STUDIA INFORMATICA 2000
Volume 21 Number 1 (39)

Aleksandra WERNER
Politechnika Śląska, Instytut Informatyki

ALGORYTMY OBLICZEŃ RÓWNOLEGŁYCH DLA ZADAŃ ALGEBRY LINIOWEJ W ARCHITEKTURZE WEKTOROWO-WIELOPROCESOROWEJ

Streszczenie. W artykule przedstawiono charakterystykę superkomputerów o architekturze wektorowej oraz przeanalizowano wybrane operacje algebry liniowej, porównując sposób i szybkość realizacji zadania przy przetwarzaniu wektorowym i wieloprocesorowym. Zaprezentowano również wady i zalety poszczególnych rozwiązań.

PARALLEL ALGORITHMS FOR LINEAR ALGEBRA IN VECTOR-MULTIPROCESSOR ARCHITECTURE

Summary. Characteristics of vector supercomputers and selected linear algebra operations analysed by comparing the way and speed of execution in vector and multiprocessor processing, are presented in this paper. Additionally, the merits and drawbacks of described approaches are also discussed in this article.

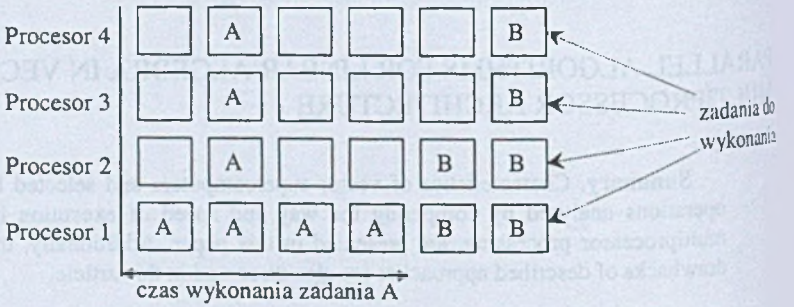
1. Wprowadzenie

W chwili obecnej, w związku z dominacją obliczeń dużego stopnia złożoności, jednym z najważniejszych kierunków rozwoju nauki i techniki jest podniesienie wydajności obliczeń. W celu sprostania tym wymogom zarówno programowe, jak i sprzętowe rozwiązania bazują na technikach przetwarzania równoległego: wieloprocesorowości, wektoryzacji i potokowości.

Pierwsze maszyny wieloprocesorowe zwiększały ogólną wydajność systemu przez zastosowanie tzw. tablic procesorów (ang. processor arrays), będących grupą procesorów, z których każdy miał określone możliwości (rys.1).

Rys.1. Realizacja zadań z wykorzystaniem tablicy procesorów
Fig. 1. Tasks realization with utilization of processor arrays

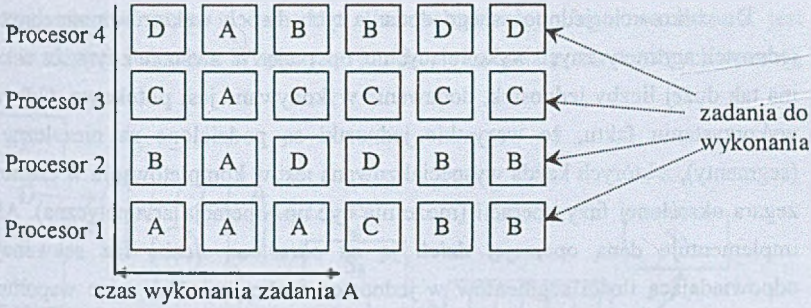
Podejście to pozwalało na wykonanie większej liczby zadań w określonym przedziale czasu, ale nie skracało czasu potrzebnego do uzyskania wyniku pojedynczego zadania. Wymóg skrócenia czasu realizacji pojedynczego zadania zaowocował pojawieniem się koncepcji zbioru procesorów, wykonujących identyczny ciąg instrukcji na różnych strumieniach danych, pozwalającej na wykonanie pojedynczego zadania na wielu procesorach, co znacznie zredukowało czas potrzebny do jego rozwiązania (rys. 2).



Rys. 2. Czas realizacji pojedynczego zadania przez grupę procesorów
Fig. 2. Time of a single task realization by the group of processors

Kolejnym krokiem w kierunku zwiększenia mocy obliczeniowej komputerów było wbudowanie mechanizmu ASAP[1] (ang. Automatic Self Allocating Processors), pozwalającego na jednoczesne zastosowanie równoległości i wieloprocesorowości (rys. 3). Osiągnięty rezultat jest taki, że żaden procesor nie znajdzie się w stanie jałowym, dopóki jest jakieś zadanie do wykonania (wykorzystanie CPU = 100%).

[1] Mechanizm ten zastosowano m. in. w systemach serii CONVEX C.



Rys. 3. Ilustracja zasady działania mechanizmu ASAP
Fig. 3. Illustration of principle of a ASAP mechanism

Zastosowanie mechanizmu ASAP w superkomputerach wektorowych miało dwa cele:

1. rozłożyć obciążenie (proces, który zajmuje 10[s] czasu jednostki centralnej CPU będzie wykonany w czasie o połowę krótszym, jeżeli praca zostanie równomiernie rozłożona np. na dwie CPU),
2. umożliwić uruchamianie programów wykorzystujących wielowątkowość na jednym lub wielu procesorach, bez jakiejkolwiek programowej modyfikacji.

2. Wady i zalety komputerów wektorowych

Superkomputery wektorowe charakteryzują się dużą szybkością obliczeń oraz dużą pamięcią podstawową i pomocniczą. Są to komputery o zwiększonej niezawodności (ich systemy operacyjne posiadają wielopoziomową strukturę bezpieczeństwa), którą dodatkowo można podnieść stosując obliczenia nadmiarowe[1].

Praca procesora wektorowego polega na wykonywaniu operacji na całych wektorach praktycznie równocześnie. Przykładem może być dodawanie dwóch wektorów $v_k = v_j + v_i$, gdzie poszczególne elementy wektora są dodawane niezależnie od siebie:

$v_k[1] = v_j[1] + v_i[1]$

$v_k[2] = v_j[2] + v_i[2]$

...

$v_k[128] = v_j[128] + v_i[128]$

[1] Jest to realizowane sprzętowo przez zastosowanie wielu procesorów, mogących przejąć realizację obliczenia, którego nie dokończył inny procesor.

Do całkowicie jednoczesnego dodania tych dwóch wektorów potrzebnych byłoby 128 jednostek arytmetycznych wykonujących tę operację. W związku z tym, że w komputerze nie ma tak dużej liczby jednostek, dodawanie wykonywane jest **potokowo**. Odbywa się to przy wykorzystaniu faktu, że wszystkie jednostki są podzielone na niezależne podjednostki (segmenty), z których każda wyspecjalizowana jest w kompletowaniu w czasie jednego taktu zegara określonej fazy operacji (może nią być np. operacja arytmetyczna). Algorytm, który implementuje daną operację, dzieli ją na określoną liczbę faz sekwencyjnych, ściśle odpowiadającą ilości segmentów w jednostce funkcyjnej. Tak więc wspomniana operacja dodawania składa się z kilku kroków wykonywanych przez kolejne części jednostki arytmetycznej. Kolejne liczby z rejestrów wektorowych[1] są pobierane do pierwszej podjednostki po jej opuszczeniu przez poprzednią liczbę, nie czekając na dokończenie operacji dodawania na tej liczbie. Czas między pobraniem kolejnych elementów wektora jest równy czasowi najdłuższego kroku elementarnego tej operacji.

Pewnym ograniczeniem jest niewątpliwie fakt, że rejestry wektorowe superkomputerów wektorowych mają ograniczoną pojemność (np. rejestry komputera CONVEX C3820 mogą zmieścić maksymalnie 128 elementów). Jeżeli chcemy wykonać operacje na tablicy o większej liczbie elementów, wówczas następuje podział tej tablicy na części po 128 (w przypadku C3820) elementów. Operacje wektorowe są wykonywane po kolei na tych częściach. Jest to tzw. **strip mining**.

Większość jednostek funkcyjnych jest całkowicie niezależna. Oznacza to, że mogą one równolegle operować na niezależnych zbiorach danych. Rozważmy sytuację, w której wynik pierwszej operacji (wektor v_k) jest argumentem drugiej operacji:

$$v_k = v_i + v_j$$

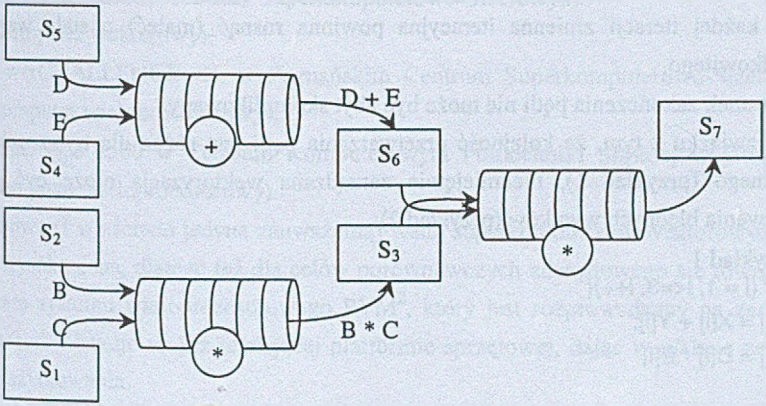
$$v_m = v_k * v_l$$

Aby rozpocząć mnożenie, wystarczy otrzymać pierwszy element wektora v_k. Operacja druga rozpoczyna się w momencie obliczenia tego elementu przez jednostkę dodającą i wykonywana jest potokowo aż do wymnożenia ostatniego otrzymanego elementu wektora v_k:

	pojawiające się elementy wyniku:
czas t_0:	$v_k[1]$
czas $2t_0$:	$v_k[2]$, $v_m[1]$
czas $3t_0$:	$v_k[3]$, $v_m[2]$
czas $4t_0$	$v_k[4]$, $v_m[3]$
...	

[1] Elementy tablic, po spełnieniu innych warunków, są ładowane z pamięci do rejestrów wektorowych.

Jest to tzw. przetwarzanie w łańcuchu (**łańcuchowanie**). Jego warunkiem jest wykorzystywanie różnych jednostek arytmetycznych do wykonania poszczególnych operacji (rys. 4).



Rys. 4. Schemat realizacji operacji podstawienia: A = B * C * (D + E)
Fig. 4. Schematic diagram for the computation of: A = B * C * (D + E)

Za swego rodzaju wadę komputerów wektorowych można uważać konieczność znajomości przez programistę warunków wektoryzacji programów (czego konsekwencją jest potrzeba stosowania w programach dyrektyw dla kompilatora).

2.1. Warunki wektoryzacji

Komputery wektorowe wyposażone są w autowektoryzujące kompilatory dostarczane przez producenta. Oznacza to, że użytkownik nie musi znać architektury komputera i programować operacji wektorowych. Może programować w tradycyjnym C lub Fortranie, a kompilator sam wygeneruje kod, uwzględniający możliwości procesora. W praktyce jednak, aby w pełni wykorzystać możliwości tkwiące w procesorze, potrzebna jest wiedza o technikach stosowanych przez kompilator przy wektoryzacji programu.

Automatyczna optymalizacja obejmuje pętle: FOR oraz WHILE.

By optymalizacja mogła być wykonana, pętle muszą być pisane z uwzględnieniem pewnych założeń:

- nie mogą zawierać instrukcji GOTO ani wielu wejść lub wyjść do/z pętli,
- nie mogą zawierać wywołań funkcji,
- nie mogą zawierać rekurencyjnych operacji na tablicach,

- liczba iteracji pętli musi być znana przed rozpoczęciem jej wykonywania, a warunek zakończenia nie może być ustalany na podstawie innych warunków, których nie można przewidzieć przed rozpoczęciem wykonania pętli,
- w każdej iteracji zmienna iteracyjna powinna rosnąć (maleć) o stałą wartość typu całkowitego,
- warunek zakończenia pętli nie może być zbyt skomplikowany.

W związku z tym, że kolejność przetwarzania pętli jest różna dla trybu wektorowego i skalarnego (przykład 1), nieumiejętnie zarządzana wektoryzacja może być przyczyną powstawania błędnych wyników (przykład 2).

- Przykład 1

```
for (i = 1, i<=3, i++){
Z[i] = X[i] + Y[i];
F[i] = D[i] * E[i]
}
```

kolejność wykonywania operacji:

tryb skalarny:	tryb wektorowy:
Z[1] = X[1] + Y[1]	Z[1] = X[1] + Y[1]
F[1] = D[1] * E[1]	Z[2] = X[2] + Y[2]
Z[2] = X[2] + Y[2]	Z[3] = X[3] + Y[3]
F[2] = D[2] * E[2]	F[1] = D[1] * E[1]
Z[3] = X[3] + Y[3]	F[2] = D[2] * E[2]
F[3] = D[3] * E[3]	F[3] = D[3] * E[3]

- Przykład 2

```
For (i = 1, i<=2, i++){
X[i] = Z[i];
Y[i] = X[i + 1]
}
```

kolejność wykonywania operacji:

tryb skalarny:	tryb wektorowy:
X[1] = Z[1]	X[1] = Z[1]
Y[1] = X[2]	X[2] = Z[2]
X[2] = Z[2]	Y[1] = X[2]
Y[2] = X[3]	Y[2] = X[3]

Operacja wystąpiła wcześniej niż podstawienie: Y[1] = X[2]

3. Przebieg i wyniki przeprowadzonych badań

Przykładowe programy napisano w jezyku C, a obliczenia przeprowadzono na grupie stacji roboczych SUN oraz na superkomputerach:

- CONVEX C3820 w Akademickim Centrum Komputerowym CYFRONET w Krakowie (superkomputer wektorowy zaliczany do klasy MIMD w klasyfikacji wg M. Flynna[1]),
- Cray J90 w Poznańskim Centrum Superkomputerowo-Sieciowym (superkomputer wektorowy),
- SGI PowerCHALLENGE XL w Poznańskim Centrum Superkomputerowo-Sieciowym (superkomputer wieloprocesorowy),
- SUN Enterprise 6500 w Centrum Komputerowym Politechniki Śląskiej w Gliwicach (superkomputer wieloprocesorowy).

Podstawową (i właściwie jedyną zauważalną) wadą superkomputerów wektorowych jest ich bardzo wysoka cena, dlatego też dla celów porównawczych zdecydowano się również na wykorzystanie systemu wieloprocesorowego PVM[2], który jest rozprowadzany na zasadach „public domain" i bazuje na już istniejącej platformie sprzętowej, dając w efekcie zerowe[3] koszty jego użytkowania.

3.1. Sposób przeprowadzenia testów

By móc zmierzyć czasy wykonania całego programu i poszczególnych jego części na superkomputerach, użyto specjalnego narzędzia programowego – profilera. Narzędzie to dostarczyło informacji o liczbie wywołań podprogramów, pętlach, częściach programu, które były wykonywane równolegle i pozwoliło znaleźć te sekwencje instrukcji, które wykonywały się najdłużej lub najczęściej (tabela 1).

Tabela 1

Komendy wykorzystywane przy kompilacji programów

Komputer	Komenda kompilująca program	Narzędzie do śledzenia czasu wykonania zadania
Convex C3820	cc – p – O2 *nazwa_programu*.c	prof
Cray J90	cc Gp – O2 – l prof *nazwa_programu*.c	profview –rt *zbiór.prof*
SGI Challenge XL	cc –p *nazwa_programu*.c	prof
SUN Enterprise 6500	cc –p *nazwa_programu*.c	workshop
PVM	cc *nazwa_programu*.c –lpvm3	xpvm

[1] Jest to klasyfikacja systemów równoległych, której podstawą jest liczba strumieni: danych i rozkazów, współpracujących z danym systemem komputerowym.

[2] W systemie tym komunikacja między procesorami odbywa się za pośrednictwem komunikatów przesyłanych przez sieć połączeń.

[3] Pomijając koszt ściągnięcia odpowiedniej wersji systemu z sieci Internet.

3.2. Zestawienie porównawcze czasów wykonania zadań

3.2.1. Iloczyn skalarny dwóch wektorów

Iloczyn skalarny dwóch wektorów jest sumą iloczynów ich składowych. W programie obliczany jest iloczyn skalarny wektorów: wekta oraz wektb, przy czym ilość składowych tych wektorów wahała się w granicach od 100 do 1500 elementów.

- Wersja sekwencyjna programu

```
#include <stdio.h>
#define DL 100

main()
{
int i,dl = DL;
float wekta[DL],wektb[DL];
int iloczyn =0;

while (dl--){
        wekta[dl] = dl;
        wektb[dl] = dl;
}
dl = DL;
for (i=0; i<dl;i++){
iloczyn+=wekta[i] * wektb[i];
}
printf("iloczyn skalarny = %dn", iloczyn);
}
```

- Rozwiązanie przy użyciu systemu PVM (wersja równoległa)

```
#include <stdio.h>
#include "pvm3.h"

#define ILDANYCH 100                    /* długość wektorów                  */
#define ILKOMP     5                    /* ilość procesorów                  */

main(argc, argv)
int argc;
char **argv;
{
        if (pvm_parent() == PvmNoParent)
                glowny (argv[0]);
        else
                potomny ();
}

glowny (nazwa)
char *nazwa;
{
  int wekta[ILDANYCH], wektb[ILDANYCH];   /* deklaracja wektorów               */

  int i, wynik;                           /* wynik częśc., uzyskany z 1 procesora */
```

```
int iloczyn = 0;                                  /* wynik końcowy = iloczyn skalarny   */
                                                  /* zadeklarowanych wektorów           */

int iledanych=ILDANYCH/ILKOMP;                    /* ilość danych dla jednego procesora */

int tid[ILKOMP];                                  /*tablica identyfikatorów zadań  */

pvm_spawn(nazwa, 0, PvmTaskDefault, "", ILKOMP, tid);

for (i=0; i<ILKOMP; i++) {                        /* przesłanie danych pocz. do procesów */
   pvm_initsend (PvmDataDefault);                 /* potomnych                           */
   pvm_pkint (&wekta[i*iledanych], iledanych, 1);
   pvm_pkint (&wektb[i*iledanych], iledanych, 1);
   pvm_send (tid[i], 100);
}

for (i=0; i<ILKOMP; i++) {                        /* odczyt danych z poszczeg. procesów */
   pvm_recv(-1, 200);
   pvm_upkint (&wynik, 1, 1);
   iloczyn += wynik;
}

printf ("Iloczyn skalarny = %d\n", iloczyn);

pvm_exit();                                       /* opuszczenie systemu PVM            */
exit(0);
}

potomny()
{
int sklada[ILDANYCH/ILKOMP];                      /* deklaracja wektorów składowych     */
int skladb[ILDANYCH/ILKOMP];
int masterid, i, wynik = 0;

masterid=pvm_parent();
pvm_recv(-1, 100);
pvm_upkint(sklada, ILDANYCH/ILKOMP, 1);           /* pobranie danych z bufora           */
pvm_upkint(skladb, ILDANYCH/ILKOMP, 1);

for (i=0; i < ILDANYCH/ILKOMP; i++)
      wynik += sklada[i] * skladb[i];

pvm_initsend(PvmDataDefault);                     /* zwrócenie częściowych wyników do   */
pvm_pkint(&wynik, 1, 1);                          /* procesu nadrzędnego                 */
pvm_send(masterid, 200);

pvm_exit();
exit(0);
}
```

Porównanie szybkości działania programu w wersji sekwencyjnej i równoległej oraz przy wykorzystaniu komputerów wieloprocesorowych przedstawiono odpowiednio w tabelach 2 i 3.

Tabela 2

Szybkość realizacji zadania w komputerach wieloprocesorowych i stacji SUN SPARCclassic1

Komputery obliczające iloczyn skalarny dwóch wektorów	Długość wektora	
	100 elementów	100000 elementów
SGI PowerCHALLENGE XL	2 [ms]	89 [ms]
SUN Enterprise 6500	1 [ms]	45 [ms]
SPARCclassic1	143 [ms]	2160 [ms]

Tabela 3

Czas wykonania zadania w wersji sekwencyjnej i równoległej

Konfiguracja maszyny wirtualnej	Czas wykonania zadania obliczania iloczynu skalarnego wektorów o długości 100000 elementów	
	wersja sekwencyjna	rozwiązanie w PVM
SPARCstation 10	2805 [ms]	436 [ms]
SPARCstation 10 SPARCstation IPX		270 [ms]
SPARCstation10 SPARCstation IPX SPARCstation Ultra SPARCclassic1 SPARCclassic2		81 [ms]

W związku z tym, że kompilatory komputerów wektorowych umożliwiają wybór optymalizacji: skalarnej (opcja –O1) bądź wektorowej (opcja –O2), przeprowadzono wiele testów, dla których otrzymano wyniki wyszczególnione w tabeli 4.

Tabela 4

Wykaz czasów wykonania zadania w komputerach wektorowych dla różnych długości wektorów składowych

Stacje liczące iloczyn skalarny wektorów	Długość wektora [elementy]									
	100		200		300		500		1500	
	opcja O1[ms]	opcja O2[ms]	opcja O1[ms]	opcja O2[ms]	opcja O1[ms]	opcja O2[ms]	opcja O1[ms]	opcja O2[ms]	opcja O1[ms]	opcja O2[ms]
CONVEX C3820	0,6204	0,6178	0,6832	0,6603	0,6991	0,6998	0,7301	0,6918	0,9364	0,9144
Cray J90	0,3603	0,3802	0,3591	0,3668	0,3598	0,3685	0,3769	0,3774	0,5383	0,5030

Analiza programów asemblerowych ujawniła, że nie bez znaczenia był również przyjmowany w programie typ składowych wektorów. Przy zmianie typu na double wykorzystywany był dodatkowy rejestr wektorowy VS, określający krok pobierania wartości

z pamięci do rejestrów wektorowych. Nie miało to jednak wpływu na uzyskane wyniki czasowe.

Obserwowana prawie dwukrotnie dłuższa realizacja obliczeń w komputerze C3820 w porównaniu do szybkości obliczeń wykonywanych w komputerze Cray jest spowodowana liczbą procesorów znajdującą się w obu wymienionych superkomputerach (CONVEX składa się z 4 procesorów, Cray J90 – z 8). Analizując otrzymane rezultaty, daje się też wyraźnie zauważyć, że czas wykonania zadania skompilowanego z opcją skalarną przyrasta w przybliżeniu w sposób liniowy (w przeciwieństwie do czasu wykonania zadania skompilowanego z opcją O2, co jest spowodowane efektem strip mining).

3.2.2. *Mnożenie dwóch wektorów*

Program został włączony do wykonywanych eksperymentów w celu zaobserwowania opisywanego wcześniej efektu **strip mining** (por. pkt. 2). Żeby odnotować jakiekolwiek nakłady czasowe przy wykonaniu zadania w komputerach wektorowych, funkcję mnożącą wektory: wekta oraz wektb „sztucznie" obudowano pętlą wykonującą się 10000 razy (stała ILE_RAZY).

```
#include <stdio.h>
#define DL 110
#define ILE_RAZY 10000
float wekta[DL];
float wektb[DL];
float wektc[DL];

void mno ()
{ int i;
  for ( i = 0; i < DL; i++ ) {
      wektc[i] = wekta[i] * wektb[i]; }
}
main ()
{ int j;
  for ( j = 0; j < ILE_RAZY; j++) {
      mno() }
}
```

Uzyskane wyniki czasowe przedstawiono w tabeli 5.

Tabela 5

Porównanie czasu realizacji zadania w komputerze Cray J90 i w komputerach o architekturze wieloprocesorowej

Superkomputery wykonujące obliczenia		Długość wektorów składowych		
		110	120	130
Cray J90	opcja O1	8,956[ms]	33,588[ms]	14,819[ms]
	opcja O2	28,391[ms]	26,786[ms]	18,702[ms]
SGI PowerCHALLENGE XL		247[ms]	266 [ms]	287[ms]
SUN Enterprise 6500		107[ms]	117[ms]	125[ms]

3.2.3. *Filtr*

Program napisano z myślą o sprawdzeniu wydajności superkomputerów wektorowyc przy dużej ilości obliczeń. Zgodnie z oczekiwaniami program został najszybciej wykonany po skompilowaniu z opcją optymalizacji wektorowej na superkomputerze Cray J90 (tabela 6).

```
#include <stdio.h>
#define ILE_RAZY 1000

int vect_we[322][202],vect_wy[320][200],filtr[3][3];

void filtr()
{
 int w, k, w_f, k_f, i;

for ( i = 0 ; i < ILE_RAZY ; i++ )

 for ( w = 0 ; w <= 319 ; w++ )
  for ( k = 0 ; k <= 199 ; k++ )
   for ( x_f = 0 ; x_f < 3 ; x_f++ )
    for ( y_f = 0 ; y_f < 3 ; y_f++ )
     vect_wy[w][k] += vect_we[w + x_f][k + y_f] * filtr[x_f][y_f];
}

main()
{
 filtr();
}
```

Tabela 6

Czas realizacji zadania opisanego w pkt. 3.2.3

Komputer prowadzący obliczenia		Czas realizacji zadania
Cray J90	opcja O1	430,129[s]
	opcja O2	255,159[s]
SUN Enterprise 6500		250,602[s]
SPARCclassic1		1374,541[s]

4. Wnioski

System PVM wydaje się być atrakcyjną alternatywą dla drogich systemów wieloprocesorowych. Jednak poważnym jego ograniczeniem jest konieczność umiejętnego posługiwania się grupą funkcji dostępnych dla programisty po dołączeniu do programów głównego i potomnego pliku „pvm3.h" (por. rozwiązanie w wersji sekwencyjnej i przy wykorzystaniu systemu PVM w pkt. 3.2.1.). Dodatkowo, jak wynika ze szczegółowych obserwacji, dla małej liczby danych (np. wektorów 100-elementowych) większość czasu

tracona jest na komunikację i synchronizację procesów, a nie na operacje arytmetyczno-logiczne (szybciej wykonywany jest program sekwencyjny).

4.1. Koszt wykonania operacji w komputerze wektorowym

Analiza wielu zadań wykonywanych w superkomputerach wektorowych dostarczyła materiału, który stał się podstawą do sformułowania następujących wniosków:

- Im mniej jest zaangażowanych podjednostek wykonujących zadanie (mniejszy potok), tym wolniej rośnie czas wykonania tego zadania.
- Również wzrost liczby taktów zegara, potrzebnych do przejścia przez poszczególne jednostki funkcyjne, powoduje znaczne pogorszenie wyników na niekorzyść jednostki wektorowej.
- Warunkiem uzyskania dobrego przyspieszenia jest następujące kryterium: najdłuższy czas wykonania podzadania przez pojedynczy element potoku komputera wektorowego musi być mniejszy od czasu wykonania operacji w komputerze skalarnym. (Oczywiście sumaryczny koszt wykonania operacji w komputerze wektorowym może być większy niż w komputerze skalarnym).

Można to łatwo zaobserwować na teoretycznych modelach jednostki skalarnej i wektorowej. Dla celów porównawczych przyjęto, że wykonanie danej operacji składa się z trzech faz:

Faza 1. Pobranie danych (w przypadku jednostki wektorowej w fazie tej przygotowywany jest cały wektor),

Faza 2. Wykonanie operacji,

Faza 3. Zapis wyniku.

Dodatkowo założono potokową realizację cyklu rozkazowego w komputerze skalarnym i potokową organizację jednostki arytmetyczno-logicznej w komputerze wektorowym oraz ustalono hipotetyczne czasy trwania poszczególnych faz w obu komputerach wyrażone w jednostkach czasu [jc] (rys 5.)[1].

Dla obydwu jednostek na łączny czas wykonania operacji składa się czas przygotowania danych i wyniku (t_{START}) oraz czas samej operacji na wektorze (t_{OP}) (1).

$$t = t_{START} + t_{OP} \text{ [jc]} \tag{1}$$

Przyjęto uproszczenie, że czas przygotowania danych dla danego procesora jest stały i nie zależy od długości przetwarzanego wektora.

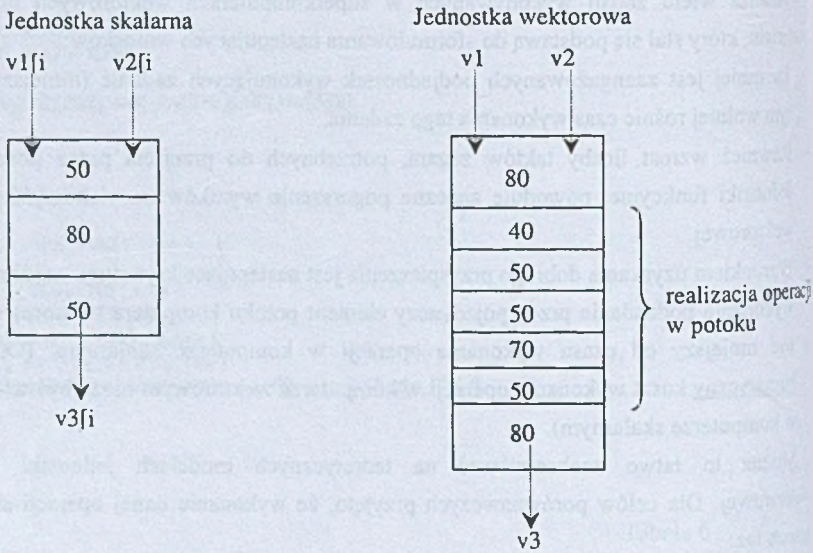
$$t1_{START} = 50 + 40 = 90 \text{ [jc]} \tag{2}$$

[1] Por. rzeczywisty czas wykonania przykładowej operacji na wektorach przedstawiony na rys. 9.

$$t2_{START} = 80 + 80 = 160 \text{ [jc]} \quad (3)$$

W przeciwieństwie do czasu przygotowania danych czas wykonania operacji jest ściśle związany z długością wektora (4).

$$t1_{OP} = 80 * n \text{ [jc]} \quad (4)$$



Rys. 5. Konceptualne modele jednostek: skalarnej i wektorowej
Fig. 5. Conceptual models of scalar and vector units

Ponieważ potokowa jednostka komputera wektorowego składa się z segmentów, dla których elementarny czas wykonania jest różny, będzie to miało wpływ na czas wykonania operacji: odstępy czasu między pobieraniem do jednostki kolejnych składowych wektora będą równe czasowi najdłuższego kroku elementarnej operacji na wektorze. W omawianym przykładzie jest to 70 [jc]. Pierwszy element rozwiązania pojawi się za:

$$40 + 50 + 50 + 70 + 50 = 260 \text{ [jc]} \quad (5)$$

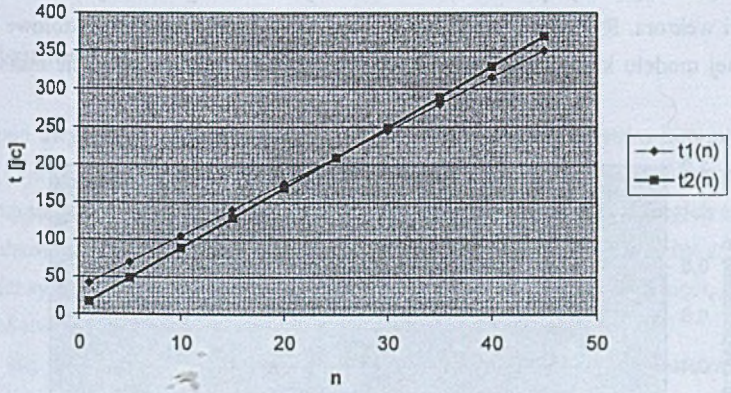
Stąd:

$$t2_{OP} = 260 + 70 * (n - 1) = 190 + 70 * n \text{ [jc]} \quad (6)$$

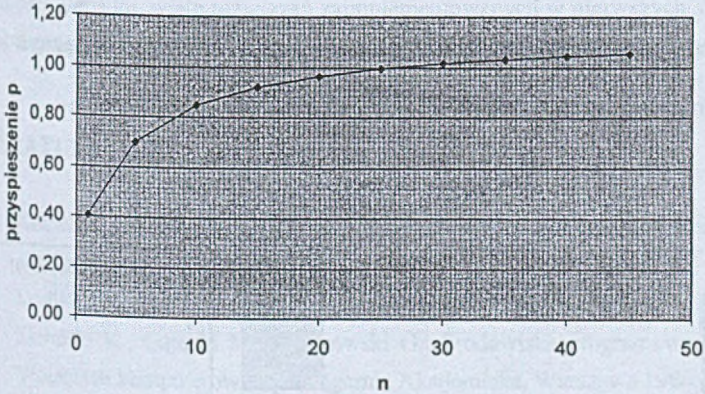
Po uwzględnieniu powyższych wartości otrzymujemy wartość przyspieszenia wyrażoną wzorem (7).

$$\frac{t1(n)}{t2(n)} = \frac{90 + 80 * n}{160 + 190 + 70 * n} = \frac{9 + 8 * n}{35 + 7 * n} \quad (7)$$

Z wykresu zależności przyspieszenia **p** od długości wektora **n** (rys. 7) wynika, że stosowanie komputera wektorowego staje się opłacalne dla wektora o długości większej od 29.



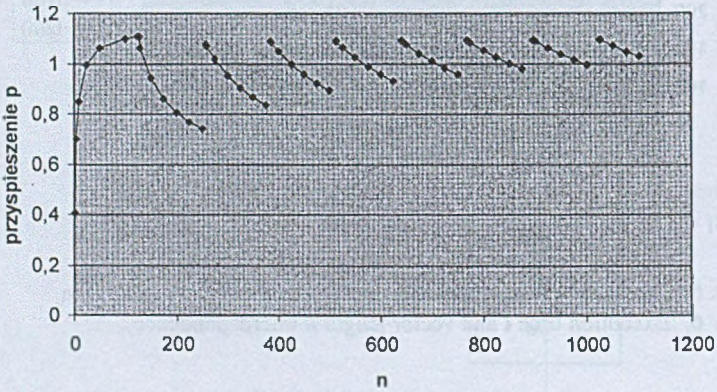
Rys. 6. Zależność czasu wykonania operacji **t** od długości wektora **n**
Fig. 6. Execution time **t** and vector length **n** interdependence



Rys. 7. Zależność przyspieszenia **p** od długości wektora **n**
Fig. 7. Acceleration **p** and vector length **n** interdependence

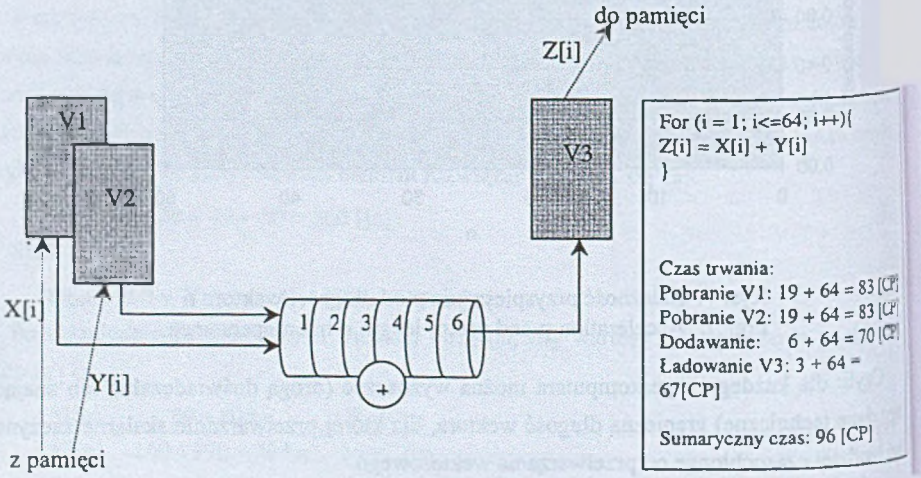
Czyli: dla każdego superkomputera można wyznaczyć (drogą doświadczalną lub znając jego dane techniczne) graniczną długość wektora, dla której przetwarzanie skalarne zaczyna być bardziej czasochłonne od przetwarzania wektorowego.

Należy dodać, że w praktyce, w związku z koniecznością przeładowywania rejestrów wektorowych, zależność przyspieszenia od długości wektora ma przebieg nieciągły zasadniczo różniący się od zobrazowanego na rys. 7. Konsekwencją tego jest znaczne przesunięcie się granicy opłacalności stosowania jednostki wektorowej w kierunku większych długości wektora. Rysunek 8 przedstawia sytuację, w której rejestry wektorowe omawianego wcześniej modelu komputera wektorowego mogły jednorazowo zmieścić maksymalnie 128 liczb.



Rys. 8. Faktyczny przebieg przyspieszenia **p** w zależności od długości wektora **n**

Fig. 8. True interdependence between acceleration **p** and vector length **n**



Rys. 9. Realizacja operacji dodawania dwóch wektorów w superkomputerze Cray J90

Fig. 9. Execution of two vectors summation in supercomputer Cray J90

Jak wynika z oceny przyspieszenia, komputer wektorowy jest kompromisem pomiędzy małą wydajnością systemów przetwarzających skalarnie (por. tabela 2, tabela 6) a dużym kosztem[1] i niezbyt dużą wydajnością systemów wieloprocesorowych (por. tabela 5).

5. Podsumowanie

Biorąc pod uwagę moc obliczeniową i strukturę oferowanych na rynku komputerowym systemów, daje się zaobserwować tendencję dążenia w kierunku superkomputerów[2]. Łączenie grup stacji roboczych w sieci – jako wciąż atrakcyjna alternatywa drogich systemów superkomputerowych – posiada znamienne ograniczenia, wynikające chociażby z problemu skalowalności systemu[3] czy niskich prędkości i wąskich pasm przetwarzania sieci, łączących poszczególne stacje robocze.

Wydaje się, że –mimo wspomnianych wcześniej kosztów– głównym nurtem rozwoju systemów komputerowych będzie dalsza ewolucja superkomputerów. Hipoteza ta ma swoje uzasadnienie w fakcie, że podczas gdy – według danych szacunkowych – komputery osobiste dezaktualizują się po upływie ok. 2 lat, superkomputery – nawet po 10 latach – stanowią wciąż wysoko wydajną platformę sprzętową dla wykonywania obliczeń wysokiej skali złożoności. Ponadto, wiele rozwiązań zaimplementowanych w pierwszych superkomputerach jest obecnie obowiązującym standardem (np. przetwarzanie potokowe).

LITERATURA

1. Kozielski S., Szczerbiński Z.: Komputery równoległe. Architektura, elementy oprogramowania. WNT, Warszawa 1994.
2. Loshin D.: Superkomputery bez tajemnic. Mikom, Warszawa 1997.
3. Zieliński K., Gajęcki M., Czajkowski G.: Środowiska programowania rozproszonego w sieciach komputerowych. Księgarnia Akademicka, Warszawa 1996.
4. CRAY Assembly Language for Cray PVP Systems, SR-3108 9.1. Adres strony internetowej: http://dynaweb.hpc.utexas.edu:8080/dynaweb/all/

---

[1] Koszty zakupu superkomputerów o architekturze wektorowej i wieloprocesorowej są porównywalne.

[2] Szybkość działania dzisiejszego komputera osobistego jest równa szybkości, z jaką wykonywał obliczenia superkomputer CRAY-1 z 1978 r.

[3] Przy wzroście liczby węzłów systemu do pewnej granicy szybkość działania przestaje wzrastać.

5. Cray J90 Optimization, 1997. Adres strony internetowej: http://sc.tamu.edu/shrtcrs/opt1_j90/opt1-2.html
6. Optimizing Application Code on UNICOS Systems, 004-2192-002. Adres strony internetowej: http://dynaweb.hpc.utexas.edu:8080/dynaweb/stnd_c_pe/004-2192-002/
7. Poznańskie Centrum Superkomputerowo-Sieciowe PCSS. Adres strony internetowej: http://www.man.poznan.pl/
8. Pittsburgh Supercomputing Center, Carnegie Mellon University. Adres strony internetowej: http://www.psc.edu/general/software/cray/j90/
9. CONVEX Architecture Reference Manual, CONVEX Computer Corporation 1992.
10. CONVEX Compiler Utilities User's Guide, CONVEX Computer Corporation 1992.
11. Wielgus M.: Optymalizacja programów fortranowskich. Akademickie Centrum Komputerowe CYFRONET, Kraków 1995.

Recenzent: Prof. dr hab. inż. Tadeusz Czachórski

Wpłynęło do Redakcji 12 kwietnia 2000 r.

Abstract

The main purpose of the paper is to compare vector, multiprocessor and scalar processing. Vector processing is almost always faster than scalar processing, because a single machine-level instruction can operate on a series of operands within a vector register. This is a form of low-level parallelism that allows many operations to run concurrently within a single CPU (Fig. 4). Therefore, most CPU optimisation techniques focus on using as much vectorization as possible within the code (point 2.1). Profiling code of source programs written in C language allows to determine which code segments consume most of CPU time.

The article is organized as follows. The first part is connected with fundamental concepts which are specific to vector processing (strip mining, multistreaming, chaining). The second section contains advantages and drawbacks of described approaches. These two sections present the basic platform to analyze several examples, which are specified in the third part. Results of experiments comparing various techniques used in examples are contained in section 4. Finally, conclusions and directions for future are presented in section 5.