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

Rozdział 1

Wstęp

Rozdział 2

Wiadomości wstępne

2.1 Ustalenia terminologiczne

Definicja 2.1 (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 2.2 (Domknięcie przechodnie zbioru)**.** Domknięciem przechodnim zbioru X nazywamy najmniejszy w sensie inkluzji zbiór przechodni, który zawiera X.

Definicja 2.3 (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 2.4 (Acykliczny graf skierowany (DAG))**.** Acyklicznym grafem skierowanym nazywamy graf skierowany nie zawierający cykli.

Definicja 2.5 (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 2.6 (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 2.7 (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 2.8 (Droga)**. Drogą** w grafie G nazywamy ścieżkę, której wierzchołki są różne.

Definicja 2.9 (Długość drogi)**. Długością** drogi w grafie G nazywamy liczbę krawędzi, które zawiera droga.

Definicja 2.10 (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 2.11 (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 2.12 (Macierz)**.** Niech $\mathbb{K}$ będzie ciałem. Macierzą o m wierszach i n kolumnach i wartościach w $\mathbb{K}$ (krótko: macierzą $m \times n$) nazwamy każde odwzorowanie $A : \{1, \ldots, m\} \times \{1, \ldots, n\} \to \mathbb{K}, (i,j) \longmapsto A_{ij}$

2.2 Klasyfikacja algorytmów

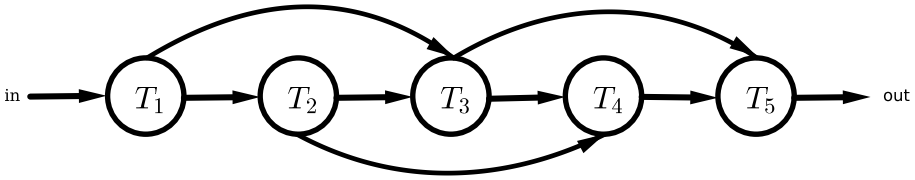
Definicja 2.13 (Algorytm)**.** Zbiór jednoznacznie określonych reguł lub zadań obliczeniowych prowadzących w skończonej ilości kroków do rozwiązania pewnego problemu [1].

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 wykonaniu algorytmu.

Definicja 2.14 (Algorytm sekwencyjny)**. Algorytm sekwencyjny** (rys. 2.1) 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.

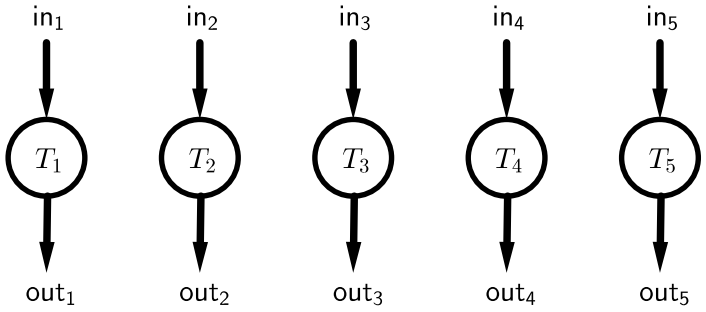


Rysunek 2.1: Algorytm sekwencyjny

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 2.15 (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 2.16 (Algorytm równoległy)**. Algorytmem równoległym** (rys. 2.2) 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 2.2: Algorytm równoległy

Definicja 2.17 (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 (*Data-level parallelism*), gdzie pracujemy na wielu bitach danych lub na wielu danych jednocześnie.

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

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

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

[2]

Przykład 2.1. 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.

2.2.1 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.

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 dodatkoych 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 zaplanowania po wykonaniu operacji przedstawionej przez węzeł i.

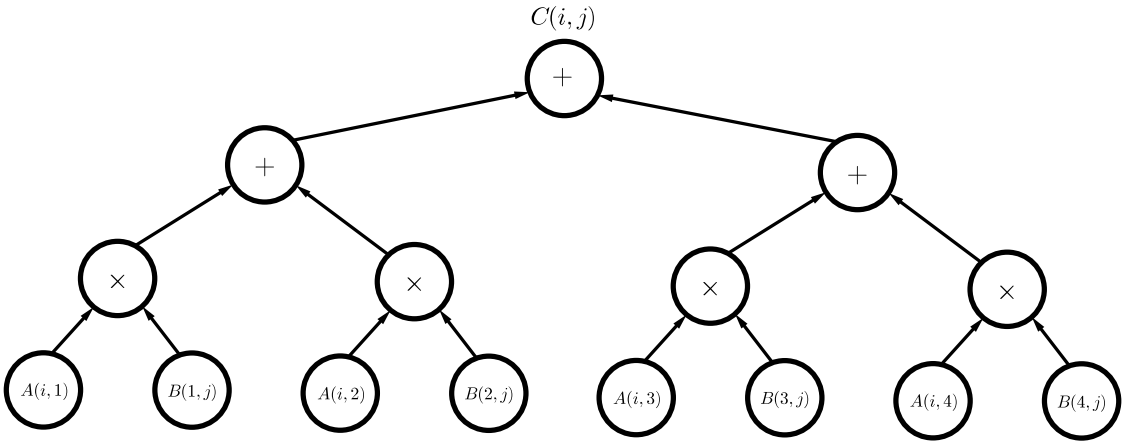
Przyjmuje się, że czas t_i węzła wejściowego i wynosi 0 oraz żaden procesor nie jest przyporządkowany do tego węzła.

Definicja 2.18 (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 $\max_{i \in N} t_i$. Złożonosc 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 2.2. 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. 2.3. 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 2.3: Standardowy iloczyn macierzowy

2.3 Ocena algorytmów

2.3.1 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.

2.3.2 Algorytmy równoległe

Definicja 2.19 (Pesymistyczna złożoność obliczeniowa[3])**.** Załóżmy że algorytm równoległy R rozwiązuje problem P o rozmiarze n. **Pesymityczną złożonością czasową algorytmu** R nazywamy funkcję:

$$T_p(n) = \sup_{d \in D_n} \{t(p,d)\}, \tag{2.1}$$

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.

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 2.20 (Przyspieszenie bezwzględne[3])**.** Niech P będzie pewnym zadaniem obliczeniowym, n – rozmiarem danych wejjściowych. Wówczas

$$S_p(n) = \frac{T^*(n)}{T_p(n)(N)} \tag{2.2}$$

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 2.2 nazywamy **przyspieszeniem bezwzględnym** algorytmu R.

Wniosek 2.1. Zgodnie z definicją 2.19 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 2.1. 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 2.2. 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 2.21 (Przyspieszenie bezwzględne[3])**.** Niech P będzie pewnym zadaniem obliczeniowym, n – rozmiarem danych wejściowych. Wówczas

$$S_p(n) = \frac{T_1(n)}{T_p(n)(N)} \tag{2.3}$$

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 2.2 nazywamy **przyspieszeniem względnym** algorytmu R.

Koszt

Definicja 2.22 (Koszt algorytmu[3])**.** Niech $T_p(n)$ będzie pędzie pesymistyczną złożonością obliczeniową algorytmu R dla p procesorów. Wówczas funkcję

$$C_p(n) = pT_p(n) \tag{2.4}$$

nazywamy kosztem algorytmu R dla p procesorów.

W myśl definicji 2.19 koszt algorytmu możemy rozumieć przez analogię do liczby operacji dominujących wykonanych łącznie przez wszystkie procesory.

Wniosek 2.2. 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 2.3. 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 2.23 (Koszt organizacji obliczeń)**.** Różnicę

$$C_p^O(n) = C_p(n) - T^*(n) = pT_p(n) - T^*(n) \tag{2.5}$$

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

Efektywność

Definicja 2.24 (Efaktywność[3])**.** Niech $T_p(n)$ będzie pesymistyczną złożonością czasową algorytmu R dla p procesorów i problemu R o rozmiarze n. Wówczas mamy

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} = \frac{T_1(n)}{C_p(n)} = \frac{S_p(n)}{p} \tag{2.6}$$

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

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:

$$T_1(n) = T_1^s(n) + T_1^r(n) \tag{2.7}$$

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

$$S(p,n) = \frac{T_1(n)}{T_p(n)} \leqslant \frac{T_1^s(n) + T_1^r(n)}{T_1^s(n) + T_1^r(n)/p + T_p^o(n)} \tag{2.8}$$

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 (2.8) przez pominięcie złożoności $T_p^o(n)$. Mamy wówczas:

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

gdzie s – część obliczeń w algorytmie które muszą być wykonane sekwencyjnie; p – liczba procesorów.
Otrzymany wzór (2.3.2) nazywamy **prawem Amdahla**.

Definicja 2.25 (Prawo Amdahla[4])**.** Niech s będzie częścią operacji w algorytmie R, która musi być wykonana sekwencyjnie, taką że $0 \leqslant s \leqslant 1$. Wówczas maksymalne przyspieszenie ϕ osiągalne przez komputer równoległy z p procesorami wykonujący algorytm R spełnia nierówność:

$$\phi \leqslant \frac{1}{s + (1-s)/p} \tag{2.9}$$

Nierówność z definicji 2.25 służy do wyznaczania górnego ograniczenia przyspieszenia będącego funkcją wielkości s oraz liczby procesorów p przy ustalonym rozmiarze problemu n.

Wniosek 2.3. Przechodząc z wyrażeniem (**??**) 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).

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:

$$\Psi_p(n) \leqslant \frac{\sigma + p\rho}{\sigma + \rho} = \sigma + p\rho = \sigma + p\,(1 - \sigma) = p + (1 - p)\,\sigma \tag{2.10}$$

Wzór (2.10) jest znany jako **prawo Gustafsona i Barsisa**.

Definicja 2.26 (Prawo Gustafsona i Barsisa)**.** 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:

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

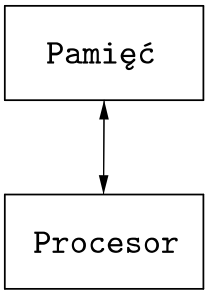
2.4 Teoretyczne modele obliczeń

2.4.1 Model RAM

Nim przejdziemy do omówienia modeli obliczeń równoległych zajmiemy się omówieniem modelu RAM zwanego również architekturą von Neumanna.

Model RAM (*Random Access Machine*) odpowiada rozważaniom zawartym w 2.3. 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. 2.4) 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 2.4: Model obliczeń sekwencyjnych RAM

Tabela 2.1: Przykładowa lista instrukcji procesora[3]

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

2.4.2 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.
Każdemu procesorowi przypożądkowany jest niepowtarzająca się liczba naturalna. Jest to lokalnie dostępny indeks, numer procesora lub jego identyfikator.

W modelu wspólnej pamięci wyróżniamy dwa podstawowe tryby operacji. W pierwszym trybie, synchronicznym, wszystkie procesory działają synchronicznie według wspólnego zegara. Model ten nazywamy równoległą maszyną o dostepie swobodnym (PRAM, parallel random-access machine).
W drugi trybie, asynchronicznym, 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.

Definicja 2.27. Klasyfikacja PRAM ze względu na dostęp do pamięci

- **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.

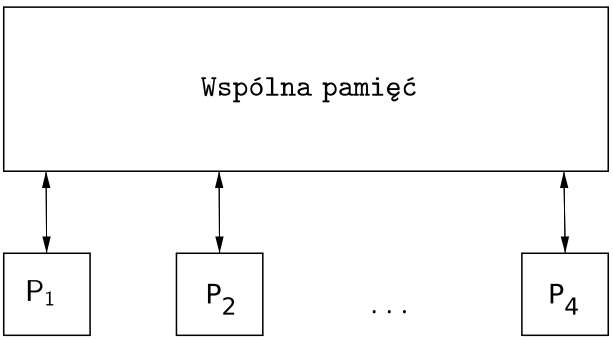
Jeśli nie poczyni się żadnych dodatkowych założeń, to nie jest jasno określone, co zostanie zapisane w komórce pamięci w wyniku jednoczesnego zapisywania do niej przez wiele procesorów w algorytmie typu CRCW. W literaturze można spotkać wiele typów maszyny PRAM, które różnią się sposobami rozwiązywania konfliktów zapisu. Można wśród nich wyróżnić[5]:

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



Rysunek 2.5: Model wspólnej pamięci

2.4.3 Model sieciowy

Sieć możemy przedstawić modelowo jako graf $G = (N, E)$, gdzie każdy węzeł $i \in N$ oznacza procesor, a każda krawędź $(i, j) \in E$ – dwukierunkową komunikację między procesorami i i j. Przyjmujemy, że każdy procesor ma swoją lokalną pamięć i nie ma żadnej pamięci współdzielonej przez procesory. Tak jak w przypadku modelu z pamięcią wspólną, operacje w sieci mogą być synchroniczne lub asynchroniczne.

W opisie algorytmów dla modelu sieciowego potrzebujemy zdefiniować dwie instrukcje do opisania komunikacji między procesorami.

1. send(X, i)

2. receive(X, j)

Procesor P wykonujący instrukcję **send** wysyła kopię X do procesora P_i, następnie natychmiast przechodzi do wykonywania kolejnej instrukcji.
Procesor P wykonujący instrukcję **receive** zatrzymuje wykonanie programu aż do chwili, gdy otrzyma dane z procesora P_j, a następnie przechowuje dane w Y i kontynuuje wykonanie programu.

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.

Charakterysuje ją kilka parametrów:

1. średnica – maksymalna odległość (krawędziowa) między dowolną parą węzłów; im miejsza, tym lepiej.
2. maksymalny stopień wierzchołka – maksymalna liczba łączy do dane procesora
3. szerokość połowienia sieci – minimalna liczba krawędzi, które muszą zostać usunięty, aby podzielić ją na dwie równe podsieci
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 procesorami; w najprostrzym przypadku mierzony liczbą krawędzi

Sieć liniowa

Definicja 2.28 (Sieć liniowa)**.** Model składa się z p procesorów $P_1, P_2, \ldots, P_p$ połączonych ze sobą w ciąg, tzn. procesor P_i połączony jest z procesorem P_{i-1} i P_{i+1}, o ile takie istnieją. Średnica takiej sieci wynosi $p-1$, jej maksymalny stopień wynosi 2.

Definicja 2.29 (Torus)**.** Sieć liniowa z połączonymi końcami.

Sieć dwuwymiarowa

Dwuwymiarowa sieć 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

Sieć hipersześcienna

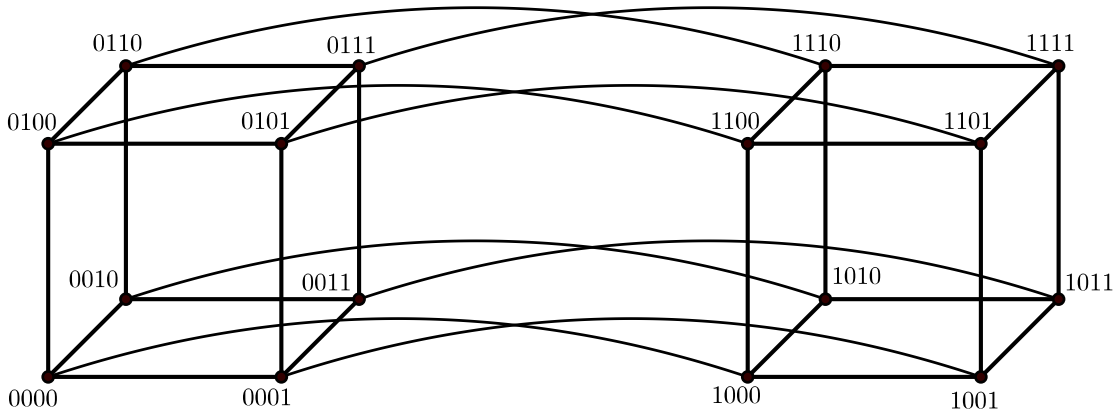
Definicja 2.30. Kostka Boola
Niech $i_{d-1}i_{d-2}\ldots i_0$, gdzie $0 \leqslant i \leqslant p-1$ będzie binarną reprezentacją i. Wówczas procesor i jest połaczony z procesorem $P_{i^{(j)}}$, gdzie $i^{(j)} = i_{d-1}\ldots\overline{i_j}\ldots i_0$ i

$\overline{i_j} = 1 - i_j$. Innymi słowy, dwa procesory są ze sobą połączone wtedy i tylko wtedy, gdy ich wskaźniki różnią się tylko jednym bitem.

Sieć w topologii hipersześcianu składa się z $p = 2^d$ procesorów połączonych w d-wymiarową kostkę Boola.

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

Średnica d-wymiarowego hipersześcianu wynosi $d = \log p$. Jest tak ponieważ odległośc w grafie między dwoma procesorami 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 2.6: Sieć w topologii hipersześcianu

Rozdział 3

Klasyczne algorytmy mnożenia macierzy

W swojej pracy *„Gaussian Elimination is not Optimal”* z 1969 roku Volker Strassen pokazał rekursyjny algorytm mnożenia macierzy kwadratowych wymiaru $m2^k$ o złożoności $\mathcal{O}(n^{2,81})$[6]. 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[7][8] do najczęściej implementowanej dzisiaj postaci algorytmu Strassena-Winograda (implementacja zawiera się na przykład w bibliotece GEMMW[9]). W przypadku macierzy 2×2 wykonuje on 7 operacji mnożenia i 15 dodawań[10].

3.1 Algorytm naiwny

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

3.2 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

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{3.1}$$

(3.1) możemy wyrazić inaczej:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{11} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{3.2}$$

Z powyższych konstatacji nasuwa się łatwy algorytm rekurencyjny.

3.3 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

$$C = \begin{bmatrix} H_1 + H_4 - H_5 + H_7 & H_3 + H_5 \\ H_2 + H_4 & H_1 + H_3 - H_2 + H_6 \end{bmatrix} \tag{3.3}$$

3.4 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[10]. Złożoność obliczeniową $O(n^{w_0})$ algortytmu oznacza, że jego wykonanie zatrzymuje się po osiągnięciu macierzy wymiaru 1×1.

Rozdział 4

Równoległe algorytmy mnożenia macierzy

4.1 Algorytm w modelu PRAM

Algorytm sumowania

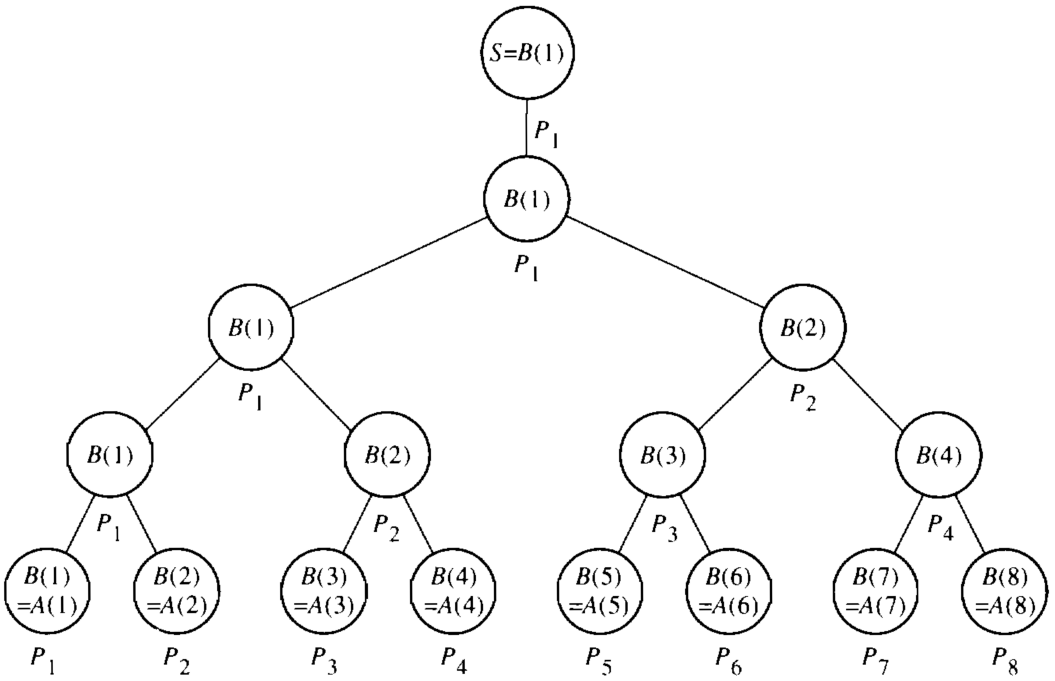
Niech dany będzie tablica A $n = 2^k$ liczb i maszyna PRAM z n-procesorami $\{P_1, P_2, \ldots, P_n\}$. Każdy z procesorów wykonuje synchronicznie poniższy algorytm

Przypadek dla $n = 8$ ilustruje rysunek 4.1. W pierwszym i drugim kroku kopia B tablicy A jest tworzona w pamięci wspólnej. Zadania obliczeniowe w kroku 3 są na podstawie wyważonego drzewa binarnego, którego liście odpowiadają elementom tablicy A. Procesor odpowiedzialny za wykonanie za wykonanie operacji jest określony przez indeks poniżej węzła reprezentującego tę operację. Zauważmy, że procesor P_1, odpowiedzialny za ustawianie wartości $B(1)$ i zapisywanie sumy S, jest zawsze aktywny w trakcie wykonywania algorytmu, podczas gdy procesory P_5, P_6, P_7, P_8 są aktywne tylko podczas kroków 1 i 2.

Uwaga 4.1. Pomijamy szczegóły operacji dotyczących dostępu do pamięci. Operacje postaci `Ustaw A:=B+C`, gdzie A, B i C są zmiennymi wspólnymi będziemy interpretować jako ciąg instrukcji

Algorytm mnożenia

Rozważmy problem obliczenia iloczynu $\mathbf{C}$ dwóch macierzy $\mathbf{A}$, $\mathbf{B} \in \mathbb{R}^{n \times n}$, gdzie $n = 2^k$, dla pewnego $k \in \mathbf{N}$. Załóżmy, że dysponujemy n^3 procesorami $P_{i,j,l}$,



Rysunek 4.1: Algorytm sumowania ośmiu elementów w modelu PRAM z osmioma procesorami. Każdy wewnętrzny wierzchołek grafu reprezentuje operację sumowania.

$1 \leqslant i, j, l \leqslant n$ maszyny PRAM. Wówczas dla każdej pary (i, j), n procesorów $P_{i,j,l}$, gdzie $1 \leqslant l \leqslant n$, oblicza sumę $\sum_{l=1}^{n} A(i,l)B(l,j)$ w myśl algorytmu **??**.

Uwaga 4.2. Algorytm **??** wymaga równoległego odczytu ponieważ w trakcie wykonania kroku (1) 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 (1) wszystkie wymagają dostępu do $A(i,l)$.

4.2 Algorytmy w modelu sieciowym

4.2.1 Algorytm w dwuwymiarowym torusie

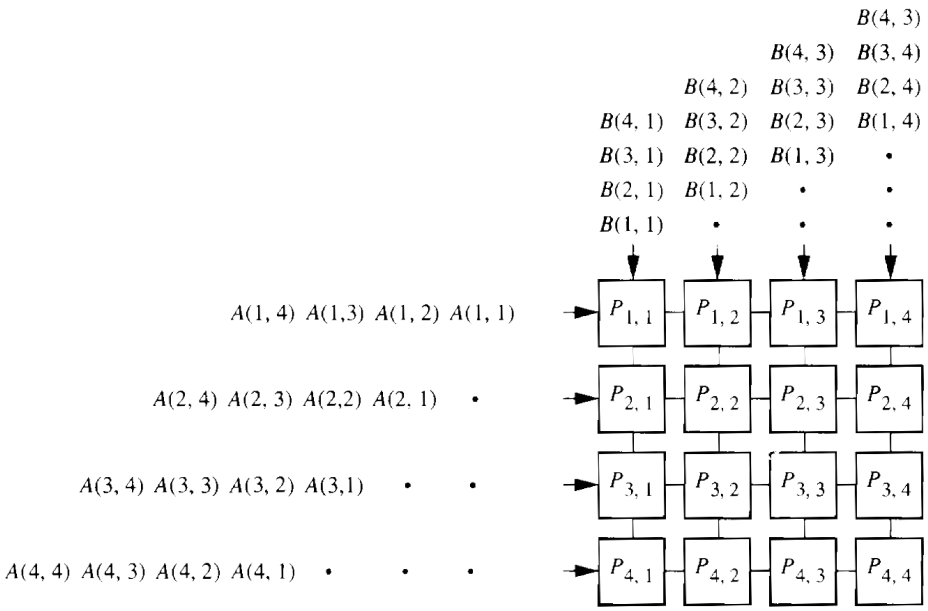
4.2.2 Algorytm systoliczny w sieci dwuwymiarowej

Rysunek 4.2 przedstawia możliwy schemat obliczenia iloczynu $AB = C$ w myśl paradygmatu systolicznego. Wiersze macierzy A są wprowadzane synchroniczne skośnie od lewej strony sieci; kolumny macierzy B są wprowadzane synchronicznie skośnie od góry sieci.

Gdy procesor $P_{i,j}$ odbierze dwie dane wejściowe $A(i,l)$ i $B(l,j)$, to przeprowadza operację $C(i,j) := C(i,j + A(i,l)(B(l,j)$; po tym przesyła $A(i,l)$ do swojego prawego sąsiada, a $B(l,j)$ do sąsiada poniżej.

Po $O(n)$ krokach, każdy procesor $P_{i,j}$ będzie miał szukaną wartość $C(i,j)$.

Algorytmy systoliczne pracują całkowicie synchronicznie; w każdej jednostce czasu, procesor otrzymuje dane od pewnego sąsiada, przeprowadza na nich lokalne obliczenia i następnie wysyła dane do któregoś swojego sąsiada.



Rysunek 4.2: Mnożenie macierzy w modelu sieciowym za pomocą algorytmu systolicznego. Wiersze macierzy **A** są synchronicznie umieszczane w sieci od lewej strony, podczas, gdy równocześnie „od góry" umieszczane są synchronicznie kolumny macierzy **B**. Gdy elementy $A(i,l)$ i $B(l,j)$ są dostępne na procesorze $P_{i,j}$, wykonywane jest działanie $C(i,j) = C(i,j) + A(i,l)B(l,j)$, $A(i,l)$ zostaje wysłany do procesora $P_{i,j+1}$ (o ile taki istnieje) oraz $B(l,j)$ zostaje wysłany do procesora $P_{i+1,j}$ (o ile taki istnieje).

4.2.3 Algorytm w topologii hipersześcianu

Rozgłaszanie „jednego do wszystkich" w hipersześcianie

Rozważmy problem rozgłaszania elementu X przechowywanego w rejestrze $D(0)$ procesora P_0 do wszystkich procesorów P_i p-procesorowego hipersześcianu, gdzie $p = 2^d$.

Algorytm polega na przechodzeniu z podkostek najniższego wymiaru, kolejno do najwyższego w d iteracjach. Pierwsza iteracja polega na wysłaniu kopii X przez procesor P_0 do procesora P_1. W drugiej iteracji procesor P_0 i P_1 wysyłają kopię X do P_2 i P_3 odpowiednio. Analogiczną operację przeprowadza się d-razy.

Algorytm **??** ma złożoność równoległą $\mathcal{O}(\log p)$

Algorytm mnożenia

Rozważmy problem mnożenia macierzy $AB = C$ w synchronicznym hipersześcianie z $p = n^3$ procesorów, gdzie wszystkie macierze są wymiaru $n \times n$.
Niech $n = 2^q$ i stąd $p = 2^{3q}$. Przypiszmy procesorom indeksy (l, i, j) takie, że $P_{l,i,j}$ oznacza procesor P_r, gdzie $r = ln^2 + in + j$. Innymi słowy, rozkładając indeks r binarnie otrzymuje, że q najbardziej znaczących bitów odpowiada indeksowi l, następne q najbardziej znaczących bitów odpowiada indeksowi i i ostatecznie q najmniej znaczących bitow odpowiada wskaźnikowi j. W szczególności, jeśli ustalimy dowolną parę wskaźników spośród l, i oraz j oraz będziemy przechodzili z pozostałym wskaźnikiem po wszystkich jego możliwych wartościach, otrzymamy podkostkę wymiaru q.

Wejściowy ciąg A jest zapamiętany w podkostce wyznaczonej przez procesory $P_{l,i,0}$, gdzie $0 \leqslant l, i \leqslant n-1$, tak, że $A(i,l)$ jest zapamiętane w procesorze $P_{l,i,0}$.
Podobnie ciąg B jest zapamiętany w podkostce procesorów $P_{l,0,j}$, gdzie procesor $P_{l,0,j}$ zapamiętuje $B(l,j)$.
Celem jest obliczeniem $C(i,j) = \sum_{l=0}^{n-1} A(i,l)B(l,j)$ dla $0 \leqslant i, j \leqslant n-1$.
Algorytm składa się z trzech etapów:

1. Dane wejściowe są rozdystrybuowane tak, że procesor $P_{l,i,j}$ pamięta $A(i,l)$ i $B(l,j)$ dla $0 \leqslant l, i, j \leqslant n-1$.

2. Procesor $P_{l,i,j}$ oblicza iloczyn $C'(l,i,j) = A(i,l)B(l,j)$ dla wszystkich $0 \leqslant i, j, l \leqslant n-1$.

3. Dla wszystkich $0 \leqslant i, j \leqslant n-1$ procesorów $P_{l,i,j}$, gdzie $0 \leqslant l \leqslant n-1$, obliczają sumę $C(i,j) = \sum_{i=0}^{n-1} C'(l,i,j)$

Implementacja pierwszego etapu składa się z dwóch części. W pierwszej z nich *rozgłaszamy*, dla każdego $i, l, A(i,l)$, z procesora $P_{i,l,0}$ do $P_{l,i,}$) dla $0 \leqslant j \leqslant n-1$. Ponieważ zbiór procesorów $\{P_{l,i,j} | 0 \leqslant j \leqslant n-1\}$ wyznacza q-wymiarową

kostkę dla każdej z par i oraz l, możemy użyć **algorytmu rozgłaszania**, żeby rozgłosić $A(i,l)$ od procesora $P_{l,i,0}$ do wszystkich procesorów $P_{l,i,j}$. W drugiej części każdy element $B(l,j)$ przechowywany w procesorze $P_{l,0,j}$ jest rozgłaszany do procesorów $P_{l,i,j}$ dla wszystkich $0 \leqslant i \leqslant n-1$. Pod koniec procesor $P_{l,i,j}$ będzie przechowywał dwie wartości: $A(i,l)$ i $B(l,j)$. Używając algorytmu (tutaj referencja) etap pierwszy ma równoległą złożonośc czasową $O(\log n)$.

Drugi etap polega na wykonywaniu pojedyńczych mnożeń na każdym z procesorów $P_{l,i,j}$. Stąd etap na etap ten składa się tylko jeden równoległy krok obliczeniowy. Pod koniec procesor $P_{l,i,j}$ przechowuje $C'(l,i,j)$.

Trzeci etap polega na obliczeniu n^2 sum $C(i,j)$. Wartości $C'(l,i,j)$ każdej z sum znajudują się w q-wymiarowym hipersześcianie $\{P_{l,i,j} \mid 0 \leqslant l \leqslant n-1\}$. Obliczanie takich sum ma równoległą złożoność czasową $O(\log n$. Procesor $P_{0,i,j}$ będzie przechowywał wartość $C(i,j)$ iloczynu.

Wobec powyższych iloczyn macierzy wymiaru $n \times n$ w sieci hipersześciennej może być obliczony w czasie $O(\log n)$ dysponując n^3 procesorami.

4.2.4 Algorytm Cannona w dwuwymiarowej sieci

[11]

4.2.5 Algorytm 2.5D

[12]

4.2.6 Algorytm 3D

[11]

4.2.7 Równoległy algorytm Strassena CAPS

[10]

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] H. Gupta and P. Sadayappan. Communication efficient matrix-multiplication on hypercubes. Technical Report 1994-25, Stanford Infolab, 1994.

[12] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. Technical Report UCB/EECS-2011-72, EECS Department, University of California, Berkeley, Jun 2011.