UMCS

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: Matematyka
Specjalność: Informatyczna

Rafał Szczerski
nr albumu: 2524659

Równoległe algorytmy mnożenia macierzy

Parallel matrix multiplication algorithms

Praca licencjacka
napisana w Zakładzie Informatyki
pod kierunkiem dr Beaty Byliny

Lublin rok 2015

Spis treści

Wstęp

Można wyróżnić trzy zasadnicze powody obecnego zainteresowania obliczeniami równoległymi.

1. Stały spadek kosztów sprzętu komputerowego,
2. Rozwój VLSI (*Very-large-scale integration*) do poziomu umożliwiającego projektowanie układów scalonych zawierających miliony tranzystorów na pojedyńczym chipie,
3. Osiągnięcie fizycznych ograniczeń czasu cyklu procesora w architekturze von Neumanna (rys 1.5).

Obliczenia równoległe, w świetle ograniczeń fizycznych procesorów jednordzeniowych, są odpowiedzią na potrzebę wykonywania szybszych obliczeń. Szybsze obliczenia pozwalają na obliczenia w większej skali i otrzymywanie szybszych rozwiązań.

Rozdział 1

Wiadomości wstępne

1.1 Ustalenia terminologiczne

Definicja 1.1 (Zbiór przechodni)**.** Zbiór A nazywamy **przechodnim**, wtedy i tylko wtedy, gdy $\forall x\,(x \in A \wedge y \in x \implies y \in A)$.

Definicja 1.2 (Domknięcie przechodnie zbioru)**.** Domknięciem przechodnim zbioru X nazywamy najmniejszy w sensie inkluzji zbiór przechodni, który zawiera X.

Definicja 1.3 (Graf skierowany (DG))**.** Powiedzmy, że:

1. $V \neq \emptyset$ jest zbiorem
2. $E \subseteq V \times V$

Grafem skierowanym G nazwiemy dwójkę (V, E).

Definicja 1.4 (Acykliczny graf skierowany (DAG))**.** Acyklicznym grafem skierowanym nazywamy graf skierowany nie zawierający cykli.

Definicja 1.5 (Domknięcie przechodnie grafu)**.** Niech $G = (V, A)$ będzie grafem skierowanym. Graf skierowany $G^+ = (V, A^+)$ nazywamy **domknięciem przechodnim** grafu G, gdy A^+ jest zbiorem wszystkich takich par (a, b) wierzchołków zbioru V, że w grafie G istnieje droga z a do b.

Definicja 1.6 (Graf zależności)**.** Niech dane będą zbiór $S \neq \emptyset$, relacja przechodnia $R \subseteq S \times S$. **Grafem zależności** nazywamy graf $G = (S, T)$ i $T \subseteq R$, gdzie R jest przechodnim domknięciem T.

Definicja 1.7 (Ścieżka)**. Ścieżką** łączącą v_0 z v_n o długości n nazywamy ciąg wierzchołków $(v_0, v_1, \ldots, v_n)$ taki, że dla każdego $k \in \{0, 1, \ldots, n-1\}$ istnieje krawędź z v_k do v_{k+1}.

Definicja 1.8 (Droga)**. Drogą** w grafie G nazywamy ścieżkę, której wierzchołki są różne.

Definicja 1.9 (Długość drogi)**. Długością** drogi w grafie G nazywamy liczbę krawędzi, które zawiera droga.

Definicja 1.10 (Cykl)**.** Drogę zamkniętą długości co najmniej 1 z ciągiem wierzchołków $x_1x_2 \ldots x_nx_1$ nazywamy **cyklem**, jeśli wszystkie wierzchołki $x_1, x_2 \ldots x_n$ są różne.

Definicja 1.11 (Stopień wierzchołka)**. Stopień** $d_G(v)$ **wierzchołka** v definiujemy jako liczbę incydentnych z v krawędzi. Każdemu wierzchołkowi v grafu skierowanego G możemy przypisać stopień wyjściowy (ang. *indegree*) $d_G^+(v)$ i stopień wejściowy (ang. *outdegree*) $d_G^-(v)$:

$$d_G^+(v) = \#\{w|(v, w) \in E\}$$
$$d_G^-(v) = \#\{w|(w, v) \in E\}$$

Definicja 1.12 (Macierz)**.** Niech $\mathbb{K}$ będzie ciałem. Macierzą o m wierszach i n kolumnach i wartościach w $\mathbb{K}$ (krótko: macierzą $m \times n$) nazywamy każde odwzorowanie $A : \{1, \ldots, m\} \times \{1, \ldots, n\} \to \mathbb{K}, (i, j) \longmapsto A_{ij}$

1.2 Klasyfikacja algorytmów

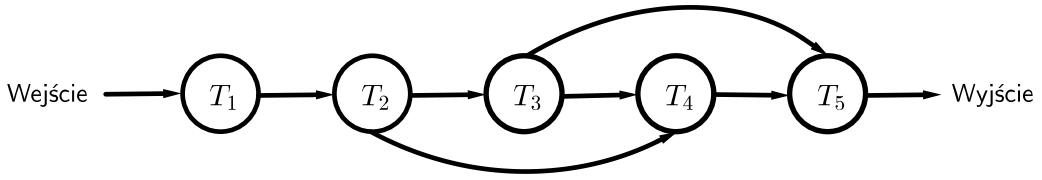
Definicja 1.13 (Algorytm)**.** Zbiór jednoznacznie określonych reguł lub zadań obliczeniowych prowadzących w skończonej ilości kroków do rozwiązania pewnego problemu [1].

Określone w ten sposób zadania obliczeniowe są z reguły względem siebie niezależne. Pewne z zadań mogą być wykonywane równolegle, inne muszą być wykonywane sekwencyjnie, jedno po drugim. Wobec tego algorytm może być określony częściowo równolegle, częściowo sekwencyjnie.

Podstawowymi elemetami określającymi dowolny algorytm są:

1. zadania do wykoniania,
2. zależności pomiędzy zadaniami polegające na określeniu czy dane wyjściowe któregoś z zadań nie są danymi wejściowymi dla innego zadania,
3. zbiór danych wejściowych wymaganych przez algorytm,
4. zbiór danych wyjściowych otrzymywanych po wykonania algorytmu.

Definicja 1.14 (Algorytm sekwencyjny)**. Algorytm sekwencyjny** (rys. 1.1) jest ciągiem dokładnie sprecyzowanych zadań obliczeniowych T_i, $i \in \mathbb{N}$ rozwiązujących dany problem, tj. wyznaczających dane wyjściowe na podstawie danych wejściowych. Zakłada się, że w algorytmie sekwencyjnym zadania wykonywane są przez jeden procesor.

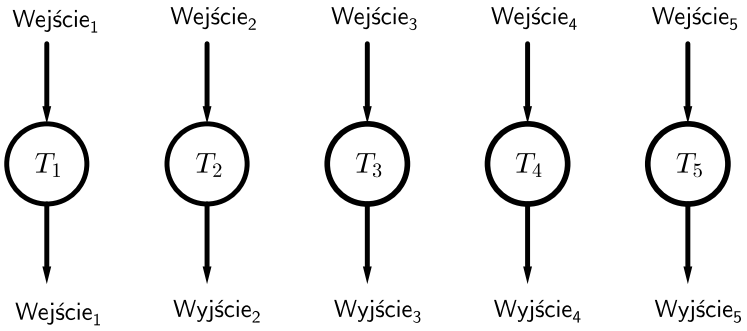


Rysunek 1.1: Algorytm sekwencyjny

W celu rozwiązania problemu za pomocą większej liczby procesorów należy go zdekomponować na podproblemy, które mogą być rozwiązane równolegle. Każdy z podproblemów rozwiązywany jest przez odrębny algorytm będący składową algorytmu równoległego.

Definicja 1.15 (Równoległość)**. Równoległość** w odniesieniu do oprogramowania jest to symultaniczny transfer, występowanie albo przetwarzanie poszczególnych części pewnej całości, takich jak bity składające się na znak albo znaki pewnego słowa, używając osobnych urządzeń dla ich różnych części [1].

Definicja 1.16 (Algorytm równoległy)**. Algorytmem równoległym** (rys. 1.2) nazywamy każdy algorytm, w którym spośród określonych w nim zadań T_1, T_2, ..., T_n co najmniej dwa zadania T_i, T_j, $i \neq j$ dzięki ich wzajemnej niezależności, mogą być wykonane równocześnie [2].



Rysunek 1.2: Algorytm równoległy

1.3 Architektury równoległe

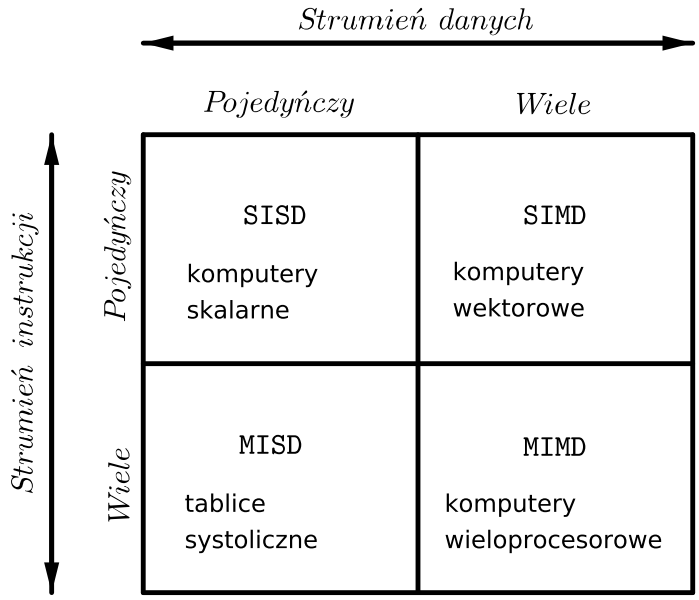
Definicja 1.17 (Architektura równoległa)**. Architektura równoległa** jest to architektura wieloprocesorowa, na której można wykonywać przetwarzanie równoległe [1].

Algorytmy równoległe i architektury równoległe są ze sobą blisko spokrewnione. Równoległość może być zaimplementowana na wielu poziomach używając technik sprzętowych i programowych.

1. Równoległość na poziomie danych (ang. *Data-level parallelism*), gdzie pracujemy na wielu bitach danych lub na wielu danych jednocześnie.

2. Równoległość na poziomie instrukcji (ang. *Instruction-level parallelism*, ILP), gdzie jednocześnie procesor może wykonać więcej niż jedną instrukcję.

3. Równoległość na poziomie wątków (ang. *Thread-level parallelism*, TLP). Wątek jest częścią programu, która współdzieli zasoby procesora z innymi wątkami. W TLP wiele programowych wątków jest uruchamianych jednocześnie na jednym bądź wielu procesorach.



Rysunek 1.3: Klasyfikacja Flynna

4. Równoległość na poziomie procesów (ang. *Process-level parallelism*). Proces to program, który jest uruchomiany na komputerze. Rezerwuje on własne zasoby komputera, takie jak przestrzeń pamięciową i rejestry.[2]

Przykład 1.1. Prostym przykładem algorytmu równoległego jest serwer sieciowy, który każde zapytanie przychodzące przetwarza niezależnie od innych zapytań. Innym przykładem są wielozadaniowe systemy operacyjne radzące sobie z jednoczesną obsługą kilku uruchomionych programów.

Klasyfikacja Flynna

Architektury komputerowe można podzielić na klasy ze względu na liczbę równolegle wykonywanych instrukcji oraz dostępnych strumieni danych. Klasyfikację taką (rys. 1.3) zaproponował Michael J. Flynn w 1966 roku i przyjęła ona swoją nazwę od jego nazwiska (rys. 1.3).

SISD. Klasa SISD (ang. *Single Instruction, Single Data*) odnosi się do komputerów wykonujących pojedyńczy strumieniem instrukcji i przetwarzających pojedyńczy strumień danych. Są to komputery całkowicie sekwencyjne, które nie wykonują żadnych obliczeń równoległych.

SIMD. Klasa SIMD (ang. *Single Instruction, Multiple Data*) odnosi się do komputerów obsługujących pojedyńczy strumień instrukcji i przetwarzających

wiele strumieni danych. Na różnych zbiorach danych wykownywane są te same operacje. Jako przykład takiej architektury warto wymienić przede wszystkim wczesne komputery macierzowe (nazywane niekiedy wektorowymi) ze wspólną pamięcią i macierzą jednostek przetwarzających nadzorowanych przez jednostkę sterującą, takie jak komputer ILLIAC IV wykorzystywany przez NASA w latach '70.

MISD. Klasa MISD (ang. *Multiple Instruction, Single Data*) odnosie się do komputerów wykonujących jednocześnie wiele instrukcji przetwarzających jeden współny strumien danych. Przykładem takiej architektury jest tablica systoliczna[1]. Tablica systoliczna jest to układ prostych jednostek przetwarzających połączonych w sieć z sąsiadującymi jednostkami, które synchronicznie wykonują pewne elementarne operacje obliczeniowe.

MIMD. Klasa MIMD (ang. *Multiple Instruction, Multiple Data*) odnosi się do komputerów równolegle wykonujących wiele instrukcji z których każda przetwarza własne strumienie danych. Do tej kategorii zaliczają się multiprocesory[2] (większość współczesnych komputerów PC) i multikomputery[3].

Większość obecnie używanych komputerów równoległych to klastry o architekturze mieszanej. Klaster jest układem niezależnych jednostek obliczeniowych (węzły) połączonych szybką siecią komunikacyjną.[3].

1.4 Reprezentacja algorytmów

Wiele obliczeń możemy repezentować za pomocą acyklicznych grafów skierowanych. Każde wejście jest oznaczane przez węzeł bez dochodzących do niego łuków. Operacje oznaczamy przez węzły do których wchodzą łuki z innych węzłów oznaczających argumenty (operandy). Stopień wejściowy dowolnego węzła wynosi co najwyżej 2. Węzeł, którego stopień wyjściowy jest równy 0 oznacza wyjście. Zakładamy, że każdy węzeł przedstawia operację, która wymaga jednej jestostki czasu wykonania.

[1]Nazwa pochodzi od skurczu mięśni serca przez analogię „pompowania" danych do jednostek przetwarzających na wzór krwi w naczyniach krwionośnych.

[2]Komputery z wieloma jednostkami centralnymi przyłączonymi do pamięci współdzielonej (ang. *shared memory.*)

[3]Wiele komputerów połączonych siecią, każdy z własną przestrzenią adresową.

Za pomocą acyklicznych grafów skierowanych możemy analizować zachowanie równoległych algorytmów przy założeniu, że każdy z procesorów ma dostęp do danych obliczonych przez inny procesor bez dodatkowych narzutów. Implementacja algorytmu polega na *planowaniu* wykonania każdego węzła na wybranym procesorze.

Powiedzmy, że dla danych p procesorów, chcemy przyporządkować każdemu węzłowi i parę (j_i, t_i), gdzie $j_i \leqslant p$ oznacza indeks procesora, zaś t_i jednostkę czasu, taką że zachodzą poniższe warunki:

1. Jeśli $t_i = t_k$ dla pewnego $i \neq k$, to $j_i \neq j_k$. Oznacza to, że każdy procesor może wykonać pojedyńczą operację podczas każdej jednostki czasu.

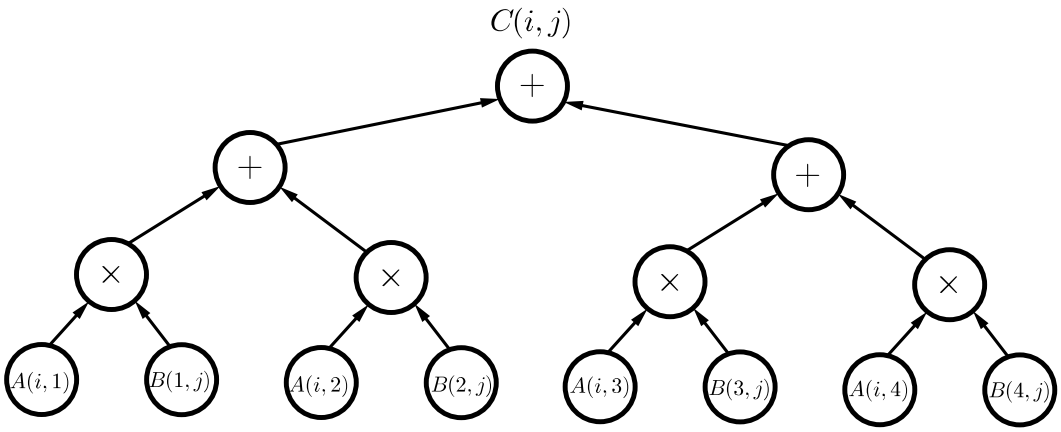
2. Jeśli (i, k) jest łukiem grafu, to $t_k \geqslant t_i + 1$. Oznacza to, że operacja, którą przedstawia węzeł k powinna być zaplanowana po wykonaniu operacji przedstawionej przez węzeł i.

Przyjmuje się, że czas t_i węzła wejściowego i wynosi 0 oraz żaden procesor nie jest przyporządkowany do tego węzła.

Definicja 1.18 (Plan)**.** Ciąg $\{(j_i, t_i) | i \in N\}$ nazywamy **planem** równoległego wykonania DAG przez p procesorów, gdzie N oznacza zbiór węzłów DAG.

Dla dowolnego planu, odpowiadający mu czas wykonania (złożoność czasowa) algorytmu jest określony przez $\max_{i \in N} t_i$. Złożoność równoległa DAG'a jest określona przez $T_p(n) = \min \{\max_{i \in N} t_i\}$, gdzie minimum bierzemy po wszystkich planach, które używają p procesorów.

Przykład 1.2. Niech $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$. Rozważmy standardowy algorytm obliczający iloczyn macierzy $AB = C$. Każdy $C(i, j)$ obliczamy za pomocą wyrażenia $C(i, j) = \sum_{l=1}^{n} A(i, l)B(l, j)$. Odpowiadający obliczeniu DAG dla $n = 4$ przedstawia rys. 1.4. Mając n^3 procesorów, operacje mogą być zaplanowane poziom po poziomie, używając n procesorów do obliczenia każdego z elementów macierzy wynikowej C. Stąd widać, że możemy zaplanować DAG do obliczenia o złożoności $O(\log n)$



Rysunek 1.4: Standardowy iloczyn macierzowy

1.5 Ocena algorytmów

1.5.1 Złożonośc czasowa

Algorytmy sekwencyjne

Ograniczenia zasobów (np. czasu i przestrzeni) wymagane przez algorytmy sekwencyjne mierzymy jako funkcję rozmiaru danych wejściowych $T(n)$, tzw. złożoność czasową. Ograniczenia te wyrażamy asymptotycznie używając notacji:

1. $T(n) = O(f(n))$, jeśli istnieje dodatnie stałe c i n_0 takie, że $\forall n \geqslant n_0 : (T(n) \leqslant cf(n))$
2. $T(n) = \Omega(f(n))$, jeśli istnieje dodatnie stałe c i n_0 takie, że $\forall n \geqslant n_0 : (T(n) \geqslant cf(n))$
3. $T(n) = \Theta(f(n))$, jeśli $T(n) = O(f(n))$ i $T(n) = \Omega(f(n))$

Czas działania algorytmu sekwencyjnego szacuje się przez liczbę operacji podstawowych wymaganych przez algorytm jako funkcję ilości danych wejściowych.

Algorytmy równoległe

Definicja 1.19 (Pesymistyczna złożoność obliczeniowa[4])**.** Załóżmy że algorytm równoległy R rozwiązuje problem P o rozmiarze n. **Pesymityczną**

złożonością czasową algorytmu równoległego R nazywamy funkcję:

$$T_p(n) = \sup_{d \in D_n} \{t(p,d)\}, \tag{1.1}$$

gdzie $t(p,d)$ oznacza liczbę kroków obliczeniowych (operacji dominujących) wykonanych dla zestawu danych d od momentu rozpoczęcia obliczeń algorytmu R przez pierwszy procesor do chwili zakończenia obliczeń przez wszystkie procesory, p – liczbę procesorów, D_n – zbiór wszystkich zestawów danych wejściowych d o rozmiarze n.

1.5.2 Przyspieszenie

Potencjalną korzyść z równoległego wykonania zadania obliczeniowego możemy zmierzyć licząć czas jaki zajmuje wykonanie go na jednym procesorze i porównanie wyniku z wykonaniem tego samego zadania równolegle na N procesorach.

Definicja 1.20 (Przyspieszenie bezwzględne[4])**.** Niech P będzie pewnym zadaniem obliczeniowym, n – rozmiarem danych wejjściowych. Wówczas

$$S_p(n) = \frac{T^*(n)}{T_p(n)(N)} \tag{1.2}$$

gdzie $T^*(n)$ jest pesymistyczną złożonością czasową najszybszego znanego algorytmu sekwencyjnego R_s rozwiązującego problem P na jednym procesorze, $T_n(N)$ jest pesymistyczną złożonością algorytmu R, gdzie R jest równoległą wersją algorytmu R_s. Wyrażenie 1.2 nazywamy **przyspieszeniem bezwzględnym** algorytmu R.

Wniosek 1.1. Zgodnie z definicją 1.19 przez $T_1(n)$ rozumiemy złożoność algorytmu równoległego R wykonywanego przy użyciu jednego procesora. Jeśli algorytm R nie jest najlepszą równoległą wersją znanego algorytmu sekwencyjnego, to równość $T_1(n) = T^*(n)$ nie zachodzi.

Uwaga 1.1. Maksymalną wartością przyspieszenia $S(p,n)$ jest p, ponieważ używając p procesorów można przyspieszyć obliczenia najlepszego algorytmu sekwencyjnego co najwyżej p razy. Zwykle uzyskiwane przyspieszenie jest mniejsze niż p. Przyczyną tego może być niewystarczający stopień zrównoleglenia problemu P, opóźnienia w komunikacji między procesami lub narzut czasu wykonania spowodowane synchronizacją procesów.

Uwaga 1.2. Istnieją problemy dla których najlepszy znany algorytm sekwencyjny R_s nie może zostać zrównoleglony. Wówczas równoległe rozwiązanie problemu w postaci pewnego algorytmu R działa na innej zasadzie. Wówczas pomocne w ocenie korzyści z jest posługiwanie się *przyspieszeniem względnym.*

Definicja 1.21 (Przyspieszenie bezwzględne[4])**.** Niech P będzie pewnym zadaniem obliczeniowym, n – rozmiarem danych wejściowych. Wówczas

$$S_p(n) = \frac{T_1(n)}{T_p(n)(N)} \tag{1.3}$$

gdzie $T_1(n)$ jest pesymistyczną złożonością czasową algorytmu równoległego R rozwiązującego problem P na jednym procesorze, $T_n(N)$ jest pesymistyczną złożonością algorytmu R wykonanego na n procesorach. Wyrażenie 1.2 nazywamy **przyspieszeniem względnym** algorytmu R.

1.5.3 Koszt

Definicja 1.22 (Koszt algorytmu[4])**.** Niech $T_p(n)$ będzie pędzie pesymistyczną złożonością obliczeniową algorytmu R dla p procesorów. Wówczas funkcję

$$C_p(n) = pT_p(n) \tag{1.4}$$

nazywamy kosztem algorytmu R dla p procesorów.

W myśl definicji 1.19 koszt algorytmu możemy rozumieć przez analogię do liczby operacji dominujących wykonanych łącznie przez wszystkie procesory.

Wniosek 1.2. Łatwo widać, że koszt osiąga minimalną wartość $C_1(n) = T^*(n)$ dla najlepszego znanego algorytmu sekwencyjnego. Stąd koszt algorytmu równoległego R jest minimalny wtedy i tylko wtedy, gdy wykonywane są w nim tylko te operacje, które są wykonywane w najlepszym algorytmie sekwencyjnym R_s.

Uwaga 1.3. W praktyce uzyskanie równości kosztów $pT_p(n) = T^*(n)$ wymaga minimalizacji komunikacji między procesorami lub uruchomienia algorytmów na architekturach w których komunikacja odbywa się na tyle szybko, że jej dodatkowe koszty są pomijalne. Różnicę między kosztem wykonania algorytmu równoległego a kosztem wykonania najlepszego algorytmu sekwencyjnego nazywamy *kosztem organizacji obliczeń równoległych.*

Definicja 1.23 (Koszt organizacji obliczeń)**.** Różnicę

$$C_p^O(n) = C_p(n) - T^*(n) = pT_p(n) - T^*(n) \tag{1.5}$$

nazywamy *kosztem ogranizacji obliczeń równoległych* algorytmu R dla problemu P o rozmiarze n

Definicja 1.24 (Koszt optymalny)**.** Mówimy, że koszt algorytmu R jest *optymalny*, jeśli koszt obliczeń równoległych $C_p(n)$ jest asymptotycznie równy minimalnemu kosztowi obliczeń sekwencyjnych $T^*(n)$, czyli:

$$C_p(n) = \Theta(T^*(n)) \tag{1.6}$$

1.5.4 Efektywność

Definicja 1.25 (Efektywność[4])**.** Niech $T_p(n)$ będzie pesymistyczną złożonością czasową algorytmu R dla p procesorów i problemu R o rozmiarze n. Wówczas mamy

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} = \frac{T_1(n)}{C_p(n)} = \frac{S_p(n)}{p} \tag{1.7}$$

Funkcję $E_p(n)$ nazywamy efektywnością wykorzystania procesorów algorytmu R.

1.5.5 Prawo Amdahla

W złożoności $T_p(n)$ można wyróżnić operacje obliczeniowe, które muszą być wykonane sekwencyjnie, $T_1^s(n)$, oraz obliczenia, które mogą być wykonane równolegle, $T_1^r(n)$. Inaczej:

$$T_1(n) = T_1^s(n) + T_1^r(n) \tag{1.8}$$

Zakładając, że obliczenia $T^r(n)$ da się równomiernie rozdzielić między p procesorami, przyspieszenie $S(p, n)$ wynosi wówczas

$$S_p(n) = \frac{T_1(n)}{T_p(n)} \leqslant \frac{T_1^s(n) + T_1^r(n)}{T_1^s(n) + T_1^r(n)/p + T_p^o(n)} \tag{1.9}$$

gdzie $T_p^o(n)$ jest złożonością dodatkową wynikającą z organizacji obliczeń równoległych.

Rozważmy teraz algorytm sekwencyjny o złożoności $T_1(n)$ rozwiązujący zadany problem P o ustalonym rozmiarze n. Niech s oznacza część operacji algorytmu, która musi być wykonana sekwencyjnie, zaś r część operacji, która może być wykonana równolegle. Oznaczmy: $T^s(n) = sT_1(n)$, $T^r(n) = rT_1(n)$, gdzie $s + r = 1$.

Przyspieszenie algorytmu uzyskane po jego zrównolegleniu można wyznaczyć upraszczając wzór (1.9) przez pominięcie złożoności $T_p^o(n)$. Mamy wówczas:

$$
\begin{aligned}
S_p(n) = \frac{T_1(n)}{T_p(n)} \leqslant \\
&\leqslant \frac{T_1^s(n) + T_1^r(n)}{T_1^s(n) + T_1^r(n)/p + T_p^o(n)} \leqslant \\
&\leqslant \frac{sT_1(n) + rT_1(n)}{sT_1(n) + rT_1(n)/p} = \frac{s+r}{s+r/p} = \frac{1}{s+r/p} = \\
&= \left(s + \frac{1-s}{p}\right)^{-1}
\end{aligned}
\tag{1.10}
$$

gdzie s – część obliczeń w algorytmie które muszą być wykonane sekwencyjnie; p – liczba procesorów.
Otrzymany wzór (1.10) nazywamy **prawem Amdahla**.

Definicja 1.26 (Prawo Amdahla[5])**.** Niech s będzie częścią operacji w algorytmie R, która musi być wykonana sekwencyjnie, taką że $0 \leqslant s \leqslant 1$. Wówczas maksymalne przyspieszenie Ψ osiągalne przez komputer równoległy z p procesorami wykonujący algorytm R spełnia nierówność:

$$\Psi_p(n) \leqslant \frac{1}{s + (1-s)/p} \tag{1.11}$$

Uwaga 1.4. Nierówność z definicji 1.26 służy do wyznaczania górnego ograniczenia przyspieszenia będącego funkcją wielkości s oraz liczby procesorów p przy ustalonym rozmiarze problemu n.

Przykład 1.3. Przypuśćmy, że dysponujemy sześciordzeniowym procesorem i chcemy ocenić czy warto szukać równoległej wersji programu dla rozwiązania pewnego problemu. Ustaliliśmy, że 90% czasu wykonania programu przeznacza się na wykonanie pewnej funkcji, którą chcemy zrównoleglić. Pozostałe 10% czasu wykonania zajmują funkcję, które musimy wykonywać na jednym procesorze. Chcemy ocenić największe przyspieszenie jakiego możemy się spodziewać po równoległej wersji naszego programu. W tym celu możemy skorzystać z prawa Amdahla. Mamy:

$$S_p(n) \leqslant \frac{1}{0.1 + (1-0.1)/6} = 4.$$

Powinniśmy się zatem spodziewać przyspieszenia o wartości co najwyżej 4.

Wniosek 1.3. Przechodząć z wyrażeniem (1.10) do granicy $p \to \infty$ mamy $\lim_{p\to\infty} \frac{1}{s+(1-s)/p} = \frac{1}{s}$. Widać, że maksymalne przyspieszenie $S_p(n)$, jakie można osiągnać nie zależy od liczby użytch procesorów p, ale od ilości obliczeń sekwencyjnych s (pomijając dodatkowe koszty ogranizacji obliczeń).

Przykład 1.4. Przypuśćmy, że 25% operacji w algorytmie równoległym musi być wykonanych równolegle. Wówczas maksymalne osiągalne przyspieszenie przy przeprowadzaniu obliczeń na coraz większej liczbie procesorów wynosi:

$$\lim_{p\to\infty} \frac{1}{0.25 + (1 - 0.25)/p} = 4.$$

1.5.6 Prawo Gustafsona i Barsisa

Niech p oznacza liczbę procesorów, σ – część czasu obliczeń algorytmu równoległego przypadającą na wykonanie obliczeń w sposób sekwencyjny, a ρ – część czasu obliczeń algorytmu równoległego przypadającą na wykonywanie obliczeń w sposób równoległy takie, że $\sigma + \rho = 1$. Czas wykonania tego samego algorytmu w hipotetycznym komputerze sekwencyjnym jest proporcjonalny do sumy $\sigma + p\rho$, gdzie wyrażenie $p\rho$ odpowiada czasowi wykonania części równoległej obliczeń przez jeden procesor. Przyspieszenie, które zostałoby uzyskane, gdyby obliczenia równoległe zostały przeprowadzone w komputerze sekwencyjnym wyraża się przez:

$$\Psi_p(n) \leqslant \frac{\sigma + p\rho}{\sigma + \rho} = \sigma + p\rho = \sigma + p\,(1 - \sigma) = p + (1 - p)\,\sigma \tag{1.12}$$

Wzór (1.12) jest znany jako **prawo Gustafsona i Barsisa**.

Definicja 1.27 (Prawo Gustafsona i Barsisa[5])**.** Dla danego algorytmu R rozwiązującego problem P ustalonego rozmiaru n na p procesorach oznaczmy przez σ część całkowitego czasu wykonania algorytmu. Wówczas maksymalne przyspieszenie Ψ algorytmu R spełnia nierówność:

$$\Psi_p(n) \leqslant p + (1 - p)\sigma$$

Uwaga 1.5. Prawo Gustafsona i Barsisa określa tzw. **skalowane przyspieszenie**, ponieważ wraz ze zmianą liczby procesorów skaluje się odpowiednio rozmiar problemu, tak aby utrzymać stały czas obliczeń równoległych (z założenia $\sigma + \rho = 1$[4].

Przykład 1.5. Powiedzmy, że pewien program wykonany na 64 procesorach wykonuje się w czasie 220 sekund. Testy pokazują, że 5% czasu wykonania

przeznaczone jest na obliczenia sekwencyjne. Skalowane przyspieszenie tego programu wynosi:

$$\Psi_{64}(n) \leqslant 64 + (1 - 64)(0.05) = 64 - 3,15 = 60,85.$$

1.5.7 Miara Karpa-Flatta

Przyspieszenia uzyskiwane ze wzorów (1.10) i (1.12) nie uwzględniają złożoności $T_p^O(n)$ związanej z prowadzeniem obliczeń i dlatego ich wartości są większe niż uzyskiwane doświadczalnie.

Zgodnie z wyrażeniem 1.9 czas wykonywania algorytmu równoległego jest równy:

$$T_p(n) = T_1^s(n) + T_p^0(n) + \frac{T_1^r(n)}{p} \tag{1.13}$$

Jeśli przez f oznaczymy część operacji algorytmu, których nie można zrównoleglić (część *inherentnie sekwencyjną*) oraz złożoność dodatkową wynikającą z organizacji obliczeń, to mamy:

$$f = \frac{T_1^s(n) + T_p^O(n)}{T_1(n)} \tag{1.14}$$

Z (1.13) i (1.14):

$$T_p(n) = T_1^s(n) + T_p^O(n) + \frac{T_1^r(n)}{p} = fT_1(n) + \frac{(1-f)T_1(n)}{p} \tag{1.15}$$

Dzieląć obie strony równania przez $T_1(n)$ otrzymujemy

$$f = \frac{\frac{1}{S_p(n)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{1.16}$$

Wyrażenie (1.16) nazywamy **miarą Karpa-Flatta**.

Definicja 1.28 (Miara Karpa-Flatta)**.** Dla danego algorytmu równoległego R rozwiązującego problem P o rozmiarze n przy pomocy $p > 1$ procesorów, doświadczalnie wyznaczona część sekwencyjna obliczeń f wyraża się przez

$$f = \frac{1/S_p(n) - 1/p}{1 - 1/p} \tag{1.17}$$

Przykład 1.6. Powiedzmy, że testując algorytm równoległy na $1, 2, \ldots, 8$ procesorach otrzymaliśmy pewne przyspieszenia w zależności od ilości procesów i z wyrażenia 1.16 obliczyliśmy eksperymentalnie wyznaczoną część sekwencyjną f. Dane zebrane są w tablicy 1.1.

p	2	3	4	5	6	7	8
$S_p(n)$	$1,82$	$2,50$	$3,08$	$3,57$	$4,00$	$4,38$	$4,71$
f	$0,10$	$0,10$	$0,10$	$0,10$	$0,10$	$0,10$	$0,10$

Tabela 1.1: Dane dla przykładu 1.6.

Ponieważ eksperymentalnie wyznaczona część sekwencyjna algorytmu nie rośnie wraz z ilością procesorów, możemy wnioskować, że zbyt duża część obliczeń jest inherentnie sekwencyjna.

Przykład 1.7. Powiedzmy, że testując algorytm równoległy na $1, 2, \ldots, 8$ procesorach otrzymaliśmy pewne przyspieszenia w zależności od ilości procesów i z wyrażenia 1.16 obliczyliśmy eksperymentalnie wyznaczoną część sekwencyjną f. Dane zebrane są w tablicy 1.2.

p	2	3	4	5	6	7	8
$S_p(n)$	$1,87$	$2,61$	$3,23$	$3,73$	$4,14$	$4,46$	$4,71$
f	$0,070$	$0,075$	$0,080$	$0,085$	$0,090$	$0,095$	$0,10$

Tabela 1.2: Dane dla przykładu 1.7.

Ponieważ eksperymentalnie wyznaczona część sekwencyjna algorytmu rośnie wraz z ilością procesorów, możemy wnioskować, że przyczyną niskiego przyspieszenia jest organizacja obliczeń równoległych, tj. czas poświęcony uruchomieniu procesów, komunikacji między nimi, synchronizacji lub ograniczenia samej architektury.

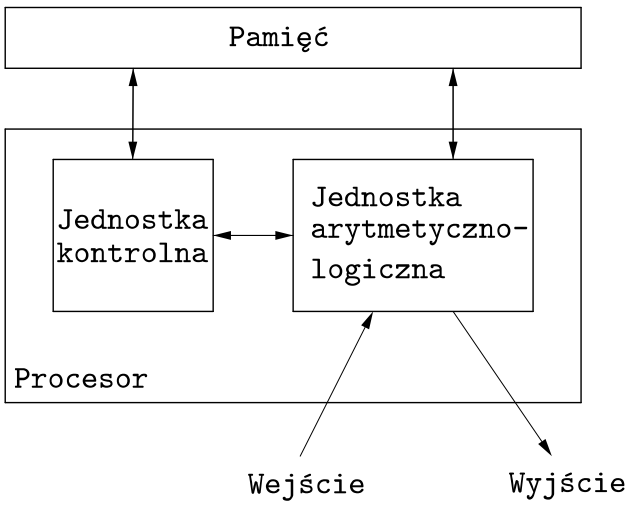
1.6 Teoretyczne modele obliczeń

1.6.1 Model RAM

Nim przejdziemy do omówienia modeli obliczeń równoległych zajmiemy się omówieniem modelu RAM.

Model RAM (*Random Access Machine*) odpowiada rozważaniom zawartym w 1.5.1. Zakłada on:

1. Istnienie pewnego procesora wyposażonego w:
   (a) skończoną listę instrukcji, które może on realizować.
   (b) pewną liczbę rejestrów arytmetycznych procesora $R_1, R_2, \ldots, R_n$, $n > 1$ które mogą przechowywać dowolne skończone liczby w zapisie binarnym.
   (c) specjalny rejestr sterujący L zwany licznikiem programu.
2. Istnienie pamięci złożonej z potencjalnie nieskończonej liczby komórek M_i, $i = 1, 2, 3, \ldots$ (Rys. 1.5) w których można przechowywać dowolną skończoną liczbę w zapisie binarnym.
3. Stały czas zapisu i odczytu wartości do/z komórki pamięci (inaczej *dostęp swobodny*).



Rysunek 1.5: Model obliczeń sekwencyjnych RAM – architektura Von Neumanna

Tabela 1.3: Przykładowa lista instrukcji procesora[4]

Instrukcja	Argument	Znaczenie
`LOAD`	k, a	$R_k := w(a)$
`STORE`	k, b	$M_{w(b)} := R_k$
`ADD`	k, c	$R_k := R_k + w(c)$
`SUB`	k, c	$R_k := R_k - w(c)$
`MULT`	k, c	$R_k := R_k \times w(c)$
`DIV`	k, c	$R_k := \lfloor R_k / w(c) \rfloor$
`JUMP`	i	$L := i$
`JPOS`	k, i	`if` $R_k > 0$ `then` $L := i$ `else` $L := L + 1$
`JZERO`	k, i	`if` $R_k == 0$ `then` $L := i$ `else` $L := L + 1$
`JNEG`	k, i	`if` $R_k < 0$ `then` $L := i$ `else` $L := L + 1$
`READ`	k	Wczytaj daną z urządzenia zewnętrznego do rejestru R_k
`WRITE`	k	Wydrukuj daną z rejestru R_k
`HALT`		Zakończ obliczenie

1.6.2 Model PRAM

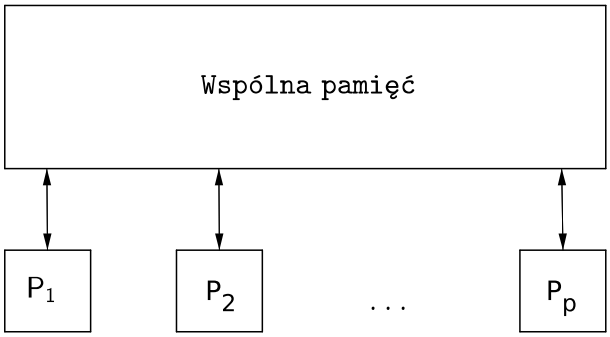
Model wspólnej pamięci składa się z pewnej liczby procesorów, z których każdy posiada własną pamięć i może lokalnie wykonywać programy. Wszystkie procesory mogą komunikować się za pomocą wspólnej globalnej pamięci (rys. 1.6).

Definicja 1.29 (Złożoność komunikacyjna[4])**.** Maksymalny rozmiar danych przesłanych w trakcie wykonywania algorytmu PRAM między pamięcią wspólną, a pamięcią lokalną dowolnego procesora nazywamy *złożonością komunikacyjną*.

Każdemu procesorowi przypożądkowana jest niepowtarzająca się liczba naturalna. Jest to lokalnie dostępny indeks, numer procesora lub jego identyfikator.

W modelu wspólnej pamięci wyróżniamy dwa podstawowe tryby operacji.

- Tryb synchroniczny. Wszystkie procesory działają synchronicznie według wspólnego zegara. Model ten nazywamy równoległą maszyną o dostepie swobodnym (PRAM, parallel random-access machine).



Rysunek 1.6: Model wspólnej pamięci

- Tryb asynchroniczny. Każdy procesor pracuje według osobnego zegara. W tym trybie programista jest odpowiedzialny za odpowiednią synchronizację procesorów, jeśli zachodzi taka potrzeba. Dokładniej mówiąc, jeśli procesor ma pobrać dane, to odpowiedzialnością programisty jest upewnienie się, że odpowiednie dane są już uzyskane, ponieważ wartości wspólnych zmiennych są określane dynamicznie w trakcie wykonania programu na różnych procesorach.

Ponieważ każdy procesor może uruchomić swój program lokalnie, ten model jest typu MIMD w klasyfikacji Flynna. Znaczy to tyle, że każdy procesor może wykonać pewną instrukcję lub operację na danych niezależnie od tych wykonanych na jakimkolwiek innym procesorze w trakcie danej jednostki czasu.

Dla danego algorytmu, rozmiar danych wymienionych pomiędzy pamięcią globalną i pamięcią lokalną różnych procesorów wyraża rozmiar **komunikacji** wymaganej przez algorytm.

Możemy wyróżnić kilka wariantów modelu PRAM w zależności od wymagań jakie postawimy odnośnie jednoczesnego dostępu kilku procesorów do tego samego adresu w pamięci globalnej.

- **EREW** – algorytmy z wyłącznym odczytem i wyłącznym zapisem; nie pozwala na jednoczesny zapis do pamieci.
- **CREW** – algorytmy z jednoczesnym odczytem i wyłącznym zapisem; pozwala na jednoczesny dostęp do pamięci dla instrukcji odczytu.
- **CRCW** – algorytmy z jednoczesnym odczytem i jednoczesnym zapisem.
- **ERCW** – algorytmy z wyłącznym odczytem i jednoczesnym zapisem.

Jeśli nie poczyni się żadnych dodatkowych założeń, to nie jest jasno określone, co zostanie zapisane w komórce pamięci w wyniku jednoczesnego zapisywania do niej przez wiele procesorów w algorytmie typu CRCW. W literaturze można spotkać wiele typów maszyny PRAM, które różnią się sposobami rozwiązywania konfliktów zapisu. Można wśród nich wyróżnić[6]:

1. jednolity (ang. *common*) – procesory muszą zapisać do tej samej komórki pamięci jednolitą wartość.

2. dowolny (ang. *arbitrary*) – zapamiętywana jest dowolna wartość z wartości zapisywanych do tej samej komórki pamięci.

3. priorytetowy (ang. *priority*) – zapamiętywana jest wartość zapisywana przez procesor o najmniejszym numerze.

4. mieszany (ang. *combining*) – zapamiętywana wartość jest pewną, jednak ściśle określoną kombinacją zapisywanych wartości.

1.6.3 Model sieciowy

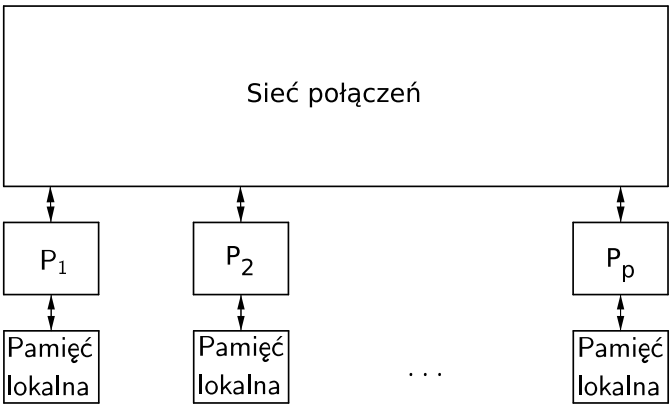
W sieciowym modelu obliczeń przyjmujemy, że każdy węzeł wyposażony jest w pamięć lokalną i wykonuje własne operacje niezależnie od pozostałych. Węzeł może przetwarzać dane w swojej pamięci lokalnej oraz wysyłać i odbierać dane od innych węzłów sieci w postaci *komunikatów*. Pamięć współdzielona przez węzły (rys. 1.7) nie istnieje w modelu sieciowym.

Do każdego węzła sieci przypisany jest numer identyfikacyjny (*pid*). μ-ty węzeł oznaczamy przez P_μ. Powiemy, że P_λ jest *sąsiadem* P_μ, jeśli istnieje bezpośrednie fizyczne połączenie między nimi.[7]

Sposób łączenia układu jednostek obliczeniowych (procesorów) w sieć nazywamy *topologią sieci*. Topologię sieci można przedstawić jako graf $G = (N, E)$, gdzie każdy węzeł $i \in N$ oznacza jednostkę obliczeniową, a każda krawędź $(i, j) \in E$ – dwukierunkową komunikację między jednostkami obliczeniowymi i i j.

Tak jak w przypadku modelu z pamięcią wspólną, operacje w sieci mogą być synchroniczne lub asynchroniczne. Procesory pracujące w sieci asynchronicznej zarządzają swoimi zadaniami przez wymianę komunikatów. Schemat taki nazywamy *modelem wymiany komunikatów*. Procesory te niekoniecznie muszą być ze sobą sąsiadujące.

Charakteryzuje ją kilka parametrów:



Rysunek 1.7: Model sieciowy

1. średnica (ang. *diameter*) – maksymalna odległość (krawędziowa) między dowolną parą węzłów; im miejsza, tym lepiej, ponieważ w najgorszym przypadku wiadomość musi zostać przesłana przez liczbę łączy równą średnicy sieci

2. maksymalny stopień wierzchołka – maksymalna liczba łączy do danego węzła

3. szerokość połowienia sieci (ang. *bisection width*) – minimalna liczba krawędzi, które muszą zostać usunięte, aby podzielić ją na dwie równe podsieci z dokładnością do jednego węzła

4. spójność krawędziowa – minimalna liczba krawędzi, które muszą ulec awarii, aby sieć stała się niespójna

5. koszt sieci – koszt wykonania, zarządzania i utrzymania połączeń między węzłami; w najprostrzym przypadku mierzony liczbą krawędzi

W opisie algorytmów mnożenia macierzy dla modelu sieciowego potrzebujemy zdefiniować dwie instrukcje odnoszące się do wysyłania i odbierania komunikatów:

1. `send(macierz, pid węzła odbierającego)`

2. `recv(macierz, pid węzła wysyłającego)`

Jeśli P_μ wykonuje instrukcję `send(`V_{loc}`,` λ`)`, wówczas wysyłana jest kopia macierzy V_{loc} z pamięci lokalnej P_μ do węzła P_λ i wykonanie kolejnych instrukcji

P_μ jest natychmiast kontynuowane [4]. Węzeł może również wysłać komunikaty do samego siebie.

Jeśli P_μ wykona instrukcję `recv(`U_{loc}`, `λ`)`, wówczas wykonywanie kolejnych instrukcji na tym węźle jest zatrzymane dopóki komunikat od P_λ nie zostanie odebrany[5]. Jeśli odbieranie jest zakończone, komunikat jest zapisywany w pamięci lokalnej, a węzeł P_μ kontynuuje wykonywane programu.

Uwaga 1.6. Zaproponowana notacja nie uwzględnia pewnych istotnych detali:

Składanie danych. Często komunikat składa się z danych, które nie stanowią ciągłego obszaru w lokalnej pamięci węzła. Wówczas spotykamy się z dodatkowymi nakładami czasu wykonania.

Etykietowanie. Komunikaty nie muszą być dostarczane w takiej kolejności, w jakiej są wysyłane. Wówczas pojawia się konieczność etykietowania ich w taki sposób, żeby węzeł odbierający mógł jednoznacznie określić który komunikat ma w danej chwili odebrać. W zaproponowanym modelu przyjmujemy, że komunikaty są dostarczane w kolejności takiej, w jakiej są wysyłane.

Interpretacja. W praktyce przejście od komunikatu zawierającego macierz do zapisania macierzy w pamięci lokalnej zabiera pewien czas. Jest tak ze względu na czas potrzebny na interpretację informacji o wymiarach macierzy lub typ danych.

Wybrane typy danych dla obliczeń rozproszonych

Niech $x \in \mathbb{R}^n$ będzie rozdystrybuowane między pamięci lokalne sieci składającej się z p węzłów. Załóżmy roboczo, że $n = rp$. Możemy wyróżnić dwa najczęstsze podejścia do reprezentacji wektora x w sieci: zapis kolumnowy (ang. *store-by-column*) i zapis wierszowy (ang. *store-by-row*).

W pierwszym z nich, zapisie kolumnowym, rozpatrujemy wektor x jako macierz $r \times p$

$$x_{r\times p} = [x(1:r) \quad x(r+1:2r) \ldots x(1+(p-1)r:n)]\,,$$

Każda *kolumna* zapisana jest w osobnym węźle, tj. $x(1+(\mu-1)r\colon \mu r) \in P_\mu$. (W tym kontekście predykat „$x \in y$" oznacza „x jest zapisany w y.") Zauważmy, że każdy węzeł zawiera *ciągłą* część wektora x.

[4] Dla interfejsu MPI odpowiednikiem jest nieblokująca funkcja `MPI_Isend`.

[5] W interfejsie MPI odpowiednikiem tej funkcji jest `MPI_Recv`.

W zapisie wierszowym x traktujemy jako macierz wymiaru $p \times r$

$$x_{p\times r} = [x(1:p) \quad x(p+1:2p) \ldots x((r-1)p:n)]\,,$$

Każdy *wiersz* jest wówczas zapisany w odpowiednim węźle, tj. $x(\mu\colon p\colon n) \in P_\mu$. Podejście to przypomina *rozdawanie* (ang. *wrap method*) danych węzłom sieci przez analogię do rozdawania kart graczom przy stole.

Jeśli n nie jest wielokrotnością p wówczas powyższe podejścia stosuje się z niewielką modyfikacją. Rozważmy zapis kolumnowy dla $n = 14$ i $p = 4$:

$$x^r = [\underbrace{x_1x_2x_3x_4}_{P_0} \,|\, \underbrace{x_5x_6x_7x_8}_{P_1} \,|\, \underbrace{x_9x_{10}x_{11}}_{P_2} \,|\, \underbrace{x_{12}x_{13}x_{14}}_{P_3}] \tag{1.18}$$

Ogólniej, jeśli $n = pr + q$, gdzie $0 \leqslant q < p - 1$, to $P_0, P_1, \ldots, P_q$ mogą zgromadzić po $r + 1$ elementów, zaś $P_{q+1}, P_{q+2}, \ldots, P_{p-1}$ — r elementów. Metoda wierszowa pozwala zgromadzić węzłowi P_μ wektor $x(\mu\colon p\colon n)$.

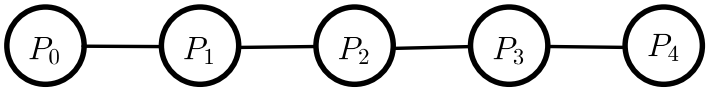
W podobny sposób możemy podejść do dystrybucji macierzy. Jeśli $A \in \mathbb{R}^{\kappa\times\kappa}$ i $n = rp$ możemy wyróżnić cztery podejścia:

Tabela 1.4: Sposoby reprezentacji macierzy w sieci z q węzłami

Orientacja	Styl	Zawartość węzła
Kolumnowy	Ciągły	$A(:\,,\ 1 + (\mu - 1)r\colon \mu r)$
Kolumnowy	Rozdawany	$A(:\,,\ \mu\colon p\colon n)$
Wierszowy	Ciągły	$A(1 + (\mu - 1)r\colon \mu r,\ :\,)$
Wierszowy	Rozdawany	$A(\mu\colon p\colon n,\ :\,)$

Metody dla macierzy blokowych są analogiczne do tych z tabeli 1.4.

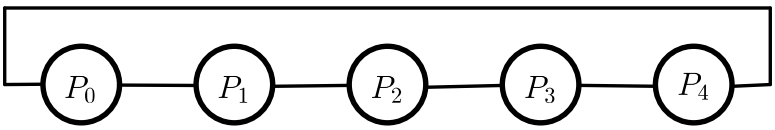
Przykłady



Rysunek 1.8: Siatka jednowymiarowa (liniowa)

Przykład 1.8 (Sieć liniowa)**.** Model składa się z p węzłów $P_1, P_2, \ldots, P_p$ połączonych ze sobą w ciąg, tj. węzeł P_i połączony jest z węzłem P_{i-1} i P_{i+1}, o ile takie istnieją (rys. 1.8). Średnica takiej sieci wynosi $p - 1$, jej maksymalny stopień wynosi 2.

Przykład 1.9 (Torus)**.** Siecią w topologii torusa nazywamy siatkę liniowa z połączonymi końcami (rys. 1.9).

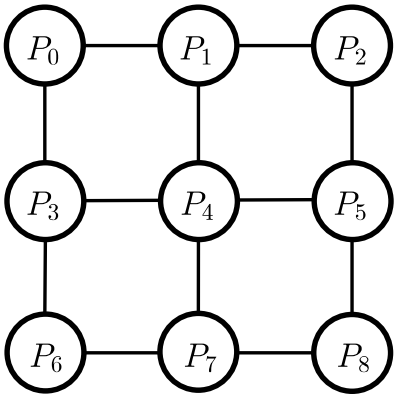


Rysunek 1.9: Torus jednowymiarowy

Tabela 1.5: Wybrane parametry charakteryzujące przykładowe sieci połączeń

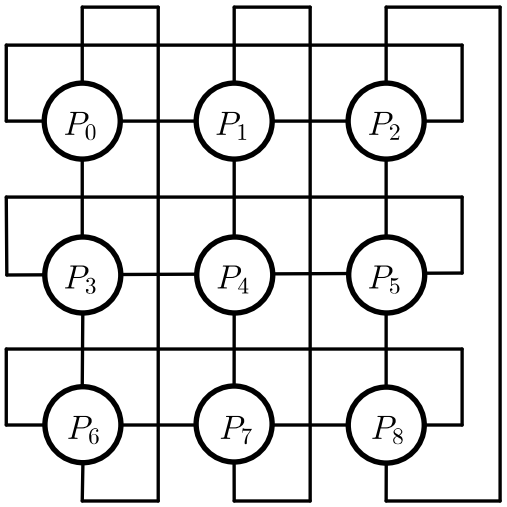
Rodzaj sieci	Średnica	Maksymalny stopień	Szerokośc połowienia	Spójność krawędziowa	Koszt
Siatka jednowymiarowa	$p-1$	2	1	1	$p-1$
Torus jednowymiarowy	$\lfloor p/2 \rfloor$	2	2	2	p
Siatka dwuwymiarowa	$2\sqrt{p}-1$	4	$\sqrt{p}$	2	$2(p-\sqrt{p})$
Torus dwuwymiarowy	$2\lfloor \sqrt{p}/2 \rfloor$	4	$2\sqrt{p}$	4	$2p$
Hipersześcian	$\log p$	$\log p$	$p/2$	$\log p$	$\frac{p \log p}{2}$

Przykład 1.10 (Siatka dwuwymiarowa)**.** Dwuwymiarowa siatka jest dwuwymiarową wersją sieci liniowej. Składa się ona z $p = m^2$ procesorów ułożonych w siatkę $m \times m$ taką, że procesor $P_{i,j}$ jest połączony z procesorem $P_{i\pm1,j}$ i $P_{i,j\pm1}$. Średnica takiej sieci złożonej z $p = m^2$ procesorów wynos $\sqrt{p}$ a jej maksymalny stopień 4



Rysunek 1.10: Siatka dwuwymiarowa

Przykład 1.11 (Dwuwymiarowy torus)**.** Siatka, której skrajne procesory są połączone ze sobą *ang.wraparound connections.*



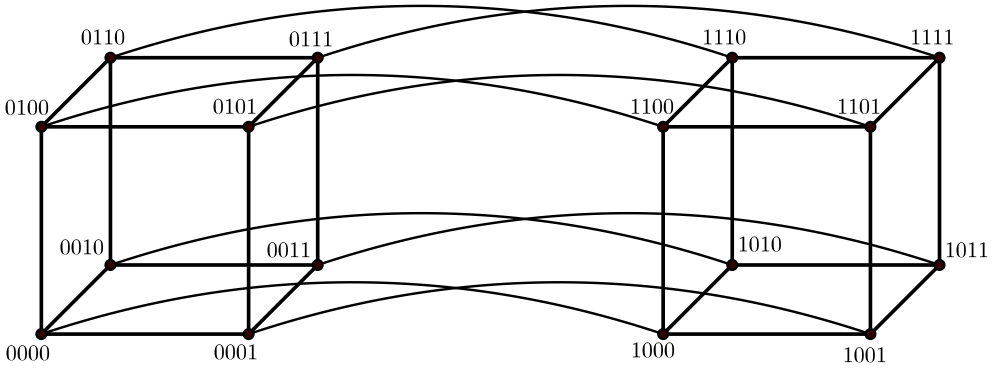
Rysunek 1.11: Torus dwuwymiarowy 3×3

Definicja 1.30 (Kostka Boola)**.** Niech $i_{d-1}i_{d-2}\ldots i_0$, gdzie $0 \leqslant i \leqslant p-1$ będzie binarną reprezentacją i. Wówczas węzeł i jest połaczony z węzłem $P_{i^{(j)}}$, gdzie $i^{(j)} = i_{d-1}\ldots\overline{i_j}\ldots i_0$ i $\overline{i_j} = 1 - i_j$. Innymi słowy, dwa węzły są ze sobą połączone wtedy i tylko wtedy, gdy ich wskaźniki różnią się tylko jednym bitem.

Przykład 1.12 (Siatka hipersześcienna)**.** Sieć w topologii hipersześcianu składa się z $p = 2^d$ węzłów połączonych w d-wymiarową kostkę Boola.

Hipersześcian ma strukurę rekursywną. Kostkę d-wymiarową możemy rozszerzyć do $d+1$ wymiarów przez połączenie poszczególnych węzłów do d-wymiarowych kostek.

Średnica d-wymiarowego hipersześcianu wynosi $d = \log p$. Jest tak ponieważ odległośc w grafie między dwoma węzłami P_i i P_j jest równa liczbie pozycji bitów, którymi wskaźniki i i j różnią się między sobą. Stąd jest ona mniejsza lub równa d, a ponadto odległość między P_0 a P_{2^d-1} wynosi d. Każdy węzeł jest stopnia $d = \log p$.



Rysunek 1.12: Sieć w topologii hipersześcianu

Rozdział 2

Równoległe algorytmy mnożenia macierzy

2.1 Przegląd algorytmów klasycznych

W swojej pracy *„Gaussian Elimination is not Optimal”* z 1969 roku Volker Strassen pokazał rekursyjny algorytm mnożenia macierzy kwadratowych wymiaru $m2^k$ o złożoności $\mathcal{O}(n^{2,81})$[8]. W przypadku macierzy 2×2 oznaczało to, że mnożenie można wykonać już za pomocą 7 mnożeń i 18 dodawań. Algorytm za sprawą Shmuela Winograda został zoptymalizowany[9][10] do najczęściej implementowanej dzisiaj postaci algorytmu Strassena-Winograda (implementacja zawiera się na przykład w bibliotece GEMMW[11]). W przypadku macierzy 2×2 wykonuje on 7 operacji mnożenia i 15 dodawań[12].

Algorytm naiwny

Niech $\mathbf{A}$, $\mathbf{B} \in \mathbb{R}^{n \times n}$. Rozważmy algorytm sekwencyjny wyznaczania macierzy $\mathbf{C} = \mathbf{AB}$ o złożoności $\mathcal{O}(n^3)$.

Algorytm 1 Sekwencyjny algorytm mnożenia macierzy.

```
for i ← 1, n do
    for j ← 1, n do
        for k ← 1, n do
            C(i, j) = C(i, j) + A(i, k)B(k, j)
        end for
    end for
end for
```

Algorytm „dziel i rządź”

Dla danych macierzy wejściowych $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$ oraz macierzy wyjściowej $\mathbf{C} \in \mathbb{R}^{n \times n}$ mamy:

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

gdzie

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{2.1}$$

(2.1) możemy wyrazić inaczej:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{11} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{2.2}$$

Z powyższych konstatacji nasuwa się łatwy algorytm rekurencyjny.

Algorytm 2 Algorytm „dziel i zwyciężaj”.

```
1: function MULTIPLY(A, B)
2:     if n = 1 then
3:         C_1 1 ← A_11 B_11
4:     else
5:         C_11 = MULTIPLY(A_11 B_11) + MULTIPLY(A_12 B_21)
6:         C_12 = MULTIPLY(A_11 B_12) + MULTIPLY(A_12 B_22)
7:         C_21 = MULTIPLY(A_21 B_11) + MULTIPLY(A_22 B_21)
8:         C_22 = MULTIPLY(A_21 B_12) + MULTIPLY(A_22 B_22)
9:     end if
10:    return C
11: end function
```

Algorytm Strassena

Niech A i B będą macierzami $m2^k \times m2^k$. Definiując następujące macierze pomocnicze

$$\begin{aligned} H_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) & H_2 &= (A_{21} + A_{22})B_{11} \\ H_3 &= A_{11}(B_{12} + A_{22}) & H_4 &= A_{22}(B_{21} + A_{11}) \\ H_5 &= (A_{11} + A_{12})B_{22} & H_6 &= (A_{21} + A_{11})(B_{11} + B_{12}) \\ H_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

otrzymujemy

$$C = \begin{bmatrix} H_1 + H_4 - H_5 + H_7 & H_3 + H_5 \\ H_2 + H_4 & H_1 + H_3 - H_2 + H_6 \end{bmatrix} \tag{2.3}$$

Algorytm Strassena-Winograda

Dla danych macierzy wejściowych $\mathbf{A}, \mathbf{B}$ oraz macierzy wyjściowej $\mathbf{C}$ mamy

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Następnie ustalmy odpowiednio po siedem kombinacji liniowych T_i, S_i, $i \in \{1, 2, \ldots, 7\}$ dla każdej z podmacierzy $\mathbf{A}$ i $\mathbf{B}$.

$$\begin{aligned} T_0 &= A_{11} & S_0 &= B_{11} \\ T_1 &= A_{12} & S_1 &= B_{21} \\ T_2 &= A_{21} + A_{22} & S_2 &= B_{12} + B_{11} \\ T_3 &= T_2 - A_{12} & S_3 &= B_{22} - S_2 \\ T_4 &= A_{11} - A_{21} & S_4 &= B_{22} - B_{12} \\ T_5 &= A_{12} + T_3 & S_5 &= B_{22} \\ T_6 &= A_{22} & S_6 &= S_3 - B_{21} \end{aligned}$$

oraz

$$
\begin{aligned}
Q_0 &= T_0 S_0 & U_1 &= Q_0 + Q_3 \\
Q_1 &= T_1 S_1 & U_2 &= U_1 + Q_4 \\
Q_2 &= T_2 S_2 & U_3 &= U_1 + Q_2 \\
Q_3 &= T_3 S_3 & C_{11} &= Q_0 + Q_1 \\
Q_4 &= T_4 S_4 & C_{12} &= U_3 + Q_5 \\
Q_5 &= T_5 S_5 & C_{21} &= U_2 - Q_6 \\
Q_6 &= T_6 S_6 & C_{22} &= U_2 + Q_2
\end{aligned}
$$

Jest to jeden krok metody Strassena-Winograda. Algorytm jest rekursywny ponieważ może być użyty ponownie dla wyznaczenia Q_i, $i \in \{0, 1, \ldots, 6\}$

W praktyce stosuje się tylko kilka kroków algorytmu Strassena-Winegrada[12]. Złożoność obliczeniową $O(n^{w_0})$ algortytmu oznacza, że jego wykonanie zatrzymuje się po osiągnięciu macierzy wymiaru 1×1.

2.2 Algorytm w modelu PRAM

Rozważmy problem obliczenia iloczynu $\mathbf{C}$ dwóch macierzy $\mathbf{A}$, $\mathbf{B} \in \mathbb{R}^{n \times n}$, gdzie $n = 2^r$, dla pewnego $r \in \mathbb{Z}$, $r > 0$. Załóżmy, że dysponujemy n^3 procesorami $P_{i,j,l}$, $1 \leqslant i, j, l \leqslant n$ maszyny CREW PRAM.

Algorytm 3 Algorytm mnożenia macierzy dla n^3 procesorów.[4] (cz. I)

Wejście: Macierze A i B umieszczone w pamięci wspólnej modelu CREW PRAM o n^3 procesorach; zmienne lokalne służące do przechowywania rozmiaru n, gdzie ($n = 2^r$ dla pewnego $0 < r \in \mathbb{Z}$); numer procesora w postaci zmiennych i, j oraz k.

Dane pomocnicze: Macierz T wymiaru $n \times n \times n$ umieszczona w pamięci wspólnej; zmienna lokalna l

Wyjście: Iloczyn macierzy $C = AB$ w pamięci współdzielonej.

1: **parfor** $P_{i,j,k}$, $1 \leqslant i, j, k \leqslant n$ **do**
2: $\quad T_{i,j,k} \leftarrow A_{i,k} B_{k,j}$ ▷ Obliczanie składowych iloczynów skalarnych
3: **end parfor**
4: **for** $l \leftarrow 1, \log n$ **do** ▷ Sumowanie składowych iloczynów skalarnych
5: $\quad$ **parfor** $P_{i,j,k}$, $1 \leqslant i, j \leqslant n$, $1 \leqslant k \leqslant n/2^l$ **do**

Algorytm 3 Algorytm mnożenia macierzy dla n^3 procesorów.[4] (cz. II)

6: $T_{i,j,k} \leftarrow T_{i,j,2k-1} + T_{i,j,2k}$
7: **end parfor**
8: **end for**
9: **parfor** $P_{i,j,k}$, $1 \leqslant i, j \leqslant n$, $k = 1$ **do**
10: $C_{i,j} \leftarrow T_{i,j,1}$
11: **end parfor**

Uwaga 2.1. Algorytm 3 wymaga równoległego odczytu ponieważ w trakcie wykonania kroku (3) procesory $P_{i,l,k}$ mogą równocześnie odczytywać te same dane. Przykładowo procesory $P_{i,1,l}, P_{i,2,l}, \ldots, P_{i,n,l}$ w trakcie wykonywania kroku (3) wszystkie wymagają dostępu do elementu A_{il}.

Obliczenia w wierszach (2-4) i (10-12) wykonywane są w czasie $\mathcal{O}(1)$, a sumowanie składowych iloczynów skalarnych w wierszach (5-9) – w czasie $\mathcal{O}(\log n)$. Złożoność, przyspieszenie, koszt i efektywność algorytmu są następujące[4]:

$$\begin{aligned} T_p(n)|_{p=n^3} = T(n) &= \mathcal{O}(\log n), \\ S(n) &= \mathcal{O}(\frac{n^3}{\log n}), \\ C(n) &= \mathcal{O}(n^3 \log n), \\ E(n) &= \mathcal{O}(\frac{n^3}{n^3 \log n}) = \mathcal{O}(\frac{1}{\log n}. \end{aligned}$$

Zgodnie z def. 1.24 algorytm *nie* jest optymalny względem kosztu. Zaprezentujemy poniżej algorytm 5 będący modyfikacją powyższego algorytmu wykonaną na n procesorach. Jak pokażemy poźniej, jego koszt jest optymalny.

Algorytm 4 Algorytm mnożenia macierzy dla n procesorów.[4] (cz. I)

Wejście: Macierze A i B umieszczone w pamięci wspólnej modelu CREW PRAM o n^3 procesorach; zmienne lokalne służące do przechowywania rozmiaru n, gdzie ($n = 2^r$ dla pewnego $0 < r \in \mathbb{Z}$); numer procesora w postaci zmiennych i, j oraz k.

Dane pomocnicze: Macierz T wymiaru $n \times n \times n$; zmienna lokalna l.

Wyjście: Iloczyn macierzy $C = AB$ w pamięci współdzielonej.

1: **parfor** $P_i, 1 \leqslant i \leqslant n$ **do**
2: **for** $j \leftarrow 1, n$ **do**
3: **for** $k \leftarrow 1, n$ **do**

Algorytm 5 Algorytm mnożenia macierzy dla n procesorów.[4] (cz. II)

```
4:              T_{i,j,k} ← A_{i,k}B_{k,j}  ▷ Obliczanie składowych iloczynów skalarnych
5:          end for
6:      end for
7:  end parfor
8:  for l ← 1, log n do                ▷ Sumowanie składowych iloczynów skalarnych
9:      parfor P_i, 1 ⩽ i ⩽ n do
10:         for j ← 1, n do
11:             for k ← 1, n/2^l do
12:                 T_{i,j,k} ← T_{i,j,2k-1} + T_{i,j,2k}
13:             end for
14:         end for
15:     end parfor
16: end for
17: parfor P_i, 1 ⩽ i ⩽ n do
18:     for j ← 1, n do
19:         C_{i,j} ← T_{i,j,1}
20:     end for
21: end parfor
```

W algorytmie 5 między n procesorów zostało rozdzielone n^3 iloczynów $A_{ik}B_{kj}$ tak, że w wierszach (2-8) procesor P_i oblicza elementy składowe iloczynów skalarnych $i-tego$ wiersza macierzy A oraz wszystkich kolumn macierzy B.

Najpierw każdy procesor $P_i, (1 \leqslant i \leqslant n)$ pobiera n elementów i-tego wiersza macierzy A oraz n^2 elementów macierzy B i zapisuje do pamięci wspólnej n^2 iloczynów $t[i, j, k]$. Instrukcja w wierszu 12 wykonuje się na każdym procesorze $n^2/2^l$ razy, gdzie l przebiega od 1 do $\log n$. Łącznie daje to $2n^2(1 - \frac{1}{2r}$ wykonań wiersza 12 na procesor, przy czym każde wykonanie wymaga dwukrotnego odczytania danych z pamięci wspólnej, wykonania sumowania skalarów i zapisu wyniku. W wierszach (18-22) każdy z procesorów wykonuje n odczytów i zapisów do pamięci współdzielonej. Wynika stąd, że złożoność czasowa algorytmu 5 to $\mathcal{O}(n^2)$ i odpowiednio koszt algorytmu wynosi $\mathcal{O}(n^3)$.

W myśl definicji 1.29 i na podstawie powyższych rozważań złożoność komunikacyjna będzie równa $n + n^2 + n^2 + 3(2n^2(1 - \frac{1}{2r})) + 2n = \mathcal{O}(n^2)$.

2.3 Algorytmy w modelu sieciowym

2.3.1 Algorytm Cannona

Powiedzmy, że chcemy przeprowadzić obliczenie $D = C + AB$ gdzie $A, B, C \in \mathbb{R}^{n\times n}$ w dwuwymiarowym torusie o rozmiarach $p_1 \times p_1$ oraz, że $n = rp_1$. Macierze $A = (A_{ij})$, $B = (B_{ij})$, $C = (C_{ij})$ możemy rozpatrywać jako macierze blokowe, gdzie A_{ij}, B_{ij}, C_{ij} są macierzami $r \times r$. Przyjmijmy, że węzeł P_{ij} zawiera blok A_{ij}, B_{ij} i C_{ij} oraz, że jego zadanie polega na nadpisywaniu macierzy C_{ij} poprzez

$$D_{ij} = C_{ij} + \sum_{k=1}^{p_1} A_{ik}B_{kj}. \tag{2.4}$$

Zanim przejdziemy do przypadku ogólnego, pokażemy algorytm dla przypadku $p_1 = 3$. Rozważmy sieć w topologii dwuwymiarowego torusa 3×3 (rys. 2.1).

P_{11}	P_{12}	P_{13}
P_{21}	P_{22}	P_{23}
P_{31}	P_{32}	P_{33}

Rysunek 2.1: Węzły w dwuwymiarowym torusie 3×3

Weźmy pod uwagę wyłącznie aktywność na węźle P_{11}. Wykonuje on obliczenie

$$D_{11} = C_{11} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}. \tag{2.5}$$

Załóżmy, że rozmieściliśmy sześć podmacierzy A i B tak jak na rysunku 2.2.

A_{11}	B_{11}	A_{12}	$-$	A_{13}	$-$
$-$	B_{21}	$-$	$-$	$-$	$-$
$-$	B_{31}	$-$	$-$	$-$	$-$

Rysunek 2.2: Wstępne rozmieszczenie macierzy blokowych A_{ij} i B_{ij} koniecznych do wykonania algorytmu tylko na węźle P_{11}. Miejsca oznaczone znakiem „$-$” w ogólności są przeznaczone dla pozostałych danych. Rozmieszczenie danych w sieci odpowiada rysunkowi 2.1

Algorytm polega na przesuwaniu wierszy powstałych z bloków macierzy zapisanych w węzłach sieci. W każdym kroku na wybranym dla przykładu węźle P_{11}

A_{12}	B_{21}	A_{13}	–	A_{11}	–
–	B_{31}	–	–	–	–
–	B_{11}	–	–	–	–

$C_{loc} = C_{loc} + A_{12}B_{21}$

A_{13}	B_{31}	A_{11}	–	A_{12}	–
–	B_{11}	–	–	–	–
–	B_{21}	–	–	–	–

$C_{loc} = C_{loc} + A_{13}B_{31}$

A_{11}	B_{11}	A_{12}	–	A_{13}	–
–	B_{21}	–	–	–	–
–	B_{31}	–	–	–	–

$C_{loc} = C_{loc} + A_{11}B_{11}$

Rysunek 2.3: Rozmieszczenie danych dla trzech kroków metody Cannona z uwzględnienem danych koniecznych do obliczeń tylko na węźle P_{11}

wykonujemy lokalne obliczenia prowadzące do otrzymania wartości wyrażenia (2.5). Kolejne kroki algorytmu przedstawione są na rysunku 2.3.
Po wykonaniu trzech kroków węzeł P_{11} ma w pamięci lokalnej macierz D_{11}.

Przepływ danych został zorganizowany w taki sposób, że bloki A_{ij} przesuwanę są w siatce z prawej na lewą, zaś bloki B_{ij} — z dołu na górę. Widać, że węzeł P_{11} musi wykonywać algorytm 6.

Algorytm 6 Algorytm Cannona dla dwuwymiarowego torusa 3×3.

1: **for** $i \leftarrow 1, 3$ **do**
2: SEND(A_{loc}, lewo)
3: SEND(B_{loc}, góra)
4: RECV(A_{loc}, prawo)
5: RECV(B_{loc}, dół)
6: $C_{loc} = C_{loc} + A_{loc}B_{loc}$
7: **end for**

W przyjętym modelu obliczeń zadziała również algorytm 7:

Algorytm 7 Algorytm Cannona dla dwuwymiarowego torusa 3×3.

1: **for** $i \leftarrow 1, 3$ **do**
2: SEND(A_{loc}, lewo)
3: RECV(A_{loc}, prawo)
4: SEND(B_{loc}, góra)
5: RECV(B_{loc}, dół)
6: $C_{loc} = C_{loc} + A_{loc}B_{loc}$
7: **end for**

Wykonanie algorytm 7 trwa nieco dłużej ze względu zatrzymanie wykonania programu dopóki macierz A_{loc} nie zostanie wysłana[1].

Rozważymy teraz aktywność węzłów P_{12}, P_{13}, P_{21}, P_{31}. W dotychczasowych roważaniach odpowiednio pomagały one jedynie w przesuwaniu bloków A_{11}, A_{12}, A_{13} oraz B_{11}, B_{21} i A_{31}. Gdyby B_{32}, B_{12}, B_{22} przechodziły przez węzeł P_{12} w trakcie wykonywania algorytmu, wówczas moglibyśmy na węźle P_{12} otrzymać wartość wyrażenia:

$$D_{12} = C_{12} + A_{13}B_{32} + A_{11}B_{12} + A_{12}B_{22}.$$

Rozumując podobnie widzimy, że węzeł P_{13} mógłby obliczać wyrażenie:

$$D_{13} = C_{13} + A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33}.$$

o ile B_{13}, B_{23} i B_{33} znajdowałyby się na węźle odpowiednio dla kroków $t = 1, 2, 3$. Uwzględniając obliczenia na węzłach P_{12} i P_{13} możemy zainicjalizować sieć jak na rys. 2.4

A_{11}	B_{11}	A_{12}	B_{22}	A_{13}	B_{33}
–	B_{21}	–	B_{32}	–	B_{13}
–	B_{31}	–	B_{12}	–	B_{23}

Rysunek 2.4: Wstępne rozmieszczenie macierzy blokowych A_{ij} i B_{ij} koniecznych do wykonania obliczeń na węzłach P_{11}, P_{12} i P_{13}.

Zastosowanie odpowiednich przesunięć w algorytmie Cannona ilustruje rys. 2.5.

[1]W interfejsie MPI obydwa algorytmy traktowane literalnie wywołują zazębienie (ang. *deadlock*) ze względu na blokującą funkcję MPI_Send; istnieje szereg metod nieblokujących pozwalających na implementację obydwu algorytmów.

A_{12}	B_{21}	A_{13}	B_{32}	A_{11}	B_{13}
$-$	B_{31}	$-$	B_{12}	$-$	B_{23}
$-$	B_{11}	$-$	B_{22}	$-$	B_{33}

$t = 1$

A_{13}	B_{31}	A_{11}	B_{12}	A_{12}	B_{23}
$-$	B_{11}	$-$	B_{22}	$-$	B_{33}
$-$	B_{21}	$-$	B_{32}	$-$	B_{13}

$t = 2$

A_{11}	B_{11}	A_{12}	B_{22}	A_{13}	B_{33}
$-$	B_{21}	$-$	B_{32}	$-$	B_{13}
$-$	B_{31}	$-$	B_{12}	$-$	B_{23}

$t = 3$

Rysunek 2.5: Rozmieszczenie danych dla trzech kroków metody Cannona z uwzględnienem danych koniecznych do obliczeń na węzłach P_{11}, P_{12} i P_{13}.

Widać, że jeśli macierz B jest wstępnie rozmieszczona z zastosowaniem jednego przesunięcia, to po zakończeniu obliczeń w węzłach P_{11}, P_{12}, P_{13} otrzymujemy pierwszy wiersz macierzy C.

Rozważmy teraz aktywność na wszystkich dziewięciu węzłach sieci. Powiedzmy, że rozmieszczamy dane w sieci tak jak przedstawiono na rys. 2.6.

A_{11}	B_{11}	A_{12}	B_{22}	A_{13}	B_{33}
A_{22}	B_{21}	A_{23}	B_{32}	A_{21}	B_{13}
A_{33}	B_{31}	A_{31}	B_{12}	A_{32}	B_{23}

Rysunek 2.6: Wstępne rozmieszczenie macierzy blokowych A_{ij} i B_{ij} w dwuwymiarowym torusie 3×3.

Jeśli wstępnie przesuniemy drugi wiersz o jedną kolumnę w lewo i trzeci wiersz – o dwie, wówczas możemy przeprowadzić odpowiednie dodawania i mnożenia występujące w wyrażeniu (2.4) dla każdego węzła sieci w każdym kroku algorytmu.

A_{12}	B_{21}	A_{13}	B_{32}	A_{11}	B_{13}
A_{23}	B_{31}	A_{23}	B_{12}	A_{22}	B_{23}
A_{31}	B_{11}	A_{32}	B_{22}	A_{33}	B_{33}

$t = 1$

A_{13}	B_{31}	A_{11}	B_{12}	A_{12}	B_{23}
A_{21}	B_{11}	A_{22}	B_{22}	A_{23}	B_{33}
A_{32}	B_{21}	A_{33}	B_{32}	A_{31}	B_{13}

$t = 2$

A_{11}	B_{11}	A_{12}	B_{22}	A_{13}	B_{33}
A_{22}	B_{21}	A_{23}	B_{32}	A_{21}	B_{13}
A_{33}	B_{31}	A_{31}	B_{12}	A_{32}	B_{23}

$t = 3$

Rysunek 2.7: Rozmieszczenie danych dla trzech kroków metody Cannona w dwuwymiarowym torusie 3×3.

Teraz jesteśmy gotowi do przedstawienia ogólnej wersji algorytmu Cannona. Założymy, że węzły P_{ij} mają w pamięci lokalnej macierze A_{ij}, B_{ij} i C_{ij}. Żeby wstępnie przesunąć bloki macierzy A zauważmy, że i-ty wiersz węzłów sieci powinien przesłać macierze A_{ij} w lewo o $i - 1$ pozycji[2]. Podobnie, j-tą kolumnę węzłów przesuwamy w górę o $j - 1$ pozycji. Powyższe rozważania prowadzą do sformułowania algorytmu 8.

Algorytm 8 Algorytm Cannona[7] (część pierwsza)

Dane pomocnicze: p_1, (μ, λ) (współrzędne węzła w sieci), *góra*, *dół*, *lewo*, *prawo* (identyfikatory sąsiadujących węzłów), $wiersz = 1 + (mu - 1)r : \mu r$, $kolumna = 1 + (\lambda - 1)r : \lambda r$, $B_{loc} = B(wiersz, kolumna)$, $A_{loc} = A(wiersz, kolumna)$, $C_{loc} = B(wiersz, kolumna)$

1: **for** $k \leftarrow 1, \mu - 1$ **do** ▷ Wstępne przesunięcie $A_{\mu j}$ i $B_{i\lambda}$.
2: SEND(A_{loc}, lewo)
3: RECV(A_{loc}, prawo)
4: **end for**
5: **for** $k \leftarrow 1, \lambda - 1$ **do**
6: SEND(B_{loc}, góra)
7: RECV(B_{loc}, dół)
8: **end for**

[2]Numerację elementów zaczynamy od 1.

Algorytm 8 Algorytm Cannona[7] (część druga)

```
 9: for k ← 1, p_1 do                    ▷ Obliczanie iloczynu macierzy A_loc B_loc
10:     C_loc = C_loc + A_loc B_loc
11:     SEND(A_loc, lewo)
12:     SEND(B_loc, góra)
13:     RECV(A_loc, prawo)
14:     RECV(B_loc, dół)
15: end for
16: for k ← 1, μ − 1 do                 ▷ Doprowadzanie rozkładu podmacierzy
17:     SEND(A_loc, prawo)                          ▷ do stanu początkowego
18:     RECV(A_loc, lewo)
19: end for
20: for k ← 1, λ − 1 do
21:     SEND(B_loc, dół)
22:     RECV(B_loc, góra)
23: end for
```

Złożoność, przyspieszenie, koszt i efektywność algorytmu 8 są następujące[4]:

$$\begin{aligned} T_p(n)|_{p=n^2} = T(n) &= \mathcal{O}(n), \\ S(n) &= \mathcal{O}(\frac{n^3}{n}) = \mathcal{O}(n^2), \\ C(n) &= \mathcal{O}(n^2 n) = \Theta(n^3), \\ E(n) &= \mathcal{O}(\frac{n^3}{n^3}) = \Theta(1). \end{aligned}$$

Zgodnie z def. 1.24 algorytm jest optymalny względem kosztu.

Operacje (16-23) algorytmu 8 nie są istotne dla samego mnożenia macierzy. Ich zadaniem jest doprowadzenie rozkładu podmacierzy A_{ij}, B_{ij} w torusie do stanu początkowego. Operacja taka daje możliwość wykorzystanie początkowego rozkładu danych do dalszych operacji na macierzach A i B.

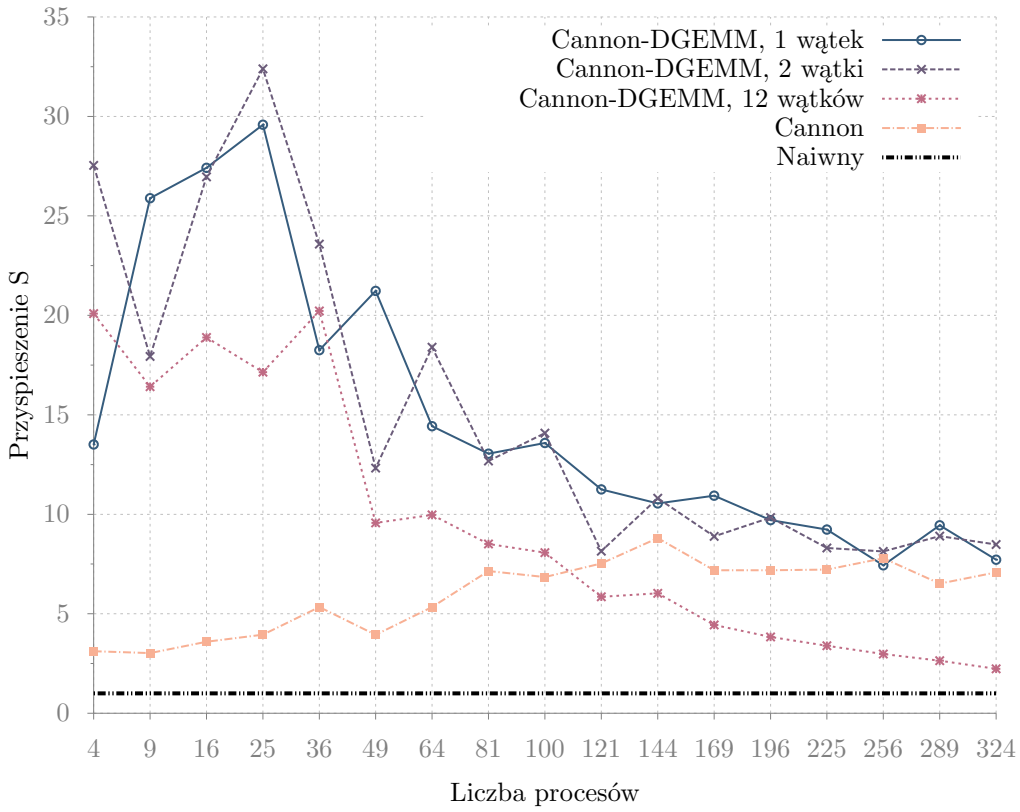
Rozdział 3

Implementacja

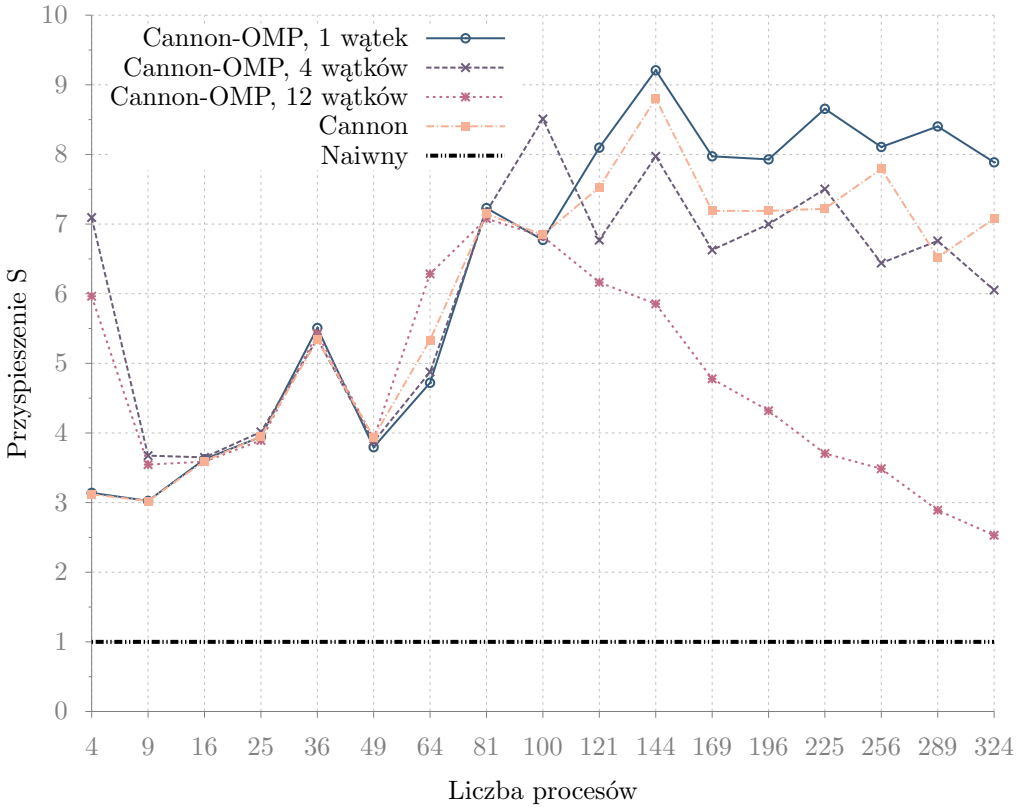
3.1 Organizacja kodu źródłowego

3.2 Opis programu

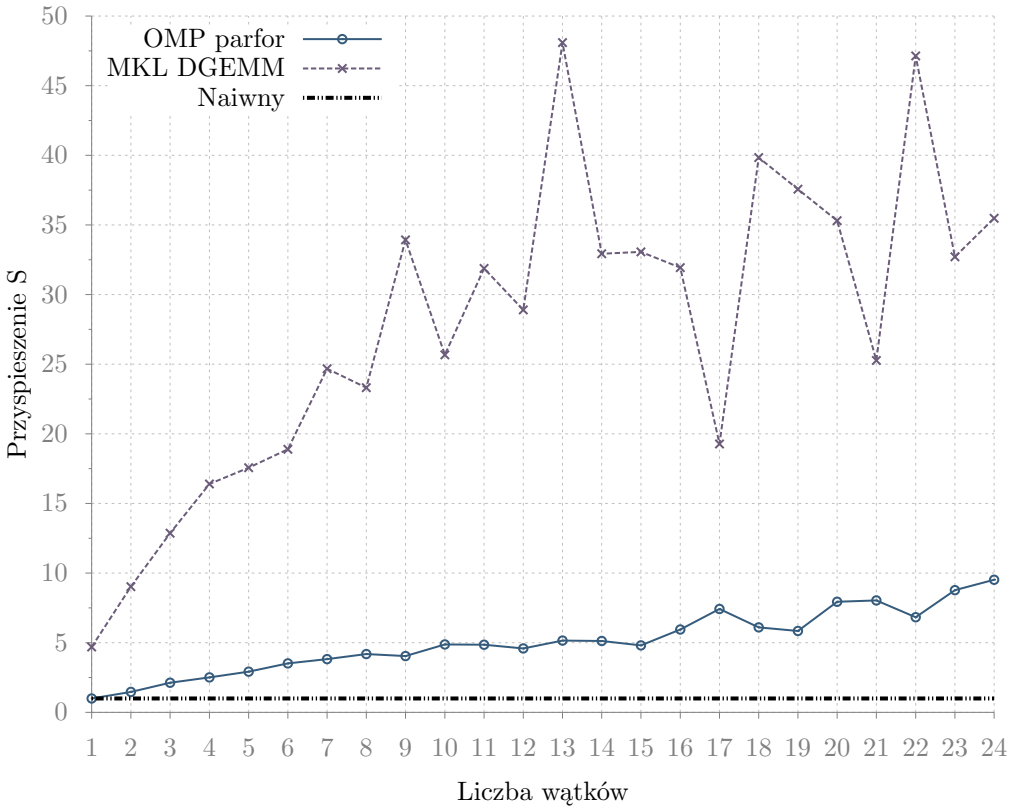
3.3 Analiza wydajności



Rysunek 3.1: test



Rysunek 3.2: test



Rysunek 3.3: test

Bibliografia

[1] Ed J. Radatz. *Standards Coordinating Committee 10, Terms and Definitions. The IEEE Standard Dictionary of Electrical and Electronics Terms.* IEEE, 1996.

[2] Fayez Gebali. *Algorithms and Parallel Computing.* Wiley Publishing, 1st edition, 2011.

[3] Krzysztof Ostrowski and Piotr Byszewski. Komputery macierzowe. 2006.

[4] Zbigniew Czech. *Wprowadzenie do obliczeń równoległych.* Wydawnictwo Naukowe PWN, Warszawa, 2013.

[5] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Education Group, 2003.

[6] Cormen Thomas H., Leiserson Charles E., Rivest Roland L., and Stein Clifford. *Wprowadzenie do algorytmów.* WNT, 2001.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.).* Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[8] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[9] Shmuel Winograd. On multiplication of 2×2 matrices. *Linear Algebra and its Applications*, 4:381–388, 1971.

[10] Jacques Loeckx, editor. *Automata, Languages and Programming, 2nd Colloquium, University of Saarbrücken, July 29 - August 2, 1974, Proceedings*, volume 14 of *Lecture Notes in Computer Science*. Springer, 1974.

[11] Craig C. Douglas, Michael Heroux, Gordon Slishman, Roger M. Smith, and Roger M. Gemmw: A portable level 3 blas winograd variant of strassen's matrix-matrix multiply algorithm, 1994.

[12] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. *CoRR*, abs/1202.3173, 2012.