Podstawowa analiza wykonania programów na komputerach

Wydajność współczesnych komputerów będziemy podawać w milionach operacji zmiennopozycyjnych na sekundę – megaflopach (Mflops) i definiować jako

$$r = \frac{N}{t} \text{ Mflops},$$

gdzie N oznacza liczbę operacji zmiennopozycyjnych wykonanych w czasie t mikrosekund. Czas wykonania programu spełnia zatem zależność

$$t = \frac{N}{r} \text{ microsec.}$$

Dla poszczególnych fragmentów programu mogą być osiągane różne wydajności systemu, a zatem wzór opisuje tylko wypadkową (średnią) wydajność. Poniżej przedstawiamy najważniejsze modele, które znacznie lepiej oddają specyfikę wykonania programów na współczesnych komputerach.

Prawo Amdahla

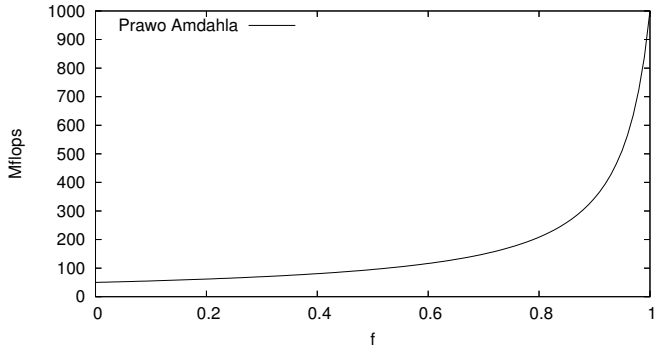
Niech f będzie częścią programu składającego się z N operacji zmiennopozycyjnych, dla której system osiąga wydajność V, zaś $1-f$ częścią charakteryzującą się wydajnością S, przy czym $V \gg S$. Wówczas otrzymujemy łączny czas wykonywania obliczeń obu części programu

$$t = f\frac{N}{V} + (1-f)\frac{N}{S} = N(\frac{f}{V} + \frac{1-f}{S})$$

oraz otrzymujemy wypadkową wydajność

$$r = \frac{1}{\frac{f}{V} + \frac{(1-f)}{S}} \text{ Mflops.}$$

Powyższy wzór nosi nazwę *prawo Amdahla* i opisuje wpływ przyspieszenia (optymalizacji) fragmentu programu na wydajność komputera realizującego obliczenia.



Rysunek 1: Prawo Amdahla dla $V = 1000$ oraz $S = 50$ Mflops.

Jako przykład rozważmy sytuację, gdy $V = 1000$ oraz $S = 50$ Mflops. Rysunek 1 pokazuje wydajność komputera (Mflops) w zależności od wartości f. Możemy zaobserwować, że relatywnie duża wartość $f = 0.8$, wykonywana z maksymalną możliwą wydajnością skutkuje wypadkową wydajnością równą 200 Mflops, a zatem wykorzystaniem zaledwie 20% teoretycznej maksymalnej wydajności (ang. *peak performance*). Oznacza to, że aby uzyskać bardzo dużą szybkość wykonania programu, należy zadbać o zoptymalizowanie jego najwolniejszych części. Zauważmy też, że w omawianym przypadku największy wzrost wydajności uzyskujemy przy zmianie wartości f od 0.9 do 1.0. Zwykle jednak najwolniejsze (najmniej efektywne) fragmenty programu, to obliczenia przyjmujące postać rekurencji, bądź też realizujące dostęp do pamięci w sposób nieoptymalny, które wymagają zastosowania specjalnych algorytmów dla osiągnięcia zadowalającej szybkości obliczeń.

Model Hockney'a-Jesshope'a

Innym modelem, który dokładniej charakteryzuje obliczenia wektorowe jest model Hockney'a - Jesshope'a obliczeń wektorowych, pokazujący szybkość wykonania pętli na danym procesorze. Rozważmy pętlę składającą się z N obrotów. Szybkość jej wykonania wyraża się wzorem

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \text{ Mflops}, \tag{1}$$

gdzie r_∞ oznacza szybkość wykonania (w megaflopach) „nieskończonej" pętli (bardzo długiej), zaś $n_{1/2}$ jest długością (liczbą obrotów) pętli, dla której osiągnięta jest szybkość wykonania około $r_\infty/2$. Przykładowo, operacja `DOT` wyznaczenia iloczynu skalarnego wektorów $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$

$$dot \leftarrow \mathbf{x}^T\mathbf{y}$$

ma postać następującej pętli o liczbie obrotów równej N.

```
dot=0.0
do i=1,N
  dot=dot+y(i)*x(i)
end do
```

Łączna liczba operacji zmiennopozycyjnych wykonywanych w powyższej konstrukcji wynosi zatem $2N$. Stąd czas wykonania operacji `DOT` dla wektorów o N składowych wynosi w sekundach

$$T_{DOT}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N). \tag{2}$$

Podobnie operacja `AXPY` może być w najprostszej postaci zaprogramowana jako następująca konstrukcja iteracyjna.

```
do i=1,N
  y(i)=y(i)+alpha*x(i)
end do
```

Na każdy obrót pętli przypadają dwie operacje arytmetycze, stąd czas jej wykonania wyraża się wzorem

$$T_{AXPY}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N). \tag{3}$$

Oczywiście wielkości r_∞ oraz $n_{1/2}$ występujące odpowiednio we wzorach (2) i (3) są na ogół różne, nawet dla tego samego procesora. Wadą modelu jest to, że nie uwzględnia on zagadnień związanych z organizacją pamięci w komputerze. Może się zdarzyć, że taka sama pętla operująca na różnych zestawach danych alokowanych w pamięci operacyjnej w odmienny sposób, będzie w każdym przypadku wykonywana z bardzo różnymi prędkościami. Może to być spowodowane konfliktami w dostępie do banków pamięci bądź też innym schematem wykorzystania pamięci podręcznej.

Jako przykład ilustrujący zastosowanie modelu Hockney'a-Jesshope'a do analizy algorytmów rozważmy dwa algorytmy rozwiązywania układu równań liniowych

$$L\mathbf{x} = \mathbf{b}, \tag{4}$$

gdzie $\mathbf{x}$, $\mathbf{b} \in \mathbb{R}^N$, o macierzy dolnotrójkątnej postaci

$$L = \begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{N,1} & \cdots & \cdots & a_{N,N} \end{pmatrix}. \tag{5}$$

Układ może być rozwiązany przy pomocy nastepującego algorytmu:

$$\begin{cases} x_1 = b_1/a_{11} \\ x_i = \left(b_i - \sum_{k=1}^{i-1} a_{ik}x_k\right)/a_{ii} \quad \text{dla } i = 2, \ldots, N. \end{cases} \tag{6}$$

Zauważmy, że w algorytmie dominuje operacja `DOT`. Stąd pomijając czas potrzebny do wykonania N dzieleń zmiennopozycyjnych wnosimy, że łączny czas działania algorytmu wyraża się wzorem

$$\begin{aligned} T_1(N) &= \sum_{k=1}^{N-1} T_{DOT}(k) = \frac{2 \cdot 10^{-6}}{r_\infty}\left(n_{1/2}(N-1) + \sum_{k=1}^{N-1} k\right) \\ &= \frac{2 \cdot 10^{-6}}{r_\infty}(N-1)\left(n_{1/2} + \frac{N}{2}\right). \end{aligned}$$

Inny algorytm otrzymamy wyznaczając postać macierzy L^{-1}. Istotnie, macierz L może być zapisana jako $L = L_1 L_2 \cdots L_N$, gdzie

$$L_i = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & a_{ii} & & \\ & & \vdots & \ddots & \\ & & a_{N,i} & & 1 \end{pmatrix}$$

oraz

$$L_i^{-1} = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & \frac{1}{a_{ii}} & & \\ & & -\frac{a_{i+1,i}}{a_{ii}} & 1 & \\ & & \vdots & & \ddots \\ & & -\frac{a_{N,i}}{a_{ii}} & & 1 \end{pmatrix}.$$

Oczywiście zachodzi

$$L^{-1} = L_N^{-1} L_{N-1}^{-1} \cdots L_1^{-1}.$$

Stąd otrzymujemy następujący wzór:

$$\begin{cases} \mathbf{y}_0 = \mathbf{b} \\ \mathbf{y}_i = L_i^{-1}\mathbf{y}_{i-1} \quad \text{dla } i = 1, \ldots, N \\ \mathbf{x} = \mathbf{y}_N \end{cases} \tag{7}$$

Operacja mnożenia macierzy L_i^{-1} przez wektor $\mathbf{y}_{i-1}$ nie wymaga jawnego wyznaczania postaci macierzy. Istotnie, rozpisując wzór (7) otrzymujemy

$$\mathbf{y}_i = \begin{pmatrix} y_1^{(i)} \\ y_2^{(i)} \\ \vdots \\ y_i^{(i)} \\ \vdots \\ y_{N-1}^{(i)} \\ y_N^{(i)} \end{pmatrix} = L_i^{-1} \begin{pmatrix} y_1^{(i-1)} \\ y_2^{(i-1)} \\ \vdots \\ y_i^{(i-1)} \\ \vdots \\ y_{N-1}^{(i-1)} \\ y_N^{(i-1)} \end{pmatrix} = \begin{pmatrix} y_1^{(i-1)} \\ \vdots \\ y_{i-1}^{(i-1)} \\ y_i^{(i-1)}/a_{ii} \\ y_{i+1}^{(i-1)} - a_{i+1,i} y_i^{(i-1)}/a_{ii} \\ \vdots \\ y_N^{(i-1)} - a_{N,i} y_i^{(i-1)}/a_{ii} \end{pmatrix} \tag{8}$$

W algorytmie opartym na wzorach (7) i (8) nie trzeba składować wszystkich wyznaczanych wektorów. Każdy kolejny wektor $\mathbf{y}_i$ będzie składowany na miejscu poprzedniego, to znaczy $\mathbf{y}_{i-1}$. Stąd operacja aktualizacji wektora we wzorze (8) przyjmie postać sekwencji operacji skalarnej

$$y_i \leftarrow y_i/a_{ii}, \tag{9}$$

a następnie wektorowej

$$\begin{pmatrix} y_{i+1} \\ \vdots \\ y_N \end{pmatrix} \leftarrow \begin{pmatrix} y_{i+1} \\ \vdots \\ y_N \end{pmatrix} - y_i \begin{pmatrix} a_{i+1,i} \\ \vdots \\ a_{N,i} \end{pmatrix}. \tag{10}$$

Zauważmy, że (10) to właśnie operacja `AXPY`. Stąd podobnie jak w przypadku poprzedniego algorytmu, pomijając czas potrzebny do wykonania (9) otrzymujemy

$$\begin{aligned} T_2(N) &= \sum_{k=1}^{N-1} T_{AXPY}(N-k) = \frac{2 \cdot 10^{-6}}{r_\infty} \left(n_{1/2}(N-1) + \sum_{k=1}^{N-1} (N-k) \right) \\ &= \frac{2 \cdot 10^{-6}}{r_\infty} (N-1) \left(n_{1/2} + \frac{N}{2} \right). \end{aligned} \tag{11}$$

Zatem dla obu algorytmów czas wykonania operacji wektorowych wyraża się podobnie w postaci funkcji zależnych od parametrów r_∞, $n_{1/2}$, właściwych dla operacji `DOT` i `AXPY`. Poniższa tabela pokazuje przewidywany czas realizacji obu algorytmów na komputerze Convex C3210 (zaniedbujemy jednakowy dla obu algorytmów czas potrzebny na wykonanie N operacji dzielenia).

Alg. 1	$r_\infty = 18$	$n_{1/2} = 36$	$T_1(1000) = 0.059$ sec.
Alg. 2	$r_\infty = 16$	$n_{1/2} = 26$	$T_2(1000) = 0.066$ sec.

(12)

Zauważmy, że mimo jednakowej, wynoszącej w przypadku obu algorytmów liczby operacji arytmetycznych N^2, wyznaczony czas działania każdego algorytmu jest inny.

Prawo Amdahla dla obliczeń równoległych

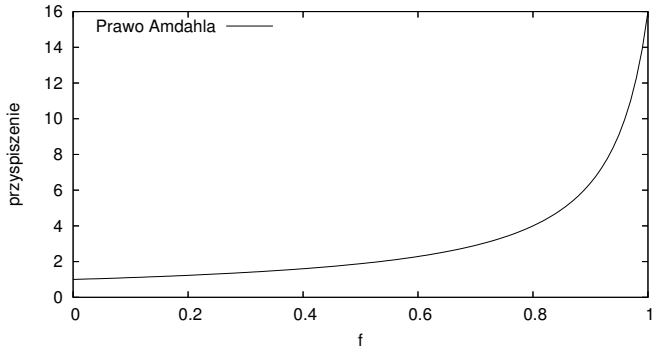
Przypuśćmy, że czas wykonania programu na jednym procesorze wynosi t_1. Niech f oznacza część programu, która może być idealnie zrównoleglona na p procesorach. Pozostała sekwencyjna część programu $(1-f)$ będzie wykonywana na jednym procesorze. Łączny czas wykonania programu równoległego przy użyciu p procesorów wynosi

$$t_p = f\frac{t_1}{p} + (1-f)t_1 = \frac{t_1(f+(1-f)p)}{p}.$$

Stąd *przyspieszenie* (ang. speedup) wyraża się wzorem

$$s_p = \frac{t_1}{t_p} = \frac{p}{f+(1-f)p}. \tag{13}$$

Rysunek 2 pokazuje wpływ zrównoleglonej części f na przyspieszenie programu. Można zaobserwować bardzo duży negatywny wpływ części sekwencyjnej (niezrównoleglonej) na osiągnięte przyspieszenie – podobnie jak w przypadku podstawowej wersji prawa Amdahl'a.

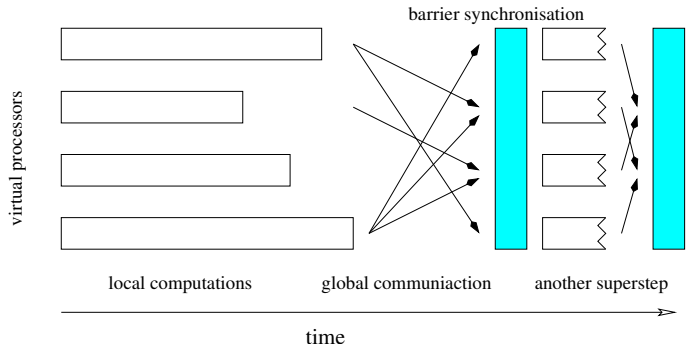


Rysunek 2: Prawo Amdahla dla dla obliczeń równoległych, $p = 16$.

Model BSP

W celu przeprowadzania analizy wykonania programów w środowisku rozproszonym, rozważmy następujący model BSP (ang. Bulk Synchronous Parallel Architecture). Program równoległy składa się z pewnej liczby superkroków (rysunek 3). Każdy superkrok składa się z obliczeń wykonywanych przez procesory na danych znajdujących się w ich pamięciach lokalnych, globalnej wymiany danych (komunikacji) oraz na koniec synchronizacji. Model BSP charakteryzuje się następującymi parametrami: liczbą dostępnych procesorów p, czasem g (liczonym w jednostkach równych czasowi wykonania jednej operacji zmiennopozycyjnej) potrzebnym do wysłania bądź odbioru jednego słowa maszynowego oraz czasem l (również liczonym w jednostach określonych przez czas wykonania operacji zmiennopozycyjnej) potrzebnym do synchronizacji wszystkich procesorów. Wykonanie operacji synchronizacji na koniec superkroku gwarantuje, że wszystkie wysyłane dane dotarły do miejsca przeznaczenia. Złożoność (inaczej koszt) superkroku definiujemy jako wielkość

$$w_{\max} + gh_{\max} + l, \tag{14}$$



Rysunek 3: Program w modelu BSP.

gdzie $w_{\max}$ oznacza maksymalną liczbę operacji arytmetycznych wykonywanych lokalnie w ramach superkroku, zaś $h_{\max}$ maksymalną liczbą słów maszynowych wysyłanych lub odbieranych przez pewien procesor. Złożonością programu nazywamy sumę złożoności jego poszczególnych superkroków. Zauważmy, że mając daną złożoność programu oraz szacunkowy czas (w sekundach) potrzebny do wykonania jednej operacji zmiennopozycyjnej możemy wyznaczyć czas wykonania programu.