Damas Gruska, RNDr., PhD.

BRATISLAVA 2006

Čestne vyhlasujem, že diplomovú prácu som vypracoval samostatne s použitím literatúry a elektronických dokumentov uvedených v zozname použitej literatúry.

.....

Jozef Vavro

Ďakujem svojmu diplomovému vedúcemu RNDr. Damasovi Gruskovi, PhD., za cenné rady a pripomienky pri vypracovávaní diplomovej práce.

Abstrakt

V tejto práci popisujem systém s názvom Program Composition Notation (PCN), ktorý vznikol v Argonne National Laboratory. Tento systém zjednodušuje písanie paralelnej aplikácie. Pozostáva z dvoch častí, a to z jazyka a run-time systému. Hlavnými prvkami jazyka je princíp skladania programov a definičné premenné, ktoré riešia prístup k zdielaným dátam. Sekvenčné výpočty v PCN je možné vykonať pomocou programov napísaných v jazykoch C a Fortran. PCN program je nezávislý od architektúry, okrem programov, ktoré používajú C/Fortran zdrojové súbory. PCN bol použitý na vývoj aplikácii v rôznych oblastiach, ako je modelovanie počasia, chémia a biológia. Posledná verzia je 2.0 z roku 1993 a pracuje pod systémami Sun-4, NeXT, RS/6000, SGI, iPSC/860, Touchstone DELTA a Symmetry/Dynix.

V prvej časti tejto práce popisujem syntax a sémantiku jazyka PCN, jeho výpočtový model a abstraktné zariadenie, ktoré vyhodnocuje program napísaný v jazyku PCN. V druhej časti špecifikujem požiadavky pre implementáciu vlastného variantu systému PCN. Je to systém MyPE (My Parallel Engine) a implementácia vlastnej verzie jazyka PCN, jazyk PCN Lite. MyPE je založený na hlavných myšlienkach systému PCN. Je programovaný v jazyku C++ a poskytuje rozhranie, pre programovanie paralelných aplikácií, pracujúcich v počítačovej sieti operačného systému Windows XP. PCN Lite je inšpirovaný jazykom PCN a implementovaný pod operačným systémom Windows XP. V druhej časti tiež prezentujem implementáciu MyPE a PCN Lite a zdôvodňujem svoje rozhodnutia, týkajúce sa spôsobu implementácie jednotlivých častí. Na záver uvádzam jednoduchý príklad napísaný v jazyku PCN, jazyku PCN Lite a MyPE.

Predhovor

Paralelné programovanie umožňuje programátorovi napísať aplikáciu, ktorá oproti sekvenčnej verzii môže byť omnoho efektívnejšia. V súčasnosti ako študent vnímam príchod procesorov, ktoré umožňujú paralelné programovanie už v segmente domácich počítačov a vidím, že tento spôsob programovania robí niektorým vývojárom problem, čo sa potom odráža na kvalite softwaru.

Paralelné programovanie ako trend sa prirodzene rozširuje a je nutné, aby dnešní a budúci vývojari mali k dispozícii vhodné vývojové nástroje. V tejto práci sa budeme zaoberať jedným z takých systémov a tiež systémom, ktorý som vytvoril ako zadanie diplomovej práce.

Obsah

Abstrakt 5
Predhovor 6
Obsah 7
    Zoznam obrázkov 9
Úvod 10
Ciele 10
Systém PCN 10
    Jazyk PCN 11
    Programovací model 11
    Základná syntax a dátové typy 11
    Sekvenčná kompozícia 13
    Podmienková kompozícia 13
    Cyklus a rekurzia 14
    Definičná premenná 14
    Paralelná kompozícia 15
    Dátový typ tuple a zoznam 15
    Viacjazyčné programovanie 16
    Moduly 17
    Metavolania 17
    Dátový typ port 18
    Mapovanie procesov a virtuálna topológia 18
    Portable Run-Time system pre PCN 19
    Core PCN 20
    The Program Composition Machine 21
My Parallel Engine 23
    História vývoja 23
    Jazyk PCN Lite 25
    Návrh kompilátora a jeho implementácia 25
    Základná syntax a dátové typy 26
    Sekvenčná kompozícia 27
    Podmienková kompozícia 27
    Cyklus a rekurzia 28
    Definičná premenná 30
    Paralelná kompozícia 32
    Dátový typ tuple a zoznam 33
    Viacjazyčné programovanie 33
    Moduly 33
    Metavolania 34
    Dátový typ port 34
    Príklad 34

Zhrnutie ... 36
Knižnica MyPE ... 37
Architektúra ... 37
Server ... 38
Inicializácia ... 39
Registrácia funkcií pre uživateľsky definované príkazy ... 39
Klient ... 40
PCN ideiy v MyPE ... 41
Zhrnutie ... 44
Príklad ... 45
Záver ... 47
Bibliografia ... 49

Zoznam obrázkov

Obr. 1. 20
Obr. 2. 37

Úvod

Tvorba programu, ktorý využíva pre svoj beh viacero threadov, je náročná úloha. Preto je potrebné mať nástroj, ktorý umožňuje jednoducho a flexibilne vyvíjať paralelné aplikácie. Takýto nástroj by mal mať jazyk, ktorý je nezávislý od hardwaru a od operačného systému. Mal by umožňovať tvorbu prenositeľného kódu a podporovať viacero architektúr. V tejto práci sa budeme jednému takému systému venovať. Jeho názov je Program Composition Notation zjednodušene PCN.

Ciele

Cieľom tejto práce je implementovať variant systému PCN. Na dosiahnutie tohto cieľa musím popísať systém PCN. Tomu je venovaná nasledujúca časť. Po nej nasleduje kapitola, v ktorej popisujem svoj systém a porovnávam s pôvodným PCN.

Systém PCN

PCN je navrhnutý pre vývoj paralelných programov a ich vyhodnocovanie. Vyznačuje sa jednoduchým, ale silným programovacím jazykom PCN. Jedna z jeho vlastností je využitie existujúcého sekvenčného kódu jazyka C alebo Fortran. Tento systém bol napísaný pre architektúry delta, ipsc860, iris, next040, rs6000 a sun4.

V čase písania tejto práce sa mi inú ako pôvodnú implementáciu nepodarilo nájsť. Aktuálna verzia je z roku 1993 vo verzii 2.0. Bol použitý napr. na modelovanie počasia (9), v biológii a chémii a pod.

Systém PCN pozostáva z programovacieho jazyka, ktorý je popísaný v kapitole Jazyk PCN, a z programového systému, ktorý popisujem v časti Portable Run-Time system pre PCN.

Jazyk PCN

V tejto časti popíšem syntax jazyka PCN, jeho programovací model a dátové typy. Jednotlivé časti popisujem detailne, ale nesnažím sa nahradiť referenčnú príručku pre PCN (1).

Programovací model

PCN používa štyri jednoduché myšlienky pre vytvorenie paralelného programovacieho modelu. Sú to:

a. **definičná premenná**, ktorá sa používa na komunikáciu a synchronizáciu procesov
b. **paralelná kompozícia** je jednoduchý mechanizmus na spájanie jednoduchých komponentov do komplexného systému. Programovací systém, ktorý podporuje paralelnú kompozíciu, je taký, ktorý zachováva vlastnosti programových komponentov po ich spojení do jedného celku. To znamená, že správanie sa celku je logickou kombináciou správania sa jednotlivých častí (8).
c. **nedeterministický výber**
d. **stav výpočtu**, pojem známy zo sekvenčných jazykov, ktorý je rovnako dôležitý pre paralelné programovanie. Je však dôležité dávať pozor na možný nedeterminizmus a tiež používať špeciálne techniky pre komunikáciu procesov a ich synchronizáciu. Samotná zmena stavu sa potom vykoná v sekvenčnej časti programu.

Základná syntax a dátové typy

Syntax jazyka je veľmi podobná jazyku C.

Dátové typy sú:

a. int – celé čísla
b. double – reálne čísla
c. char – znaky

Konštanty sú podľa normy ANSI C.

Reťazec je reprezentovaný ako pole znakov ukončené nulou tak ako v jazyku C.

Aritmetické výrazy sú ako v jazyku C, no používajú sa len operacie sčítania, odčítania, násobenia, delenia a modulo (`+`, `-`, `*`, `/`, `%`). Priorita operátorov a ich asociatívnosť je ako v jazyku C.

Operátor priradenia je :=.

Názvy premenných sú ako v jazyku C. Názov je reťazec znakov z množiny {a–z , A–Z, 0–9, _} a musí začínať písmenom alebo podčiarkovníkom _. Rozlišujú sa malé a veľké písmená. Na dĺžke nezáleží.

Komentáre začínajú s /* a končia s */ tak ako v C.

Procedúra má svoj názov, potom nasleduje zoznam formálnych parametrov. Tieto sú spracovávané referenciou, nie hodnotou ako v C. Po nich nasleduje deklarácia lokálnych premenných. Za ňou sa nachádza telo procedúry, ktoré je zložené z blokov. Blok je základný component, z ktorého sa vytvárajú procedúry a je buď kompozícia, priradenie, definičné priradenie, implikácia, alebo volanie procedúry.

```
názov(formálne argumenty)
deklarácie premenných
blok
```

Funkcie sú ako procedúry, ale môžu vrátit hodnotu pomocou kľúčového slova return. Pred funkciu sa použije kľúčové slovo function. Funkcie sa nesmú používať v guard príkazoch (podmienkach).

```
function názov(formálne argumenty)
deklarácie premenných
blok
```

Oddelovač je buď čiarka (,), alebo bodkočiarka (;). Povolené sú aj oddeľovače za posledným blokom v kompozícii, tzv. trailing delimiters.

Deklarácia premenných pozostáva z názvu typu a jedného alebo viac mien pre premenné. Ak má byť premenná pole, potom sa za jej meno pridá sufix [n], kde n je veľkosť poľa a n je celé číslo. Ak je formálny argument procedúry pole, tak sa použije sufix [].

Sekvenčná kompozícia

Sekvenčná kompozícia je známy mechanizmus zo sekvenčných jazykov, ako sú C, Fortran alebo Pascal.

Nech P_1, ..., P_n sú bloky, n ≥ 0.

Syntax:

{; P_1, …, P_n}

Sémantika:
Vykoná blok P_1, potom blok P_2, … . Nakoniec vykoná blok P_n.

Podmienková kompozícia

Syntax:

{? $guard_0$ -> $block_0$, ..., $guard_k$ -> $block_k$}

Každý $guard_i$ je postupnost testov, napr. test rovnosti, nerovnosti, porovnávanie čísel a pod.

Sémantika:

„*guard -> block*“ budeme nazývať implikáciou. Potom podmienková kompozícia vyhodnocuje každú implikáciu tak, že najprv vyhodnotí všetky podmienky. Ak jedna alebo viac podmienok platí, tak sa vyberie jedna a príslušný blok sa vyhodnotí. Ak každý neplatí, tak sa ukončí podmienková kompozícia a nič sa nevyhodnotí. Tu je prípustný nedeterminizmus. Je zaujmavé, že autor referenčnej príručky ho povoľuje, ale autor iného článku ho zakazuje.

Cyklus a rekurzia

PCN implementuje rekurziu štandardne, ako je zaužívané v jazykoch C alebo Pascal a nepoužíva pre ňu žiadnu explicitnú syntax. Cyklus sa v PCN nazýva **quantification**.

Syntax:

```
{op i over low .. high :: block}
```

Sémantika:

Príkaz *block* sa vykoná práve raz pre každé `i` z intervalu <*low, high*> sekvenčne, ak `op` = `;`, alebo sa bloky vykonajú pre každé `i` paralelne, ak `op` = `||`.

Definičná premenná

Definičná premenná je špeciálny typ premennej jazyka PCN. Nedeklaruje sa a teda sa nešpecifikuje jej typ. Na začiatku je premenná nedefinovaná. Po priradení sa stáva definovanou. Definičná premenná sa dá definovať iba jeden raz. Na priradenie do definičnej premennej sa používa operátor =.

Čítanie z definovanej premennej dá k dispozícii hodnotu, ktorou je definovaná. Čítanie z nedefinovanej premennej spôsobí, že daný proces bude čakať. Zápis do nedefinovanej premennej spôsobí definovanie tejto premennej a spôsobí, že procesy, ktoré na ňu čakali, budú môcť ďalej pokračovať. Zápis do definovanej premennej spôsobí chybu.

V texte budem explicitne uvádzať, či je nejaká premenná definičná (tiež označujem pojmom definícia). Inak je to obyčajná premenná s neobmedzeným množstvom priradení a jej hodnota nemá vplyv na to, či proces má, alebo nemá čakať.

Paralelná kompozícia

Nech P_1, ..., P_n sú bloky, $n \geq 0$.

Syntax:

`{|| P1, …, Pn}`

Sémantika:
Spustí všetky bloky, aby sa vykonávali súčasne.

Dátový typ tuple a zoznam

Tuple je špeciálna dátová štruktúra, ktorá obsahuje konečný počet prvkov, nech ich je n. Potom jej ďalej budeme hovoriť n-tica.

Syntax:

`{a0, …, an}`

Sémantika:

a_i je buď konštanta, definičná premenná, alebo Tuple, pre $\forall i \in \{0, \ldots, n\}$.

Zoznam je dvojica, t.j. n-tica pre n = 2. Jej prvým prvkom je nejaká hodnota. Druhý prvok je zvyšok zoznamu. Ukončuje ho prázdny zoznam, ktorý sa označuje {}.

Syntax:

$\{a_0, \{a_1, \ldots, \{a_n, \{\}\}\ldots\}\}$

Špeciálna syntax:

$[a_0, a_1, \ldots, a_n]$

Dve n-tice $\{a_0, \ldots, a_n\}$ a $\{b_0, \ldots, b_n\}$ sa dajú porovnať po zložkách pomocou operátora ?=, tzv. „tuple match“. Uvažujme a_i a b_i bez ujmy na všeobecnosti. Pre b_i máme nasledovné možnosti:

a. b_i je konštanta, potom a_i musí byť tiež tá istá konštanta, inak porovnanie ?= je neúspešné
b. b_i je nedefinovaná definičná premenná, potom sa definuje na hodnotu a_i, ak a_i je konštanta
c. b_i a a_i sú definičné premenné, ak sú ich hodnoty rovné, môžeme porovnávať ďalšiu dvojicu, ak nie, porovnanie ?= je neúspešné.
d. b_i je k-tica pre nejaké k a musíme vyhodnotiť a_i ?= b_i úspešne, aby porovnanie tejto dvojice bolo úspešné.

Viacjazyčné programovanie

Ďalším a dôležitým mechanizmom v jazyku PCN je možnosť znovupoužitia existujúceho sekvenčného programu. Podpora je pre jazyk C a Fortran. Pravidlá pre volanie procedúry napísanej v jazyku C alebo Fortran sú nasledovné:

- argumenty pre volanie môžu byť typu int, char a double, alebo pole týchto typov a taktiež definičné premenné;
- pokiaľ je argumentom definičná premenná, tak sa volanie uskutoční až vtedy, keď táto premenná bude definovaná;
- parametre sa spracovávajú referenciou;
- procedúra, ktorú voláme, nesmie zmeniť definičnú premennú (toto pravidlo musí dodržať programátor);
- návratová hodnota sa uloží do obyčajnej premennej.

Moduly

Jednotlivé zdrojové súbory programu napísanom v jazyku PCN zodpovedajú modulom. Modul obsahuje nula a viac procedúr. Ak chceme volať procedúru z iného modulu, najskôr ju musíme sprístupniť ostatným modulom pomocou direktívy *export*. Potom na volanie použijeme nasledovnú konštrukciu:

$$modul{:}názov\_procedúry(arg_0, \ldots, arg_n)$$

Metavolania

Metavolanie je spôsob volania procedúry, v ktorom premenná reprezentuje jej názov. Táto premenná sa použije ako názov procedúry keď chceme zavolať nejakú procedúru s nejakými argumentmi. Syntax je nasledovná:

$$`modul':'procedure'(arg_0, \ldots, arg_n)$$

Sémantika:

modul a *procedure* sú premenné, ktoré obsahujú názov modulu a procedúry, ktorá sa má zavolať. Názov je konštantný reťazec. Teda napr. *modul* = `m', *procedure* = `f' spôsobí, že sa zavolá procedúra f z modulu m.

Dátový typ port

Dátový typ port sa používa na komunikáciu medzi procesmi. Používa sa nasledovne: Najprv vytvorím pole typu port s N prvkami. N musí byť celočíselný násobok hodnoty `nodes()` (táto funkcia vráti počet virtuálnych procesorov). Každý prvok poľa obsahuje definičnú premennú, ktorá sa používa na komunikáciu medzi procesmi. Môže sa použiť len dvakrát.

Mapovanie procesov a virtuálna topológia

Paralelná kompozícia definuje procesy, ktoré bežia súčasne. Definičné premenné definujú spôsob, akým tieto procesy spolu komunikujú. Ale to ešte nie je paralelný program. Musíme definovať, akým spôsobom vybrať pre každý proces procesor, na ktorom sa pustí. Na to slúži technika, ktorá sa nazýva „process mapping“, teda mapovanie procesov.

V PCN má programátor túto možnosť priamo pomocou konkrétnych funkcií. Najprv však treba spomenúť pojem virtuálnej topológie. Ak by sme popisovali topológiu pre každý počítač priamo, mohli by nastať problémy s prenositelnosťou paralelného programu. Dokonca by počítač nemusel vyhovovať požiadavkám pre daný program. Tento problém rieši virtuálna topológia. Každý počítač má v sebe uloženú informáciu o sebe. Programátor pomocou jednoduchých funkcií môže tieto informácie získať, spracovať a rozhodnúť, či nejaký proces na tomto počítači spustí, alebo nie. Spomeňme aspoň tri základné funkcie systému PCN, ktoré slúžia na tieto účely:

a. `topology()` – vráti nejakú k-ticu, ktorá si nesie informáciu o type počítača, napr. `{"mesh", 16, 32}, {"array", 512}` a pod.;
b. `nodes()` – vráti počet tzv. node, čo sú vlastné virtuálne procesory, na ktorých sa spúšťajú procesy;
c. `location()` – vráti polohu procesu v aktuálnom počítači

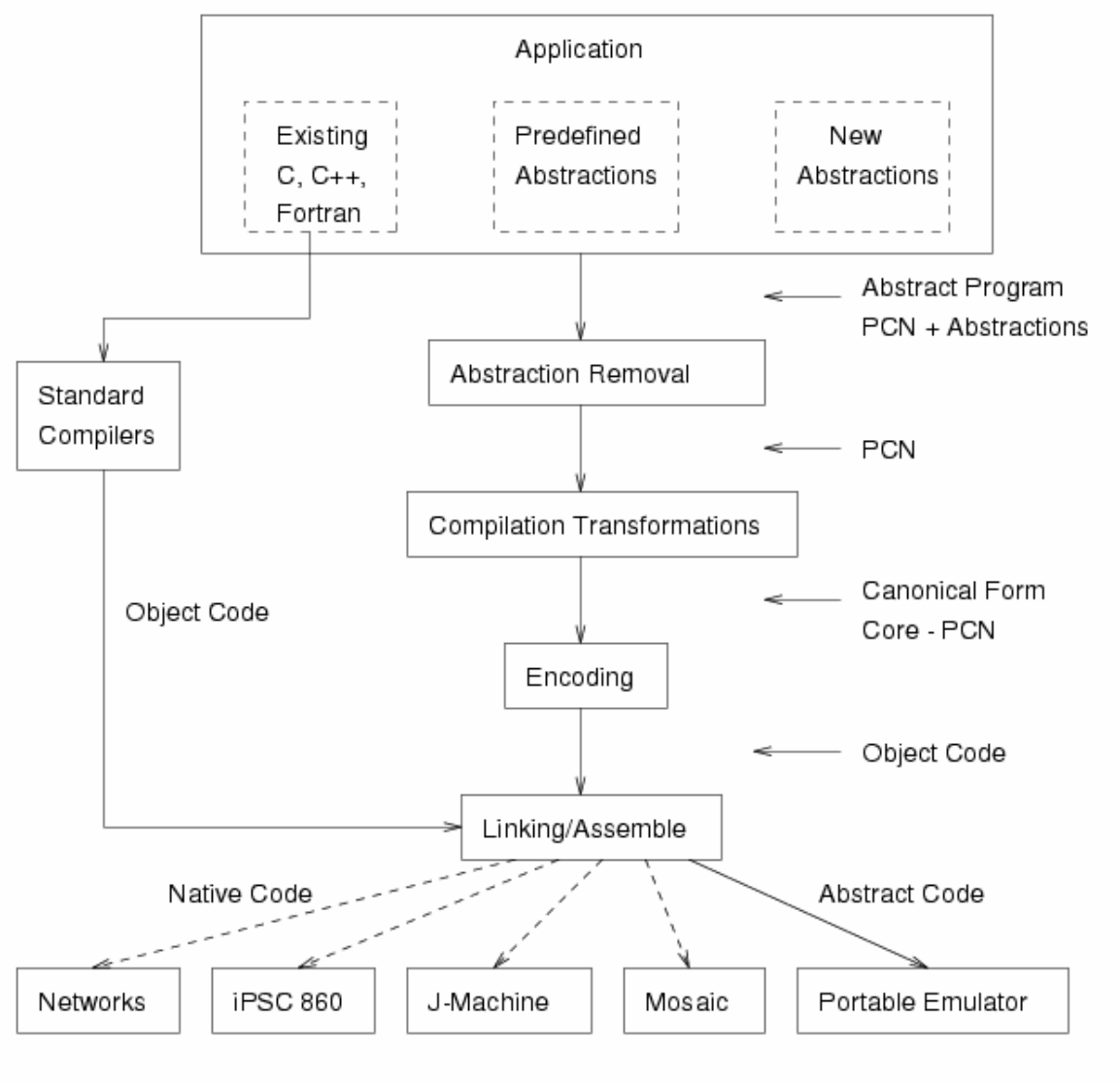
Portable Run-Time system pre PCN

Je software, ktorý zabezpečuje spúšťanie a analýzu programov napísaných v jazyku PCN. Spúšťanie zabezpečuje interpretér, ktorý je napísaný pre viaceré architektúry a tým umožňuje prenositeľnosť programov písaných v jazyku PCN. Analýza sa robí pomocou debuggeru a programu Gauge. Ten slúži na analýzu rýchlosti behu programu, t.j. napríklad počet volaní nejakej procedúry na virtuálnych procesoroch, čas strávený vyhodnocovaním nejakej procedúry, počet volaní procedúry, nevyužitý čas a iné.

V tejto časti popíšem interpretér jazyka PCN a to takým spôsobom, že popíšem abstraktné zariadenie, pomocou ktorého už nie je problém implementovať interpretér pre PCN jazyk. Pre detailné zoznámenie sa s týto zariadením doporučujem (2).

Interpretér nepracuje priamo s jazykom PCN, ale pracuje s jeho podmnožinou, ktorá sa nazýva **core PCN**. Ten je popísaný v nasledujúcej časti. Hlavné časti interpretéra sú popísané v časti Program Composition Machine. Pre každú z nich je vytvorená množina inštrukcií, ktorá s nimi pracuje. Tieto inštrukcie tu už neuvádzam a sú presne popísané v (2).

Nasledujúci obrázok z (7) názorne ukazuje proces kompilácie.



Obr. 1.

Core PCN

Definície. Proces, ktorý vykoná primitívnu operáciu, sa nazýva *kernel*. *Volanie* (*call*) je proces, ktorý spustí iný program. Kernel je *neblokujúci*, ak je to defininícia alebo ak je to priradenie, alebo iný kernel, pre ktorý sú vstupné argumenty dostupné v čase volania. Kernel je *blokujúci*, ak nevieme zistiť, či je neblokujúci.

Core PCN je podmnožina jazyka PCN, ktorého programy majú nasledovnú formu:

a. program má formu P = {? C_1, ..., C_n}, n > 0.

b. každá implikácia má tvar G → B, kde G je buď postupnosť testov, alebo prázdny test **true**.
c. každé B má formu `{; K1, ..., Kk, {|| P1, ..., Pm}}`, k, m ≥ 0, kde každý `Ki` je neblokujúci kernel a `Pj` sú volania. Ak k = 0, tak sekvenčný blok sa môže vynechať. Ak m < 2, paralelný blok sa môže vynechať.

The Program Composition Machine

je samotný interpretér a pozostáva z troch komponentov:

1. redukčný komponent

Poskytuje prostriedky na vyhodnocovanie core PCN programov. Udržuje si množinu procesov a opakovane robí výber a spustenie procesu. Spustenie procesu zahŕňa skúšanie každej implikácie v tele programu. Ak sa predpoklad niektorej implikácie vyhodnotí na true, tak sa vyhodnotí príslušná časť programu a to tak, že sa vyhodnotia kernely a vytvoria sa procesy, ktoré vyhodnotia jednotlivé volania. Inak sa proces vráti späť do množiny procesov.

Treba tu spomenúť dve optimalizácie, a to tzv. *tail recursion* a *scheduling structure*. Scheduling structure sa používa na to, aby sa nestrácal výkon pri procesoch, ktoré čakajú na dáta. Je zložené z aktívnej fronty, v ktorej sú procesy, ktoré majú dáta k dispozícii, a z tzv. suspension fronty, v ktorej sú tie procesy, ktorých dáta ešte nie sú k dispozícii. Tail recursion sa používa na to, aby sa proces po redukcii a zaradení do množiny procesov znovu nezvolil. Preto sa tento mechanizmus aplikuje len konečný počet krát a potom sa zvolí nový proces na spracovanie. Počet rekurzívnych volaní, ktoré sú povolené pred tým, než sa procesy prepnú, sa nazýva *timeslice*.

2. komunikačný komponent

Pracuje na konci každej timeslice a prijíma správy, ktoré prídu k procesoru. Môže modifikovať lokálne dátové štruktúry alebo posielať správy. Prijať môže 5 typov správ: *Read*, *Define*, *Value*, *Cancel* a *Collect*.

Read znamená, že niektorý z procesorov potrebuje lokálne dáta, keď budú k dispozícii. Value je odpoveď na Read. Define signalizuje, že niektorý z procesorov vykonal definičnú operáciu a zadefinoval lokálne dáta. Cancel oznamuje, že niektorý z procesorov už nepotrebuje niektoré referencie. Collect znamná, že nejaký procesor potrebuje tento procesor, aby mohol vykonať garbage collection.

3. garbage collection komponent

Garbage collection je proces, ktorý identifikuje nepotrebné dáta, ale stále zaberajú miesto v pamäti. Potom túto pamäť uvoľní. Tento komponent obsahuje garbage collection na dvoch úrovňach, na lokálnej a globálnej.

My Parallel Engine

Cieľom mojej diplomovej práce je implementovať variant systému PCN. Toto je veľmi voľná špecifikácia, preto v nasledujúcej časti uvádzam históriu vývoja môjho paralelného systému, v ktorej uvidíme počiatočnú špecifikáciu pre tento systém a jej postupný vývoj.

História vývoja

Na začiatku som si stanovil nasledovné požiadavky, ktoré by mal môj systém spĺňať:

1. podporovať operačný systém Windows a ak bude možné aj Linux, nie však nutne Linux;
2. systém musí byť použiteľný nielen na viacprocesorových počítačoch, ale aj v počítačovej sieti;
3. distribúciu paralelného programu, jeho spustenie na každom počítači a taktiež plánovanie spúšťania procesov (threadov) v paralelnom programe zabezpečí niektorý z voľne dostupných grid systémov;
4. použitý grid systém musí umožňovať písať aplikácie, ktoré budú využívať jeho služby, v jazyku C. Ale nesmie ísť len o spúšťanie úloh. Musí dovoliť napísať aplikáciu, ktorá sa spustí v gride, nie v počítači;
5. nepoužijem žiadny už existujúci paralelný systém, ako napr. PVM, MPI;
6. možnosť používať lubovoľný dátový typ pre definičné premenné;
7. možnosť používať komunikáciu medzi procesmi pomocou správ.

S touto špecifikáciou som už mohol začať vývoj no problematický bol bod 3. a hlavne bod 4. Nenašiel som žiaden grid systém, ktorý by umožnil napísať aplikáciu, ktorá by bežala v grid systéme. Tým myslím to, že grid sám bude vytvárať procesy a starať sa o ich plánovanie a distribúciu v počítačovej sieti. Žiaden zo systémov mi toto neumožňoval, alebo to nebolo uvedené v dokumentácii. Jediné, čo bolo možné, a to skoro u všetkých, použiť systém PVM alebo MPI. Ale situácia sa zlepšila, keď som našiel grid

systém menom Alchemi. Ten umožňuje, aby bola požiadavka 3. a 4. splnená. Nebránil v splnení požiadaviek 2. a 5. – 7., ale v požiadavke 1. som musel vypustiť Linux. Požiadavku 4. som musel upraviť tak, že namiesto jazyka C sa použije jazyk buď C#, alebo managed extensions pre jazyk C++. Z toho je zrejmé, že Alchemi pracuje pod systémom .NET. Po niekoľkých dňoch som zistil, že autori systému Alchemi zabudli na takú jednoduchú úlohu, akou je možnosť vytvoriť nový proces procesom, ktorý je práve spustený. Preto som musel hľadať dalej, no nepodarilo sa mi nájsť iný grid systém, ktorý by spĺňal moje požiadavky.

Medzičasom som vyvýjal kompilátor pre zjednodušenú verziu jazyka PCN s názvom PCN Lite. Jeho detailný popis a špecifikácia je uvedený v časti Jazyk PCN Lite.

Keďže som nenašiel žiaden systém, ktorý by spĺňal požiadavku 3. a 4. a aj by príslušne fungoval, tak som sa rozhodol napísať vlastný. Potom som zistil, že to malo význam. Musel som však trochu zmeniť požiadavky na nasledovné:

1. použijem len operačný systém Windows;
2. systém musí byť použiteľný nielen na viacprocesorových počítačoch, ale aj v počítačovej sieti;
3. nepoužijem grid, ale vytvorím vlastnú knižnicu, ktorá mi vytvorí programové prostredie pre paralelné programovanie;
4. nepoužijem žiadny už existujúci paralelný systém ako napr. PVM, MPI;
5. povolím možnosť používať lubovoľný dátový typ pre definičné premenné;
6. povolím možnosť používať komunikáciu medzi procesmi pomocou správ;
7. musí byť jednoducho rozšíriteľný a požiteľný všade tam, kde bude potreba paralelne programovať s využitím počítačovej siete a použitím jazyka C/C++;

Pri tejto špecifikácii som už ostal a vytvoril som množinu funkcií, ktorú som nazval My Paralell Engine, skrátene MyPE.

Pošas vývoja som teda najprv vyvinul kompilátor pre jazyk PCN Lite, avšak po zadefinovaní požiadaviek pre MyPE som sa rozhodol zrušiť vývoj PCN Lite. Dôvod bol jednoduchý, MyPE dokáže to isté, čo PCN Lite, a nie je potrebné učiť sa PCN Lite. Navyše je MyPE navrhnutá flexibilnejšie, dá sa jednoducho rozširovať a prispôsobovať rôznym požiadavkám. To je tým, že pri vývoji PCN Lite som sa držal prvej špecifikácie, a preto prepisovať PCN Lite do MyPE už nemalo zmysel. To však neznamená, že PCN Lite je nepoužiteľný. Kompilátor pre tento jazyk funguje, len výsledný program nebude realizovateľný prostredníctvom MyPE. Preto pre získanie reálneho paralelizmu je potrebné mať viacjadrový procesor alebo viacprocesorový počítač s operačným systémom Windows XP.

Jazyk PCN Lite

Jazyk PCN Lite je moja úprava jazyka PCN. Mojím cieľom bolo odstrániť asi najväčšiu chybu jazyka PCN. Tá spočíva v nemožnosti deklarovať definičné premenné. Z pohľadu softwarového inžinierstva a matematiky je neprijateľné, aby sa pracovalo s objektom, ktorý sme nikde nedefinovali. Druhým cieľom bolo rozšíriť množinu typov pre definičné premenné, a to z pôvodných 6 (int, char, port, string, double, tuple) na ľubovoľný počet. Oba tieto ciele sa mi podarilo splniť. Toto rozhodnutie prinieslo aj nevýhody, ktoré bližšie popíšem v nasledujúcej časti.

Návrh kompilátora a jeho implementácia

Na vývoj kompilátora som použil generátor parsera a scanera. Jeho názov je COCO/R a mojou jedinou požiadavkou bolo, aby vygenerovaný zdrojový subor bol v jazyku C alebo C++ a aby bol skompilovateľný v prostredí Microsoft Visual Studio.

COCO/R generuje parser typu LL(k), kde k je prirodzené číslo. Štadardne je k = 1. Ak užívateľ potrebuje, môže vytvoriť metódu, ktorá prezrie nasledujúcich k tokenov, aby bolo možné urobiť príslušné rozhodnutie. Číslo k je, samozrejme, pevne dané.

PCN Lite sa kompiláciou preloží do jazyka C++. Tento program potom kompilujem v prostredí Microsoft Visual Studio. Pri návrhu vygenerovaného kódu som sa snažil držať ANSI normy pre C/C++, takže výsledný program by mal ísť skompilovať aj v iných ANSI C/C++ kompatibilných kompilátoroch.

Jazyk PCN Lite tak ako PCN používa paralelnú kompozíciu a definičné premenné. Tieto dva mechanizmy sú v tomto systéme najhlavnejšie a ja som ich implementoval nasledovne:

Paralelná kompozícia n blokov spôsobí vytvorenie n nových threadov. Definičná premenná používa mechanizmus operačného systému Windows – udalosti (events). Teda ak je premenná nedefinovaná, udalosť ešte nenastala a všetci, čo jej dáta potrebujú, čakajú, pokiaľ udalosť nie je definovaná. Po zadefinovaní nastane príslušná udalosť a čakajúce thready pokračujú v činnosti. Čakanie nestojí procesorový čas.

Základná syntax a dátové typy

Dátové typy, k**onštanty**, **reťazec, aritmetické výrazy, operátor, názvy premenných** sú definované tak, ako sú v pôvodnom PCN.

Komentáre nie sú implementované. Dôvod: chyba v generátore COCO/R, ktorá bola objavená v pokročilej fáze vývoja.

Procedúra je ako v pôvodnom PCN, naviac sa v časti *deklarácie premenných* uvádzajú aj deklarácie definičných premenných.

```
názov(formálne argumenty)
deklarácie premenných
blok
```

Funkcie v PCN Lite nie sú. Stačí však urobiť procedúru a použiť niektorý z argumentov ako návratovú hodnotu, čo nie je problém, pretože argumenty sa spracovávajú referenciou.

Oddeľovač je len čiarka (,).

Deklarácia premenných pozostáva z názvu typu a jedného alebo viac mien pre premenné. Ak má byť premenná pole, potom sa za jej meno pridá sufix [n], kde n je veľkosť poľa a n je celé číslo. Ak je formálny argument procedúry pole, tak sa použije deklarácia z jazyka C a za sa to pridá *.

Sekvenčná kompozícia

Sekvenčná kompozícia je implementovaná jednoducho a to použitím sekvenčnej kompozície programov v jazyku C/C++. Teda PCN Lite program:

```
001    f()
002    {;
003
004           g(),
005           h()
006
007    }
```

Ten sa preloží do C/C++ programu:

```
001   void f()
002   {
003
004   g();
005   h();
006
007   }
```

Sémantika programu sa po preklade nezmení, čo je z načrtnutej postupnosti zreteľné.

Podmienková kompozícia

PCN Lite program s podmienkami rovnakými ako pre jazyk PCN:

```
001        {? guard0 -> block0, ..., guardk -> blockk}
```

Preklad do C/C++ bude vyzerať nasledovne:

```
001         if (guard0)
002         {
003
004               block0
005
006         }
007         else
008         {
009
010               if (guard1)
011               {
012
013                     block1
014
015                     if (guard2)
016                     {
017
018                           …
019
020                           if (guardk)
021                           {
022
023                                 blockk
024
025                           }
026
027                     }
028
029               }
030
031         }
```

V pôvodnom PCN bol nedeterminizmus dovolený. Ja som ho zakázal. Tým je môj preklad zjednodušený a je jasné, že sémantika podmienkovej kompozície je s preloženým C/C++ kódom ekvivalentná.

Cyklus a rekurzia

Programátor musí manuálne inkrementovať premennú i. Syntax (ako v PCN, nič sa nezmenilo):

```
001      {op i over low .. high :: block}
```

Sémantika:

Príkaz *block* sa vykoná práve raz pre každé i z intervalu <*low, high*> sekvenčne, ak `op = ;` alebo sa bloky pre každé i vykonajú paralelne, ak `op = ||`.

Pri preklade musíme uvažovať o dvoch prípadoch: či je `op = ;`, alebo `op = ||`.

Prípad pre `op = ";"`:

```
001      for (int i = low; i < high; i++)
002            block
```

Preklad je intuitívny a jednoduchý. Vidíme, že sémantika je dodržaná.

Prípad pre `op = "||"`:

```
001      for (int i = low; i < high;)
002      {
003
004            vytvor thread pre block a spusť ho
005
006      }
```

Tu treba vysvetliť riadok 004, ktorý v sebe skrýva viacero akcií. Pred vytvorením a spustením threadu je potrebné tomuto threadu priradiť premenné, ktoré potrebuje, a až potom ho môžeme vytvoriť a spustiť.

Sémantika tohto prekladu je nasledovná:

Sekvenčne vytváram thready a hneď ich spúšťam. Teda thready sa vykonávajú paralelne, a tým je zachovaná sémantika pôvodného príkazu. Navyše príkaz cyklu nehovorí o tom, v akom poradí sa tieto thready majú vytvárať a následne spúštať (to nie je špecifikované ani v jazyku PCN).

Rekurzia je štandardne podporovaná v jazyku C/C++ a tak sa aj pri preklade používa. Teda PCN Lite program:

```
001          f()
002          {
003
```

```
004          f()
005
006          }
```

Ten sa preloží do:

```
001          void f()
002          {
003
004                f();
005
006          }
```

Sémantika oboch programov je ekvivalentná, čo je zreteľné z uvedeného programu.

Definičná premenná

Najprv si vysvetlíme, ako implementujem definičnú premennú.

```
001       template<typename type>
002       class DefVar
003       {
004
005             public:
006
007             HANDLE defined;
008             type val;
009
010             DefVar();
011             ~DefVar();
012
013       };
014
015       DefVar<typename type>::DefVar()
016       {
017
018             defined = CreateEvent(NULL, TRUE, FALSE, NULL);
019
020       }
021
022       ~DefVar()
023       {
024
025             CloseHandle(defined);
026
027       }
```

Definičná premenná je šablónová trieda jazyka C++. Šablóna mi zabezpečí, že ju môžem použiť pre ľubovoľný dátový typ. Premenná `defined` je udalosť, ktorá

oznamuje, či je premenná definovaná, alebo nie. Premenná *val* je hodnota samotnej premennej. Konštruktor zabezpečí, že po deklarácii je premenná nedefinovaná, teda nastáva situácia, ktorú sme potrebovali. To, že sa do nej dá len raz priradiť, však nekontrolujem. Ostáva to na programátorovi. Avšak tento návrh nie je úplne dokonalý. Jeho omnoho lepšiu verziu som vytvoril v knižnici MyPE, kde ich aj porovnám.

Sémantika definičnej premennej je ako v jazyku PCN. Operátor priradenia = je tiež ten istý. Avšak dovolujem ľubovoľný typ a ten sa musí deklarovať.

Syntax:

```
001       DefVar<typ> meno;
```

Sémantika:

Deklarácia premennej s názvom meno a typu typ. Na začiatku je nedefinovaná a po priradení sa zadefinuje.

Preklad je vo výslednom súbore rovnaký ako deklarácia, s výnimkou deklarácie poľa. Tam sa najprv deklaruje premenná a potom sa vytvorí priestor v pamäti pre toto pole.

Deklarácia poľa:

```
001          DefVar<typ*> meno[n];
```

Preklad:

```
001       DefVar<typ*> array;
002       array.val = new typ[n];
```

Syntax pre priradenie do premennej:

```
001       meno = hodnota;
```

Preklad:

```
001      meno.val = hodnota;
002      SetEvent(meno.defined);
```

Paralelná kompozícia

Nech P_1, ..., P_n sú bloky, n ≥ 0.

Syntax:

```
001      {|| P1, …, Pn}
```

Sémantika:

Spustí všetky bloky, aby sa vykonávali súčasne.

Implementácia je nasledovná:

```
001    {
002
003          Vytvor dáta pre thread 1
004          Spusť thread 1 a ako argument použi dáta vytvorené
005    v predchádzajúcom kroku
006
007          ...
008
009          Vytvor dáta pre thread n
010          Spusť thread n a ako argument použi dáta vytvorené v
011    predchádzajúcom kroku
012
013          Čakaj, pokiaľ sa všetky vytvorené thready neskončia
014
015    }
```

Vytvorenie dát vytvorí štruktúru, ktorá obsahuje všetky lokálne premenné a táto štruktúra sa potom použije ako argument pri spustení príslušného threadu. Riadok 013 spôsobí, že vykonanie paralelnej kompozície skončí, keď thready P_1, ..., P_n ukončia svoju činnosť. Potom program, ktorý použil paralelnú kompozíciu v sekvenčnej, pokračuje

ďalej vo výpočte. Preto je vhodné použiť paralelnú kompozíciu ako posledný príkaz, ak sa vyskytuje v sekvenčnej kompozícii.

Dátový typ tuple a zoznam

Typ tuple som neimplementoval z rôznych dôvodov. V prvom rade je to preto, lebo C/C++ je typový jazyk a pri práci s každým objektom musí byť kompilátoru jasné, akého je typu, čo sa v čase kompilácie nedá rozhodnúť. Riešením je obmedziť množinu prípustných typov, čo som však nechcel. Ďalším riešením je, aby programátor vo vygenerovanom kóde urobil pretypovanie ručne, ale to nie je veľmi pohodlné a pri väčších programoch by to bolo veľmi ťažko realizovateľné. Pre PCN Lite je však možnosť vytvoriť si ľubovoľný dátový typ. Preto nie je problém spraviť napríklad pole definičných premenných s premenlivou dĺžkou, aby tento problém riešil. Tým sa ale stráca pohodlná syntax jazyka PCN.

Taktiež nie je implementovaný operátor `?=`. Dá sa síce implementovať manuálne, ale tiež by sme nedosiahly pohodlnú syntax jazyka PCN. Hlavný problém však je, že v čase kompilácie nevieme rozhodnúť, akého typu bude definičná premenná.

Viacjazyčné programovanie

Pôvodné PCN vie spúšťať programy písané v C a vo Fortrane. PCN Lite vie len C. Dôvod je jasný, prekladám do C a ani som nemal v úmysle použiť Fortran. Ak by ale užívateľ chcel, mohol by použiť nejaký existujúci mechanizmus pre jazyk C na volanie programov písaných vo Fortrane (ak taký existuje).

Moduly

Používanie modulov je zabezpečené na úrovni jazyka C. V PCN Lite programátor nič nešpecifikuje. Musí to urobiť ručne vo vygenerovanom kóde.

Metavolania

Nie sú implementované. Ak však programátor potrebuje pracovať s funkciami, môže, tak urobiť pomocou ukazateľov na funkcie. Musí, to však urobiť vo vygenerovanom kóde.

Dátový typ port

Typ port nie je implementovaný. Dá sa ľahko implementovať ako uživateľský typ, nakoľko jeho sémantika je veľmi jednoduchá.

Príklad

Uveďme pre porovnanie jednoduchý príklad programu, ktorý sčíta pole čísel. Program som uviedol vo verzii pre PCN z (1, str. 21) a vo verzii pre PCN Lite. Na záver som uviedol časť vygenerovaného kódu v jazyku C.

Vo verzii PCN Lite je jasne vidieť nevýhoda typového jazyka C, ktorý ovplyvnil syntax PCN Lite najviac.

Verzia v jazyku PCN

```
001     sum_array(from, to, array, sum)
002     {?
003           from < to ->
004                 {||
005                       sum_array(from + 1, to, array, sumrest),
006                       sum = array[from] + sumrest
007                 },
008           from > to -> sum = 0
009     }
```

Verzia v jazyku PCN Lite

```
001     sumarray(DefVar<unsigned int> from, DefVar<unsigned int> to,
002     DefVar<unsigned int *> array, DefVar<unsigned int> sum)
```

```
003     DefVar<unsigned int> sumrest;
004     {?
005
006           from <= to -> {||
007
008                       sumarray(from +1, to, array, sumrest),
009                       sum = array[from] + sumrest
010
011                       },
012
013           from > to -> {; sum = 0}
014
015     }
```

Vygenerovaný kód (len pre procedúru sumarray)

```
001     void sumarray(DefVar<unsigned int> & from, DefVar<unsigned int>
002     & to, DefVar<unsigned int *> & array, DefVar<unsigned int> &
003     sum)
004     {
005
006           DefVar<unsigned int> sumrest;
007
008           HANDLE defVar_xxx_handles1[defVar_xxx_numHandles1];
009
010           defVar_xxx_handles1[0] = from.defined;
011           defVar_xxx_handles1[1] = to.defined;
012
013           WaitForMultipleObjects(defVar_xxx_numHandles1,
014           defVar_xxx_handles1, TRUE, INFINITE);
015
016           if(from <= to )
017           {
018
019                 HANDLE parFHandle2[parFnbConst2];
020
021                 parF2a parF2aa(from, to, array, sum, sumrest );
022                 parFHandle2[0] = CreateThread(NULL, 0,
023                 (LPTHREAD_START_ROUTINE)parF2, & parF2aa, 0, NULL);
024                 parF3a parF3aa(from, to, array, sum, sumrest);
025                 parFHandle2[1] = CreateThread(NULL, 0,
026                 LPTHREAD_START_ROUTINE)parF3, & parF3aa, 0, NULL);
027
028                 WaitForMultipleObjects(2, parFHandle2, TRUE,
029                 INFINITE);
030
031           }
032           else
033           {
034
035                 HANDLE defVar_xxx_handles5[defVar_xxx_numHandles5];
036
037                 defVar_xxx_handles5[0] = from.defined;
038                 defVar_xxx_handles5[1] = to.defined;
039
040                 WaitForMultipleObjects(defVar_xxx_numHandles5,
```

```
041                   defVar_xxx_handles5, TRUE, INFINITE);
042
043                   if(from > to )
044                   {
045
046                         sum = 0;
047                         SetEvent(sum.defined);
048
049                   }
050
051             }
052       }
```

pozn. Vo vygenerovanom programe sa vyskytujú rôzne zvláštne názvy premenných. Sú to dočasné premenné a ich názvy generuje kompilátor.

Zhrnutie

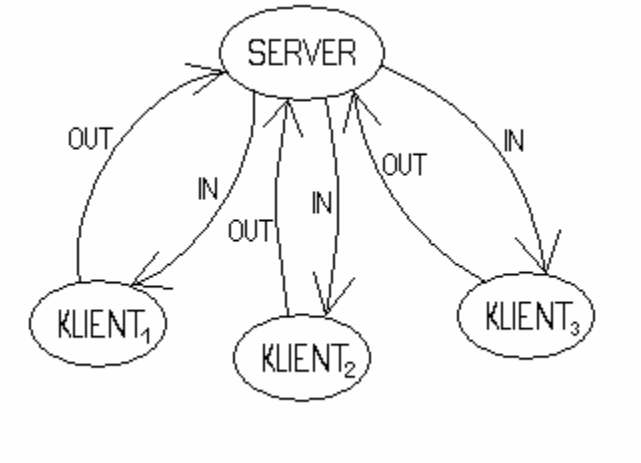
Jazyk PCN Lite som vyvíjal ešte v dobe prvej špecifikácie. Je funkčný, ale nepodporuje spúšťanie v počítačovej sieti. Preto je vhodné použiť ho len s viacjadrovým procesorom, ale na viacprocesorovom počítači s operačným systémom Windows. Som však spokojný, že som mal aspoň zjednodušenú verziu jazyka PCN, ktorú možno vyskúšať. Tu by som poznamenal, že riešenie prekladať do jazyka C síce umožnilo jednoducho znovupoužiť existujúci C program, ale prinieslo to mnohé problémy. Predovšetkým som musel vynechať štruktúru tuple, ktorá veľmi zjednodušuje programovanie v pôvodnom PCN. Preto na základe týchto skúseností by bolo lepšie použiť nejaký iný jazyk s rozhraním pre jazyk C a Fortran. Napríklad jazyk Haskell, ktorý je síce funkcionálny a mohlo by byť veľmi zaujímavé robiť preklad z PCN do Haskellu, ale problém s deklarovaním definičných premenných by to nevyriešilo. Haskell je totiž silne typový. Ak by sme chceli presnú kópiu jazyka PCN, stačilo by ho portovať z pôvodnej verzie. To by som však musel portovať pre operačný systém UNIX a nie Windows, ako som ja chcel. Najlepšie by však obstál jazyk Smalltalk, pretože je netypový a veľmi flexibilný, ale možno by bol problém s použitím existujúceho C a Fortran kódu. Na záver odporúčam PCN Lite len na základné zoznámenie sa so základnými (a tými najpodstatnejšími) mechanizmami, ktoré boli prezentované v PCN, a to definičné premenné a paralelná kompozícia.

Knižnica MyPE

MyPE je knižnica jazyka C, ktorá poskytuje dátové štruktúry a podporné funkcie pre paralelné programovanie v počítačovej sieti. Je jednoduchá, a preto ju v tejto časti kompletne predstavíme.

Architektúra

MyPE je postavená na architektúre klient/server. Server plní úlohu manažéra, ktorý spracováva užívateľsky definované funkcie. Tieto môžu byť napríklad preposielanie správ alebo rozhodovanie o spustení threadu na určitom počítači.



Obr. 2.

Z obrázku 1 sú architektúra a vzťah medzi klientom a serverov ľahko pochopiteľné. Teda medzi klientom sú vytvorené dve spojenia. Jedno označujeme ako IN a slúži na príjem správ a druhé OUT slúži na posielanie správ. Toto riešenie má svoj dôvod. Jediné spojenie by muselo zvládať prijímanie aj odosielanie správ. Príjem správ však vyžaduje čakanie na správu, buď pokiaľ správa nepríde, alebo čakanie určitú stanovenú dobu. To je však nepraktické, lebo by sa brzdilo odosielanie správ. Na realizáciu spojenia medzi počítačmi som použil technológiu Windows Socket 2 a protokol TCP.

Klient a server si medzi sebou vymieňajú správy, ktoré obsahujú príkaz a potrebné dáta na jeho spracovanie. Správa má nasledovný tvar:

Príkaz	Príjemca			Odosielateľ			Dáta
	Číslo		Adresa	Číslo		Adresa	
32 bit	32 bit	32 bit	64 bit	32 bit	32 bit	64 bit	

`Príkaz` je 32-bitová hodnota, dátový typ `int`, ktorý určuje číslo príkazu, ktorý sa má vykonať. Táto hodnota je povinná. Časť `Príjemca` a `Odosielateľ` majú rovnaký účel, označujú príjemcu a odosielateľa správy. Tieto hodnoty však nie sú povinné. Každý učastník má pridelené číslo, ktoré mu pridelí server. Toto číslo je 32-bitová hodnota, dátový typ `int`. Za ním nasleduje rezervovaná časť veľkosti 32 bitov. Hodnota `Adresa` je 64-bitový ukazateľ. Používa sa pri manipulácii s definičnými premennými a je to ich adresa v pamäti daného klienta. Časť `Dáta` je nepovinná časť a môže mať ľubovoľnú veľkosť. Je však na programátorovi, aby vedel s takýmto dátami pracovať. Správa musí mať minimálnu velkosť 36 bajtov, to znamená, že sa časti `Príkaz`, `Príjemca` a `Odosielateľ` musia uviesť. Stačí, len ak časť `Príjemca` a `Odosielateľ` budú vyplnené nulami.

V MyPE klient používa správy na získavanie hodnoty definičnej premennej, na definovanie definičnej premennej, na vytváranie threadov a správu pre ukončenie výpočtu.

Server

V tejto časti sa pozrieme na serverovú časť MyPE. Ukážem ako sú jednotlivé časti naprogramované a ako možno túto funkcionalitu rozširovať.

Server je obsiahnutý v súboroch peServer.cpp a peServer.h. Je programovaný pomocou konštrukcie namespace, t.j. priestor mien s názvom `PE_Server`. Obsahuje nasledujúce funkcie:

```
1. void regCommandFunc(int cmd, void (*f)(char*, int, int))
2. void initialize()
```

Tieto dve funkcie stačia pre prácu so serverom. Funkcia initialize sa používa na inicializáciu systému. Funkcia `regCommandFunc` sa používa na zaregistrovanie funkcie, ktorá bude spracovávať konkrétny príkaz, ak je to potrebné.

Inicializácia

Inicializácia je základ pre fungovanie servera. Počas inicializácie sa spustí nekonečný cyklus, v ktorom sa môžu klienti pripájať. Pripojenie klienta potom znamená vytvorenie threadu pre službu IN a threadu pre službu OUT. Tieto služby teda pracujú nezávisle od seba a komunikujú pomocou fronty správ, ku ktorej je priradená kritická sekcia kvôli synchronizácii. Funkcia `initialize` sa musí použiť ako posledná a skončí, až keď server skončí svoju činnosť.

Použitie:

```
PE_Server::initialize();
```

Registrácia funkcií pre uživateľsky definované príkazy

Funkcia `void regCommandFunc(int cmd, void (*f)(char *, int, int))` slúži na zaregistrovanie funkcie, ktorá spracuje prijatú správu obsahujúcu príkaz cmd. Druhý argument je ukazateľ na funkciu, ktorá tento príkaz spracuje. Jej argumenty sú nasledovné: najprv ide ukazateľ na samotnú správu, potom číslo odosielateľa a nakoniec dĺžka správy.

Použitie:

```
PE_Server::regCommandFunc(11, PE_Server::cmdF0011);
```

Tento príklad ukazuje, ako registrujem funkciu, ktorá spracováva príkaz číslo 11. Ten má za úlohu vytvoriť nový thread. Server však musí rozhodnúť o tom, na ktorom počítači sa tento thread spustí a na to je tu príslušná funkcia. Tá prijime správu a zistí, že treba vytvoriť nový thread. Rozhodne, kto ho vytvorí, a tomu správu pošle.

Server je teda veľmi jednoduchý a dá sa ľahko rozšíriť o nové funkcie, ktoré spracovávajú príslušné príkazy. Ak nejaký príkaz nemá funkciu, ktorá by ho spracovala, tak sa správa jednoducho prepošle príjemcovi, ktorý je v správe zaznamenaný.

Klient

Klient je oproti serveru omnoho zložitejší. Okrem inicializácie musí programátor ešte zaregistrovať dátové typy, s akými sa bude pracovať, a taktiež musí zaregistrovať funkcie, s ktorými bude systém MyPE pracovať paralelne. To sú práve tie funkcie, ktoré sa budú spúšťať na ostatných počítačoch.

Klient obsahuje nasledovné rozhranie:

```
void prihlasovanieFunkcii(int n, ...);
```

Táto funkcia slúži na príhlásenie funkcií, ktoré sa budú používať pri spúšťaní na iných počítačoch. Prvý argument je počet funkcií, ktorý chceme zaregistrovať. Druhý argument je zoznam ukazateľov funkcií, ktorý registrujeme.

```
void regTyp(int cislo, int byty, char * (*readF)(void *),
               void (*writeF)(void *, char *));
```

Táto funkcia registruje funkcie, ktoré zabezpečujú posielanie dát do počítačovej siete. Prvý argument je číslo typu. Druhý je počet bajtov, ktorými je reprezentovaný. Tretí je ukazateľ na funkciu, ktorá vezme hodnotu a spraví z nej dáta, ktoré sa pošlú cez počítačovú sieť. Posledný argument je ukazateľ na funkciu, ktorá prijaté dáta spracuje a priradí ich do príslušnej premennej. Jej prvý argument je tá premenná a druhý je časť správy, ktorá tieto dáta obsahuje.

```
void initialize();
```

Inicializácia klienta sa robí až po registrácii funkcií a typov. Po nej už nesmie nič nasledovať, pretože sa skončí, až keď skončí klient.

```
void setStart();
```

Povie klientovy, že je počiatok. Teda paralelný program začne ten klient, ktorý použije túto funkciu. Klienta, ktorý použije tento príkaz, nazveme počiatočným.

```
void setMain(void (*)(void));
```

Funkcia nastaví funkciu, ktorá sa má spustiť, ak klient je počiatočný.

PCN ideiy v MyPE

V predchádzajúcich častiach som uviedol iba minimum toho, čo treba na napísanie programu s využitím príkazov MyPE. Teraz ukážem, ako pracovať s definičnými premennými a ako vytvárať thready.

Najprv si uveďme, ako je definičná premenná definovaná v MyPE oproti PCN Lite.

```
001     struct rItem
002     {
003
004           void * e;
005           int t;
006           HANDLE defined;
007
008
009           bool definedB;
010
011           int owner;
012
013           void * ownerAddr;
014
015     };
```

Oproti definičnej premennej v PCN Lite máme o 4 premenné viac a na reprezentáciu už nepoužívam šablónovú triedu, ale štruktúru.

Prvou položkou je premenná `e` typu ukazateľ. Ten ukazuje na naše dáta. Druhá položka - typ dát, ktoré reprezentuje definičná premenná, a je číslo, ktoré sa prihlasuje pred inicializáciou klienta pomocou funkcie `regTyp`. Tretia premenná je udalosť, ktorá hovorí, či je premenná definovaná, alebo nie. Štvrtá premenná má ten istý význam ako tretia, ale je typu bool a je tam len z technických a historických dôvodov. Premenná `owner` je ako piata v poradí a je to identifikátor majiteľa tejto premennej. Posledná je ukazateľ na miesto v pamäti, kde je naša hodnota uložená.

Premenná `owner` je definovaná podľa nasledovného pravidla: ak sa vytvára na počítači A definičná premenná `x`, tak `x` má hodnotu owner rovnú identifikátoru A. Ak sa vytvára definičná premenná `x` na počítači B, ale pôvodná premenná je na počítači A, tak `x` na počítači B ma hodnotu `owner` nastavenú na identifikátor A. Keď použijem funkciu na prečítanie hodnoty `x`, tak MyPE bude vedieť, odkiaľ má túto hodnotu vziať.

Aby programátor nemusel ručne vytvárať definičné premenné a zaťažovať sa technickými detailmi, tak máme k dispozícii funkciu `CreateVar`, ktorá má nasledovný prototype:

```
rItem * CreateVar(int typ, void * data, bool def, rItem *
                    ownerAddr);
```

Prvý argument je typ premennej, druhý je ukazateľ na dáta. Tretí ak nadobúda hodnotu `true`, tak hodnota je už zadefinovaná a môže sa z nej čítať. Ak nadobúda hodnotu `false`, tak je hodnota nedefinovaná. Posledný argument je štandardne nastavený na `NULL`, takže ho netreba špecifikovať. Je určený na špecifikovanie adresy, kde je hodnota uložená. Ak teda vytvorím premennú na nejakom počítači a hodnota je na tomto istom počítači, tak túto hodnotu nešpecifikujeme, nastaví sa sama. V opačnom prípade ju musíme špecifikovať, ale na to je tu mechanizmus vytvárania threadov, ktorý túto úlohu za nás vyrieši. Ak by užívateľ chcel, mohol by túto hodnotu prijať od iného počítača tak, že si vytvorí správu s príkazom, ktorý toto zabezpečí.

Nasledovný príklad vysvetľuje použitie funkcie CreateVar.

```
int x = 0;
rItem * xx = CreateVar(1, &x, true);
```

V príklade predpokladám, že číslo `1` reprezentuje dátový typ `int`.

Ak chceme prečítať hodnotu definičnej premennej, tak sa najprv musíme uistiť, že je definovaná, a teda je aj k dispozícii. Na to slúži funkcia `ReadVar`, ktorá má nasledovný prototype:

```
void ReadVar(rItem * ri);
```

Jej úlohou je zabezpečiť, aby jej argument bol zadefinovaný. Ak nie je, tak čaká, kým sa definuje. Rieši všetky úlohy spojené s vyžiadaním hodnoty, jej prijatím, zadefinovaním a následným oznámením, že hodnota je definovaná. Uveďme si jednoduchý príklad:

Na počítači A nasledovný fragment programu vytvorí premennú `x` a definuje ju.

```
int x = 1;
rItem * xx = CreateVar(1, &x, true);
```

Na počítači B nasledovná časť programu premennú prečíta a použije:

```
ReadVar(yy);

int z = y + 1;
```

Vysvetlime si túto konštrukciu. Premenná `z` je nová a jej hodnota je `y + 1`. Premenná `y` je hodnota definičnej premennej `yy`. Je to niečo podobné, ako sme vytvorili `x` a `xx`. Keďže `xx` je na začiatku definovaná, tak `ReadVar(yy)` si túto hodnotu vyžiada a bude čakať, kým hodnota nepríde. Keď hodnota príde, tak sa zadefinuje a `ReadVar(yy)` skončí a vykoná sa priradenie do premennej `z`. Ak by sme pri vytváraní `xx` zmenili tretí argument funkcie `CreateVar` z `true` na `false`, tak by sme čakali, pokiaľ by ju nejaký iný thread nedefinoval.

Na priradenie hodnoty do definičnej premennej sa používa funkcia `WriteVar`. Jej argument je tak ako u `ReadVar` ukazateľ na definičnú premennú. `WriteVar` potom oznámi, že hodnota je definovaná, a tým sa spustia príslušné mechanizmy, ktoré aktivujú čakajúce thready. Ilustrujme si to na nasledujúcom príklade.

```
int x = 3;

WriteVar(xx);
```

V prvom príkaze sme definovali `x` na hodnotu `3`, v druhom sme oznámili, že premenná `xx` je definovaná. Pričom `xx` je definičná premenná pre `x`.

Teraz si ukážme, ako sa vytvára nový thread. Slúži na to funkcia `CreateThreadMessage`. Má nasledovný prototyp:

```
void CreateThreadMessage(void * f, int na, ...);
```

Prvý argument je ukazateľ na funkciu, ktorá sa má spustiť ako nový thread. Zároveň platí, že `f` je registrovaná pomocou funkcie `prihlasovanieFunkcii`. Druhý argument vyjadruje počet argumentov, ktoré za ním nasledujú. Argumenty sú potom ukazatele na definičné premenné. Uveďme si jednoduchý príklad.

```
rItem * xx;
rItem * yy;

CreateThreadMessage(f, 2, xx, yy);
```

V príklade som vytvoril nový thread, ktorý spustí funkciu `f` na niektorom klientovi s dvoma argumentami `xx` a `yy`.

Zhrnutie

MyPE je teda množina príkazov, s ktorou môžeme vytvárať paralelné programy. Je založená na klient/server architektúre a klienti počas výpočtu spracovávajú príkazy. Jednotlivé príkazy sú funkcie, ktoré môžu byť aj rekurzívne. Teda k takejto funkcii

môžeme priradiť Turingov stroj (alebo aj registrový stroj). Týchto funkcií máme k dispozícii konečný počet. Ich výpočet nám teda neuberie na sile a preto je pomocou MyPE možné naprogramovať ľubovoľnú rekurzívnu funkciu. MyPE v ničom neobmedzuje.

Príklad

V časti o jazyku PCN som uviedol príklad v ktorom program spočítal prvky poľa. Naprogramujme si to teraz pomocou MyPE. Tentokrát uvediem príklad kompletne so všetkými potrebnými časťami. V príklade je vidieť použitie všetkých funkcií, ktoré som opísal hore. Naviac je tam použitá funkcia `CreateVarEnd`, ktorá robí to isté ako `CreateVar` len s tým rozdielom, že definičná premenná, ktorú vytvorí, je uložená na inom počítači. Používa sa tak, že thread, v našom prípade reprezentovaný funciou sumarray, má 4 argumenty. Tie sa skrývajú v štruktúre `rItem ** rIa`, ako je uvedené v riadku 001. Jednotlivé argumenty z neho vyberiem pomocou `CreateVarEnd`, ako je to uvedené v príklade.

```
001     void sumarray(rItem ** rIa)
002     {
003
004           rItem * fr = CreateVarEnd(rIa[0]);
005           fr->e = new int;
006
007           rItem * t = CreateVarEnd(rIa[1]);
008           t->e = new int;
009
010           rItem * array = CreateVarEnd(rIa[2]);
011           array->e = new int[10];
012
013           rItem * sum = CreateVarEnd(rIa[3]);
014           sum->e = new int;
015
016           array->e = new int[10];
017
018           PE::ReadVar(fr);
019           int from = *((int*)fr->e);
020
021           PE::ReadVar(t);
022
023           int to = *((int*)t->e);
024
025           PE::ReadVar(array);
026
```

```
027           cout << "FROM " << from << " TO " << to << endl;
028
029           if (from < to)
030           {
031
032                 int from1 = from + 1;
033                 rItem * fr1 = CreateVar(2, &from1, true);
034
035                 int sumrest1;
036                 rItem * sumrest = CreateVar(2, &sumrest1, false);
037
038                 PE::createThreadMessage(sumarray, 4, fr1, t, array,
039     sumrest);
040
041                 PE::ReadVar(sumrest);
042                 *((int*)sum->e) = ((int *)(array->e))[from] +
043     *((int*)sumrest->e);
044                 PE::WriteVar(sum);
045
046           }
047           else
048           {
049
050                 *((int*)sum->e) = 0;
051                 PE::WriteVar(sum);
052
053           }
054
055     }
056
057     void test()
058     {
059
060           int t = 10;
061           rItem * to = CreateVar(2, &t, true);
062
063           int * ar = new int[t];
064           rItem * array = CreateVar(4, ar, true);
065
066           for (int i = 0; i < t; i++)
067           {
068
069                 ar[i] = i;
070
071           }
072
073           int fr = 0;
074           rItem * from = CreateVar(2, &fr, true);
075
076           int suma;
077           rItem * sum = CreateVar(2, &suma, false);
078
079           PE::createThreadMessage(sumarray, 4, from, to, array,
080     sum);
081
082           ReadVar(sum);
083
```

```
084             cout << "SUMA JE " << suma << endl;
085
086       }
087
088       void main(int args, char ** arg)
089       {
090
091             PE::prihlasovanieFunkcii(1, sumarray);
092
093             //zaregistrujem typ 2 = int
094             PE::regTyp(2, int_read, int_write);
095
096             //typ 4 = int array dlzky 10
097             PE::regTyp(4, int_array_read, int_array_write);
098
099             if(!strcmp(arg[1], "1"))
100             {
101                   cin.get();
102                   PE::setStart();
103
104             }
105
106             PE::setMain(test);
107
108             PE::initialize();
109
110       }
```

Záver

Mojou úlohou bolo vytvoriť systém pre paralelné programovanie, ktorý by myšlienkovo vychádzal z PCN. Aj napriek mnohým problémom, ktoré vývoj sprevádzali, predovšetkým neexistujúce grid systémy, ktoré by mohli spĺňať moju špecifikáciu, sa mi môj cieľ podarilo splniť.

Vytvoril som jazyk PCN Lite a naprogramoval k nemu kompilátor, preto si užívateľ operačného systému Windows môže vyskúšať aspoň základné prvky jazyka PCN. To však neznamená, že by PCN Lite bol menej schopný ako jazyk PCN. Chcel som urobiť preklad do jazyka C;;, kvôli čomu začala byť syntax jazyka PCN Lite komplikovanejšia ako v jazyku PCN, čo znemožnilo implementovať niektoré mechanizmy, ktoré by programovanie v PCN Lite zjednodušili. Na záver možno konštatovať, že PCN Lite vie to, čo PCN, ale PCN je oproti PCN Lite pomocou svojej

vyspelej syntax jednoduchý na použitie. Preto by som odporúčal použiť PCN Lite len pre jednoduché zoznámenie sa s jazykom PCN.

Ako som už spomínal, pri tvorbe PCN Lite som však narazil na rôzne problémy,. Preto som sa rozhodol vytvoriť zbierku príkazov, ktoré by umožňovali paralelné programovanie. Nazval som ju My Paralell Engine, skrátene MyPE. MyPE implementuje model systému PCN. Je to teda paralelná kompozícia s definičnými premennými, ktoré sú hlavnou ideou systému PCN. Ostatné časti PCN sú len vymoženosti, ktoré uľahčujú paralelné programovanie. Nie sú však zlé, naopak. Sú vynikajúco vytvorené a umožňujú programátorovi vytvárať elegantné zdrojové súbory, ktoré sú ľahko čitateľné a naviac robia to, čo chceme. Na rozdiel od MyPE, ktorého príkazy sú síce jednoduché, ale syntaktická krása jazyka PCN je znásilnená od momentu, ako som si za jazyk implemenácie zvolil jazyk C a C++.

Bibliografia

1. FOSTER, I. - TUECKE, S.: Parallel Programming with PCN. ANL-91/32, Rev. 2 : Argonne National Laboratory, 1993. 116 s.

2. FOSTER, I. - TUECKE, S.: A Portable Run-Time System for PCN. ANL/MCS-TM-137, Rev. 1 : Argonne national laboratory, 1991

3. Microsoft Corporation : Windows Socket 2. Windows Socket 2, [cit. 16. marec 2006; 15.03h GMT+01:00]. Dostupné na webovskej stránke (world wide web): http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp

4. Microsoft Corporation : DLLs, Processes and Threads. DLLs, Processes and Threads, [cit. 16. marec 2006; 15.09h GMT+01:00]. Dostupné na webovskej stránke (world wide web): http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/dllprocessthreads.asp

5. CHANDY, K. M. – TAYLOR, S.: A Primer for Program Composition Notation. Caltech-CS-TR-90-10 : California Institute of Technology, 1990

6. FOSTER, I. – TUECKE, S. – OLSON, R.: Productive Parallel Programming: The PCN Approach. Scientific Programming, Vol. 1, 1992. s. 51-66

7. FOSTER, I. – TAYLOR, S.: A Compiler Approach to Scalable Concurrent Program Design. Preprint MCS-P306-0492, Argonne National Laboratory, 1992

8. FOSTER, I. : Compositional Parallel Programming Languages. Preprint MCS-P354-0293, Argonne National Laboratory, 1996

9. FOSTER, I. - MICHALAKES, J.: MPMM: A Massively Parallel Mesoscale Model. Parallel Supercomputing in Atmospheric Sciences, World Scientific, Singapore, 1993, s. 354-363

10. RAVI, S. – ULLMAN, J. D. – AHO, A.: Compilers: Principles, Techniques and Tools. ISBN: 0201100886, Addison Wesley, 1986