Zaawansowane operacje grupowe

Zakres ćwiczenia

Celem ćwiczenia jest zapoznanie się z zaawansowanymi funkcjami biblioteki PVM wspierającymi tworzenie aplikacji rozproszonych.

Przedstawienie problemu

W tworzeniu wielu aplikacji napotyka się często na stałe i niezmienne elementy, wspólne dla wielu różnych programów. Do takich elementów należy:

- dzielenie wielkich struktur danych między wiele procesów
- zbieranie przetworzonych wielkich ilości danych
- Zbieranie i przetwarzanie wyników cząstkowych przetwarzania

Są to operacje, które każdy programista jest oczywiście w stanie samemu zaimplementować; jednakże z uwagi na częstość ich występowania biblioteka PVM zapewnia funkcje je realizujące, które pozwalają przyśpieszyć tworzenie nowych aplikacji. Poza tym dzięki stosowaniu tych funkcji zwiększa się także czytelność programu – łatwiej zrozumieć cel wywołania jednej dobrze opisanej funkcji niż przeanalizować kod zajmujący kilka linii.

Rozsyłanie i zbieranie danych

Wyobraźmy sobie problem, w którym należy przetworzyć wielką strukturę danych, przy czym na poszczególnych elementach (bądź bloki równej wielkości składające się z wielu elementów) można wykonywać operacje w sposób niezależny od operacji na innych elementach (blokach). Najprostszym i nieco abstrakcyjnym przykładem może być nieuporządkowany ciąg liczb, dla których należy sprawdzić, czy są one liczbami pierwszymi czy też nie. Jest to problem idealnie się nadający do zrównoleglenia: proces *master* powinien rozesłać równe bloki danych pomiędzy wiele procesów *slave*, z których każdy wykona na otrzymanych danych pewne obliczenia, a następnie master zbierze z powrotem przetworzone dane.

Te dwie funkcje: rozesłania elementów (tablicy) i następnie zebranie części tablicy z powrotem w jedną całość realizują dwie funkcje biblioteki PVM: `pvm_scatter` oraz `pvm_gather`.

Przygotowanie programu

Program, który utworzymy, będzie wyjątkowo prosty i będzie tylko demonstrował użycie funkcji `pvm_scatter` oraz `pvm_gather`. Przygotowana aplikacja będzie składać się z trzech plików: `def.h`, `master.c` oraz `slave.c`. Pliku `def.h` nie będziemy przedstawiać, gdyż jego budowa nie będzie odbiegać od analogicznych przykładów omawianych na poprzednich ćwiczeniach.

Proces *master* będzie rozsyłał tablicę składającą się z 22 elementów między 10 procesów typu *slave*. Każdy *slave* otrzyma więc dwa elementy tablicy, wykona na nich dowolną operację arytmetyczną, po czym proces *master* zbierze część tablicy z powrotem i wypisze je na standardowym wyjściu.

Użycie wymienionych funkcji `pvm_scatter` oraz `pvm_gather` wymaga, by procesy wcześniej stały się członkami tej samej grupy. Jeden z procesów staje się liderem przetwarzania (*korzeniem* przetwarzania, ang. *root*). To on inicjuje rozsyłanie i zbieranie wyników, ale sam również otrzymuje bloki danych i uczestniczy w ich przetwarzaniu.

Program master.c

```
1. #include "def.h"
2.
3.  int main()
4.  {
5.     int tids[10];
6.     int result[2];
7.     int i;
8.     int dane[22]={0,1,2,3, 4, 5, 6, 7, 8, 9,10,11,
                  12,13,14,15,16,17,18,19,20,21};

9.     pvm_joingroup( "GRUPA" );
10.    pvm_spawn( SLAVENAME,NULL, 0,".",10,tids);
11.    pvm_barrier( "GRUPA",11 );
12.    pvm_scatter( result, dane, 2, PVM_INT, MSG_MSTR, \
                                              "GRUPA", 0 );
13.    printf("RES %d %d\n", result[0], result[1]);
14.    result[0]++; result[1]++;
15.    pvm_gather( dane, result, 2, PVM_INT, MSG_SLV, \
                                               "GRUPA", 0);
16.    pvm_barrier( "GRUPA",11 );
17.    pvm_lvgroup( "GRUPA" );
18.    for ( i=0;i<22;i++)
19.    {
20.          printf( "Gather %d: %d\n", i, dane[i]);
21.    }
22.    pvm_exit();
23. }
```

Pierwsze jedenaście linijek nie będziemy omawiać. Jeżeli nie potrafisz odpowiedzieć na pytanie, co one robią, powinieneś ponownie przejrzeć poprzednie ćwiczenia. W linijce 12 pojawia się funkcja `pvm_scatter`. Jej argumentami po kolei są: rozsyłana tablica (tutaj: tablica `dane`), miejsce, do którego ma trafić część rozesłana tablicy (tutaj: tablica `result`), rozmiar części tablicy (tutaj: po dwa elementy na każdy proces). Następnie podajemy typ elementów tablicy (tutaj: `PVM_INT`, czyli czterobajtowa liczba całkowita), znacznik wiadomości używanej w rozsyłaniu danych (tutaj: `MSG_MSTR`), grupa użyta do rozsyłania („`GRUPA`”) oraz jaki proces ma być *korzeniem* przetwarzania. Proces wywołujący funkcję `pvm_scatter` sprawdza, jaki posiada w danej grupie numer instancji. Jeżeli taki sam, jak numer korzenia, to uznaje sam siebie za lidera i dokonuje rozesłania. Tutaj proces *master* posiada w grupie „`GRUPA`” numer instancji 0 (bo na pewno pierwszy do niej dołączył), taki sam numer jest wpisany jako ostatni parametr funkcji `pvm_scatter`, a więc to właśnie on ma być *korzeniem* przetwarzania i to właśnie on ma rozsyłać tablicę.

Proces *master* inkrementuje elementy otrzymanej przez siebie cząstki tablicy (linijka 14), po czym wywołuje funkcję `pvm_gather` (linijka 15), która jest odwrotnością funkcji `pvm_scatter`. Kolejne argumenty tej funkcji mówią, że wynik scalania ma być umieszczony w tablicy dane, wysyłane do korzenia przetwarzania będą elementy tablicy `result`, elementów tych jest dwa typu `PVM_INT`, znacznikiem wiadomości używanej do scalania jest `MSG_SLV`, używana jest grupa „`GRUPA`” a korzeniem przetwarzania jest proces, który w tej grupie posiada numer instancji 0.

Wreszcie proces *master* opuszcza grupę, wyświetla elementy całej tablicy po scaleniu i opuszcza środowisko PVM, kończąc następnie program.

Program slave.c

```
1. #include "def.h"

2.  int main()
3.  {
4.     int tids[10];
5.     int result[2];
6.     int dane[11]={0};

7.     pvm_joingroup( "GRUPA" );
8.     pvm_barrier( "GRUPA",11 );
9.     pvm_scatter( result, dane, 2, PVM_INT, 2, "GRUPA", 0 );
10.    printf( " Res: %d\n", result );
11.    result[0] += 10; result[1]+=10;
12.    pvm_gather( dane, result, 2, PVM_INT, 3, "GRUPA", 0);
13.    pvm_barrier( "GRUPA",11 );
14.    pvm_lvgroup( "GRUPA" );
15.    pvm_exit();
16. }
```

Budowa programu `slave.c` jest dość podobna. Proces *slave* wchodzi do grupy „GRUPA", a następnie wywołuje funkcję `pvm_scatter`. Należy zwrócić uwagę, że wywołanie to jest dokładnie identyczne jak wywołanie tej samej funkcji w linijce 12 w pliku `master.c`. Tutaj jednak numer instancji procesu w grupie „GRUPA" na pewno nie jest równy numerowi instancji korzenia przetwarzania (ostatni argument funkcji - zero). Proces ten więc wie, że jego zadaniem jest nie wysyłanie, ale **odbieranie** części rozsyłanej tablicy. Odebrany wynik umieszcza w tablicy `result`. Drugi argument tutaj nie ma żadnego znaczenia.

Po otrzymaniu danych, proces *slave* dodaje 10 do każdego elementu otrzymanej części tablicy i odsyła ją z powrotem do *mastera*. Jest to realizowane za pomocą funkcji `pvm_gather`. I znowu, wywołanie to jest dokładnie takie samo, jak odpowiadające mu wywołanie w linijce 16 pliku `master.c`. Proces wie jednak, że jego zadaniem jest wysyłanie *masterowi* swojej części tablicy, gdyż wie, że jego własny numer instancji **nie jest** równy zeru, a więc to nie on jest korzeniem przetwarzania. Pierwszy argument funkcji `pvm_gather` nie ma żadnego znaczenia dla procesu *slave*.

Wreszcie proces *slave* kończy opuszcza grupę, kończy pracę z środowiskiem PVM i wychodzi z programu.

Być może nie jest dla ciebie do końca jasne, do czego są potrzebne bariery w powyższych programach. Stanie się do bardziej oczywiste, gdy sobie uświadomisz, że nigdzie w podanych funkcjach nie podaje się liczby rozsyłanych i zbieranych bloków. Liczba ta zależy od rozmiaru grupy. Oczywiste jest więc, że aby przesyłanie zakończyło się sukcesem, liczność grupy musi być identyczna w momencie wywołania funkcji `pvm_scatter` i `pvm_gather` wszystkich procesów.

Ważną cechą przedstawionych dwóch funkcji właśnie jest to, że tak samo wygląda ich wywołanie zarówno u korzenia przetwarzania, jak i u pozostałych procesów. Mówi się, że te funkcje umożliwiają napisanie programów według paradygmatu SPMD (ang. *Single Program Multiple Data*), czyli identycznych programów działających na różnych fragmentach danych.

Należy teraz skompilować oba programy i umieścić pliki wykonywalne w odpowiednim katalogu:

```
gcc master.c –o master –lpvm3
gcc slave.c –o slave –lpvm3
cp master slave $PVM_HOME
```

Przykładowe uruchomienie stworzonej aplikacji może wyglądać tak (dla zwiększenia czytelności, zostały z niego usunięta część linijek, co zostało oznaczone wielokropkiem):

```
pvm> spawn -> master
[2]
1 successful
tc0037
pvm> [2:tc0037] libpvm [tc0037]: child task(s) still running.
waiting...
[2:tc0037] [tc0038] BEGIN
...
[2:tc0037] [t40055] BEGIN
[2:tc0037] RES 1 2
[2:tc0037] [t80038] EOF
[2:tc0037] [t80039] EOF
[2:tc0037] Gather 0: 2
[2:tc0037] Gather 1: 3
[2:tc0037] Gather 2: 13
[2:tc0037] Gather 3: 14
...
[2:tc0037] Gather 19: 30
[2:tc0037] Gather 20: 31
[2:tc0037] Gather 21: 32
[2:tc0037] [t100037] EOF
...
[2:tc0037] [t40055] EOF
[2:tc0037] EOF
[2] finished
```

Poznane funkcje biblioteki PVM

`int info = pvm_scatter(void *result, void *data, int count, int datatype, int msgtag, char *group, int rootginst)`

Funkcja `pvm_scatter` służy do rozesłania elementów tablicy pomiędzy wiele kooperujących procesów. Pierwszy argument określa lokalizację zmiennej, w którym będą umieszczane elementy. Zmienna ta musi mieć przydzieloną wystarczającą ilość pamięci, by pomieścić `count` elementów. Zmienna data określa oryginalną tablicę. Typ elementów jest określony przez `datatype`. Znacznik wiadomości używanej do rozsyłania określony jest przez `msgtag`. Rozsyłanie odbywa się w obrębie grupy określonej przez argument `group` a jego inicjatorem jest proces posiadający numer instancji `rootginst` w obrębie tej grupy. Dane `data` zawierać więc muszą count*<rozmiar grupy> elementów.

`int info = pvm_gather(void *result, void *data, int count, int datatype, int msgtag, char *group, int rootginst)`

Funkcja ta jest odwrotnością poprzedniej. W jej wyniku procesy należące do grupy `group` wysyłają bloki zawierające `count` elementów typu `datatype`, znajdujące się w argumencie `data`, to procesu określonego numerem instancji `rootginst`. Zmienna określona argumentem `result` musi mieć wystarczającą ilość miejsca by pomieścić `count`*<rozmiar grupy> elementów i ma sens tylko w procesie-korzeniu przetwarzania.

Operacja zebrania i przetworzenia (redukcji) wyniku

Kiedy przetwarzanie rozproszone się kończy, poszczególne procesy *slave* posiadają wyliczone wyniki cząstkowe, które należy następnie przesłać do procesu *master*, który prawdopodobnie dokona ich kombinacji przy pomocy pewnej operacji. Przykładem niech będzie omawiana na jednych z

poprzednich ćwiczeń obliczanie π metodą Monte-Carlo. Każdy z procesów *slave* losował pewną ilość punktów, sprawdzał, czy są one w obrębie koła, poczym przesyłał je do procesu nadrzędnego. Proces *master* otrzymane wyniki sumował i przetwarzał.

Jest to stały i często spotykany element obliczeń rozproszonych. Operację realizującą zebranie i przetworzenie wyników cząstkowych w celu uzyskania wyniku końcowego nazywana jest *redukcją*. Skoro operacja ta jest tak popularna, nie może dziwić fakt udostępnienia jej przez bibliotekę PVM w postaci funkcji `pvm_reduce`.

Przygotowanie programu

Napiszesz teraz nieskomplikowany program wykorzystujący operację redukcji. Celem będzie jedynie demonstracja użycia funkcji `pvm_reduce`, tak byś potrafił w przyszłości zastosować ją samodzielnie. Aplikacja składać się będzie z takich samych elementów jak poprzednio. Pliku `def.h` nie będziemy omawiać, gdyż jego budowa nie zmieni się w stosunku do poprzednich ćwiczeń.

W realizowanym przez nas zadaniu dokonamy posumowania numerów instancji w grupie wszystkich procesów. Dokonamy tego bez użycia gdziekolwiek (jawnie) operatora +, korzystając zamiast tego z wspomnianej przed chwilą funkcji `pvm_reduce`.

`Program master.c`

```
1. #include "def.h"
2.  int main()
3.  {
4.    int tids[SLAVENUM];
5.    int ginst=pvm_joingroup( "GRUPA" );
6.    pvm_spawn( SLAVENAME, NULL, 0,".",SLAVENUM, tids);
7.    pvm_barrier( "GRUPA", SLAVENUM + 1 );
8.    pvm_reduce( PvmSum, &ginst, 1, PVM_INT, MSG_MSTR, "GRUPA", 0);
9.    printf(" Suma wynosi %d\n", ginst );
10.   pvm_barrier( "GRUPA", SLAVENUM + 1 );
11.   pvm_exit();
12.  }
```

Omówimy teraz kod zawarty w pliku `master.c`. Proces *master* dołącza do grupy (linijka 5) tym samym ją tworząc, przed utworzeniem procesów potomnych (linijka 6). Gwarantuje to, że otrzyma numer instancji w grupie równy 0. Czeka, aż wszystkie procesy podrzędne dołączą do grupy (bariera w linijce 7) i wykonuje operację redukcji. Wywołując funkcję `pvm_reduce` zbiera wyniki od wszystkich członków grupy „GRUPA”, od każdego z nich oczekując na jeden element typu `PVM_INT`. Używane do tego wiadomości będą mieć znacznik MSG_MSTR. Wyniki mają być posumowane (pierwszy argument określa jak należy traktować – `PvmSum` jest jedną z kilku predefiniowanych operacji). Wynik operacji znajdzie się w zmiennej `ginst`. Ostatni argument ma takie same znaczenie jak poprzednio i określa numer instancji w grupie procesu będącego korzeniem przetwarzania. Ponieważ numer ten wynosi zero, i ponieważ numer instancji *mastera* w grupie „GRUPA” wynosi zero, proces *master* wie, że on sam jest korzeniem przetwarzania, i że to on właśnie ma zbierać i sumować wyniki.

Bariery zabezpieczają przed zmianą składu grupy przed zakończeniem operacji redukcji.

Przejdziemy teraz do analizy kodu procesów podrzędnych zawartego w pliku `slave.c`.

Program slave.c

```
1.  #include "def.h"
2.  int main()
3.  {
4.     int ginst=pvm_joingroup( "GRUPA" );
5.     pvm_barrier( "GRUPA", SLAVENUM + 1 );
6.     pvm_reduce( PvmSum, &ginst, 1, PVM_INT, MSG_MSTR, "GRUPA",0);
7.     pvm_barrier( "GRUPA", SLAVENUM + 1 );
8.     pvm_lvgroup( "GRUPA" );
9.     pvm_exit();
10. }
```

Proces *slave* dołącza do grupy „GRUPA” zapamiętując numer instancji w zmiennej `ginst`. Dwie bariery (linijki 5 i 7) otaczające wywołanie funkcji `pvm_reduce` zabezpieczają przed przedwczesnym opuszczeniem grupy. Wywołanie operacji redukcji ma postać zupełnie taką samą jak w linijce 8 w pliku `master.c`. Proces jednakże wie, że ma wysyłać komunikaty do *mastera*, gdyż jego numer instancji nie jest równy zero.

Po zakończeniu operacji redukcji wartość zmiennej `ginst` w procesach *slave* jest nieokreślona i najprawdopodobniej będzie się różniła od wartości przed wywołaniem funkcji `pvm_reduce`.

Programy należy skompilować i skopiować w odpowiednie miejsce w zwykły sposób. Wynik wywołania programu za pomocą komendy `spawn -> master` może wyglądać na przykład tak:

```
pvm> spawn -> master
[22]
1 successful
t40085
pvm> [22:t40085] libpvm [t40085]: child task(s) still running.
waiting...
[22:t40085] [t80062] BEGIN
[22:t40085] [tc0060] BEGIN
[22:t40085] [t100060] BEGIN
[22:t40085] [t140061] BEGIN
[22:t40085]  Suma wynosi 10
[22:t40085] [t80062] Moj ginst to 1
[22:t40085] [t80062] EOF
[22:t40085] [tc0060] Moj ginst to 2
[22:t40085] [t100060] Moj ginst to 4
[22:t40085] [t140061] Moj ginst to 3
[22:t40085] [tc0060] EOF
[22:t40085] [t100060] EOF
[22:t40085] [t140061] EOF
[22:t40085] EOF
[22] finished
```

Jako pierwszy argument funkcji `pvm_reduce` można podać adres swojej własnej funkcji. Powinna ona nie zwracać żadnej wartości oraz posiadać pięć argumentów, o znaczeniu kolejno: typ danych, sumowana wartość, wynik cząstkowy, liczba wyników, informacja zwrotna (o błędach). Poniżej podany jest przykład:

```
void my_num(int *datatype, void *x, void *y, int *num, int *info)
{
  *info = 1;
  printf( "%d %d %d\n", *datatype, *(int*)x, *(int*)y );
  *(int*)x += *(int*)y;
}
...
pvm_reduce( PvmSum, &ginst, 1, PVM_INT, MSG_MSTR, "GRUPA",0);
...
```

Poznane funkcje biblioteki PVM

```
int info = pvm_reduce( void (*func)(),  void *data, int count,
            int datatype, int msgtag, char *group, int rootginst)
```

Funkcja ta realizuje operację redukcji. Procesy należące do grupy `group` przesyłają do procesu o numerze instancji określonego przez `rootginst` swoje wyniki cząstkowe `data`. Liczba tych wyników i ich typ określają argumenty `count` i `datatype`. Argument `msgtag` określa typ wiadomości używany przez operację. Na wynikach dokonywana jest operacja wskazana przez `func`, która może być jedną z predefiniowanych stałych:

- `PvmMax`, `PvmMin` – wynik będzie maksimum (minimum) wyników cząstkowych (uwaga: jeżeli procesy przesyłają po dwa wyniki cząstkowe, w wyniku otrzyma się *dwa* maksima)
- `PvmSum` – wynik jest sumą (sumami) wyników cząstkowych.
- `PvmProduct` – wynik jest produktem wyników cząstkowych.

Zadanie do samodzielnego wykonania

Jako przykład problemu, w którym występuje operacja redukcji, podaliśmy metodę Monte-Carlo obliczania liczby π. Twoim zadaniem obecnie jest zmodyfikowanie wcześniej napisanego kodu tak, by używał funkcji `pvm_reduce`. Rozwiązanie znajduje się w materiałach kursu.

Podsumowanie

W wyniku ćwiczeń poznałeś i nauczyłeś się stosować trzy funkcje realizujące zaawansowane operacje realizujące często spotykane fragmenty przetwarzania, przydatne w tworzeniu rozproszonych aplikacji.

Co powinieneś wiedzieć:

- Jakiej funkcji można użyć do rozesłania (`pvm_scatter`) i zebrania (`pvm_gather`) dużej ilości danych między wiele procesów.
- Jakiej funkcji można użyć do zebrania i przetworzenia wyników cząstkowych, efektów pracy wielu kooperujących procesów (`pvm_reduce`)