



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	4
1 Wiadomości wstępne	6
1.1 Ustalenia terminologiczne	6
1.2 Klasyfikacja algorytmów	8
1.3 Architektury równoległe	9
1.4 Reprezentacja algorytmów	11
1.5 Ocena algorytmów	13
1.5.1 Złożoność czasowa	13
1.5.2 Przyspieszenie	14
1.5.3 Koszt	15
1.5.4 Efektywność	16
1.5.5 Prawo Amdahla	16
1.5.6 Prawo Gustafsona i Barsisa	18
1.5.7 Miara Karpa-Flatta	19
1.6 Teoretyczne modele obliczeń	21
1.6.1 Model RAM	21
1.6.2 Model PRAM	22
1.6.3 Model sieciowy	24
1.7 Wybrane typy danych dla obliczeń rozproszonych	29
2 Mnożenie macierzy	31
2.1 Przegląd algorytmów klasycznych	31
2.2 Algorytmy w modelu z pamięcią wspólną	34
2.3 Algorytm Cannona	37
3 Rezultaty doświadczalne	45
A Implementacja	52
A.1 Opis programu	52

A.1.1	Wprowadzenie	52
A.1.2	Użytkowanie	53
A.1.3	Przebieg testów na klastrze Solaris	56
A.2	Omówienie kodu źródłowego	57
B	Załącznik	68

Wstęp

Golub[1] we wstępie do swojej książki poświęconej w całości tylko problemowi działań na macierzach powiada, że mnożenie macierzy, choć jest pomysłem zupełnie prostym, gdy tylko spojrzeć nań z obliczeniowego punktu widzenia, oferuje wielkie bogactwo podejść i przy tym pozostaje dla każdego pierwszym problemem, który prowadzi do zajmowania się macierzami w ogóle.

Problem mnożenia macierzy był podejmowany przez wielu autorów na przestrzeni lat (przykładowo prace [2], [3], [4] jeszcze z lat '60, ale i całkiem nowa i ważna praca [5]) i pozostaje polem poszukiwań lepszych algorytmów. Podejściem na miarę współczesnych obliczeń komputerowych są obliczenia rozproszone, a w szczególności – równoległe. Powody takiego stanu rzeczy można określić w trzech punktach:

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

Jeśli macierz posiada jakąś określoną wewnętrzną strukturę, na ogół możemy ją wykorzystać. Operacje na macierzach symetrycznych na przykład ze względu na powtarzanie się elementów, czy macierze rzadkie i trójkątne ze względu na wiele elementów zerowych wymagają mniej obliczeń. To przykłady niejednorodności problemu z którym mamy do czynienia. Jest to przyczyna, która w poszukiwaniu metod optymalnych prowadzi do rozwoju metod szczególnych.

W niniejszej pracy zajmiemy się metodami najogólniejszymi – metodami dla macierzy gęstych. Szczególna uwaga została położona na metodę Cannona[4] zaproponowaną w 1969 i rozwiniętą w pracy Ho, Johnsona i Edelmana[6] z

1991 roku. Przeanalizowana w rozdziale 2 została klasyczna koncepcja zaproponowana przez Cannona, a w części A.2 omówiono pewne poprawki zmniejszające koszty komunikacji algorytmu. Ponadto przedstawione zostały metody obliczeń w modelu z pamięcią wspólną. W rozdziale 3 przedstawiono przyspieszenia uzyskane przez implementację hybrydowych wariantów algorytmu Cannona, algorytmu w wersji gruboziarnistej oraz metod wykorzystujących architektury z pamięcią wspólną. Rozdział 1 stanowi podbudowę teoretyczną pod koncepcje rozwijane w rozdziałach późniejszych.

Rozdział 1

Wiadomości wstępne

1.1 Ustalenia terminologiczne

Definicja 1.1 (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 1.2 (Acykliczny graf skierowany (DAG)). Acyklicznym grafem skierowanym nazywamy graf skierowany nie zawierający cykli.

Definicja 1.3 (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 1.4 (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 1.5 (Domknięcie przechodnie zbioru). Domknięciem przechodnim zbioru X nazywamy najmniejszy w sensie inkluzji zbiór przechodni, który zawiera X .

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, \dots, v_n) taki, że dla każdego $k \in \{0, 1, \dots, n-1\}$ 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 $x_1x_2 \dots x_nx_1$ nazywamy **cyklem**, jeśli wszystkie wierzchołki $x_1, x_2 \dots x_n$ 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*) $d_G^-(v)$:

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

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

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

1.2 Klasyfikacja algorytmów

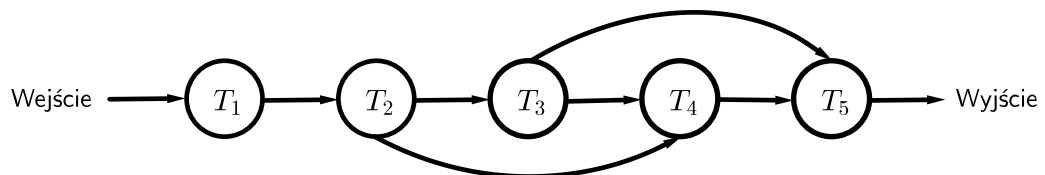
Definicja 1.13 (Algorytm). Zbiór jednoznacznie określonych reguł lub zadań obliczeniowych prowadzących w skończonej ilości kroków do rozwiązania pewnego problemu [7].

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 elementami określającymi dowolny algorytm są:

1. zadania do wykonania,
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 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.

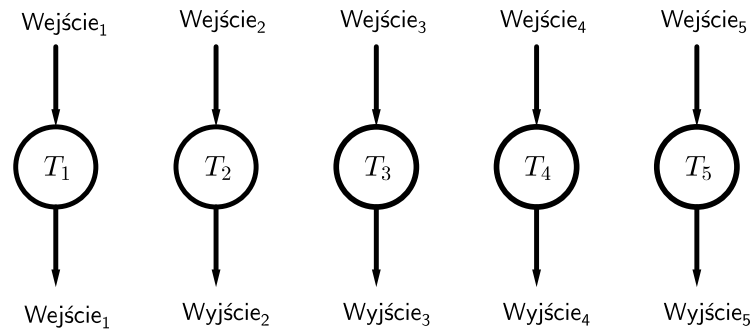


Rysunek 1.1: Algorytm sekwencyjny

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 [7].

Definicja 1.16 (Algorytm równoległy). **Algorytmem równoległym** (rys. 1.2) nazywamy każdy algorytm, w którym spośród określonych w nim zadań T_1, T_2, \dots, T_n co najmniej dwa zadania $T_i, T_j, i \neq j$ dzięki ich wzajemnej niezależności, mogą być wykonane równocześnie [8].



Rysunek 1.2: Algorytm równoległy

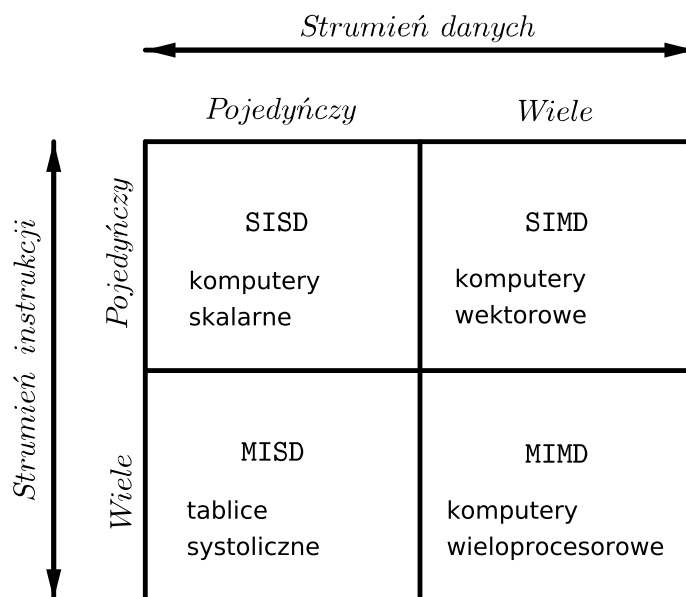
Przykład 1.1. Prostym przykładem algorytmu równoległego jest serwer sieciowy, który każde zapytanie przychodzące przetwarza niezależnie od innych zapytań. Innym przykładem są wielozadaniowe systemy operacyjne radzące sobie z jednoczesną obsługą kilku uruchomionych programów.

1.3 Architektury równoległe

Definicja 1.17 (Architektura równoległa). **Architektura równoległa** jest to architektura wieloprocessorowa, na której można wykonywać przetwarzanie równoległe [7].

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.



Rysunek 1.3: Klasyfikacja Flynna

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.
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.[8]

Klasyfikacja Flynna

Architektury komputerowe można podzielić na klasy ze względu na liczbę równoległe wykonywanych instrukcji – strumieni instrukcji oraz dostępnych strumieni danych¹. 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 pojedynczy strumień instrukcji i przetwarzających pojedynczy strumień danych. Są to komputery całkowicie sekwencyjne, które nie wykonują żadnych obliczeń równoległych.

¹Strumieniem instrukcji nazywamy sekwencję instrukcji wykonywanych przez procesor, zaś strumień danych to sekwencja danych przetwarzanych przez strumień instrukcji.

SIMD. Klasa SIMD (ang. *Single Instruction, Multiple Data*) odnosi się do komputerów obsługujących pojedynczy strumień instrukcji i przetwarzających wiele strumieni danych. Na różnych zbiorach danych wykonywane 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*) odnosi się do komputerów wykonujących jednocześnie wiele instrukcji przetwarzających jeden wspólny strumień danych. Przykładem takiej architektury jest tablica systoliczna². 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 multiprocesory³ (większość współczesnych komputerów PC) i multikomputery⁴.

Większość obecnie używanych komputerów równoległych to klastry o architekturze mieszanej. Klaster jest układem niezależnych jednostek obliczeniowych (*węzłów*) połączonych szybką siecią komunikacyjną.

1.4 Reprezentacja algorytmów

Wiele obliczeń możemy reprezentować za pomocą acyklicznych grafów skierowanych. W takim modelu każde wejście jest oznaczane przez węzeł bez dochodzących do niego łuków. Operacje oznaczane są przez węzły do których wchodzi 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 tutaj, że każdy węzeł przedstawia operację, która wymaga jednej jednostki czasu wykonania.

²Nazwa pochodzi od skurczu mięśni serca przez analogię „pompowania” danych do jednostek przetwarzających na wzór krwi w naczyniach krwionośnych.

³Komputery z wieloma jednostkami centralnymi przyłączonymi do pamięci współdzielonej (ang. *shared memory*.)

⁴Wiele 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 czasowych. Przy takim modelu 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 $j_i \leq p$ oznacza indeks procesora, zaś t_i jednostkę czasu, taką że zachodzą poniższe warunki:

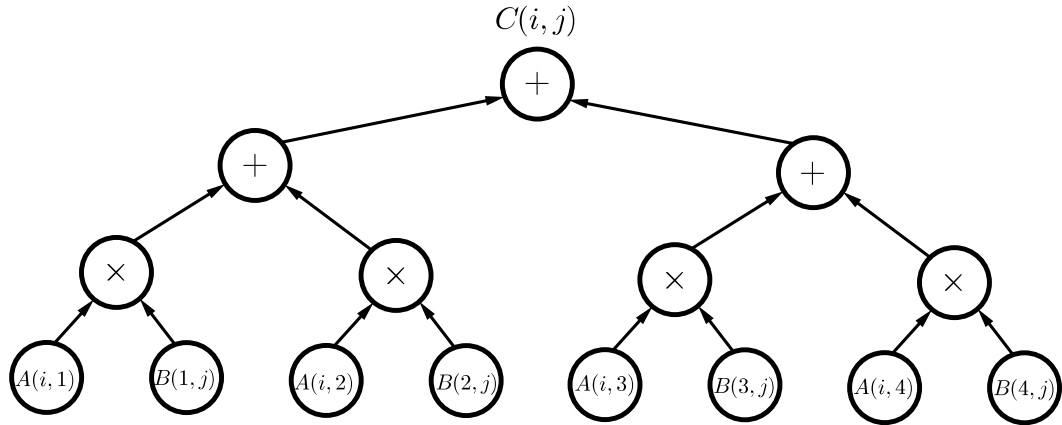
1. Jeśli $t_i = t_k$ dla pewnego $i \neq k$, to $j_i \neq j_k$. Oznacza to, że każdy procesor może wykonać pojedynczą operację podczas każdej jednostki czasu.
2. Jeśli (i, k) jest łukiem grafu, to $t_k \geq t_i + 1$. Oznacza to, że operacja, którą przedstawia węzeł k powinna być zaplanowana po wykonaniu 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 algorytm obliczający iloczyn macierzy $\mathbf{AB} = \mathbf{C}$. Każdy element $C(i, j)$ obliczamy za pomocą wyrażenia $C(i, j) = \sum_{l=1}^n A(i, l)B(l, j)$. Odpowiadający temu obliczeniu DAG dla $n = 4$ przedstawia rys. 1.4. Mając n^3 procesorów, poszczególne operacje mogą być zaplanowane poziom po poziomie, używając n procesorów do obliczenia każdego z elementów macierzy wynikowej \mathbf{C} . Stąd widać, że możemy zaplanować DAG do obliczenia o złożoności $O(\log n)$



Rysunek 1.4: Iloczyn macierzowy, algorytm w postaci grafu.

1.5 Ocena algorytmów

1.5.1 Złożoność 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 $\forall n \geq n_0 : (T(n) \leq cf(n))$
2. $T(n) = \Omega(f(n))$, jeśli istnieje dodatnie stałe c i n_0 takie, że $\forall n \geq n_0 : (T(n) \geq cf(n))$
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[9]). Załóżmy że algorytm równoległy R rozwiązuje problem P o rozmiarze n . **Pesymityczną**

złożonością czasową algorytmu równoległego R nazywamy funkcję:

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

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 licząc czas jaki zajmuje wykonanie go na jednym procesorze i porównanie wyniku z wykonaniem tego samego zadania równoległe na N procesorach.

Definicja 1.20 (Przyspieszenie bezwzględne[9]). Niech P będzie pewnym zadaniem obliczeniowym, n – rozmiarem danych wejściowych. Wówczas

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

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 względne[9]). Niech P będzie pewnym zadaniem obliczeniowym, n – rozmiarem danych wejściowych. Wówczas

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

gdzie $T_1(n)$ jest pesymistyczną złożonością czasową algorytmu równoległego R rozwiązującego problem P na jednym procesorze, $T_p(n)$ 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[9]). Niech $T_p(n)$ będzie pędzie pesymistyczną złożonością obliczeniową algorytmu R dla p procesorów. Wówczas funkcję

$$(1.4) \quad C_p(n) = pT_p(n)$$

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

$$(1.5) \quad C_p^O(n) = C_p(n) - T^*(n) = pT_p(n) - T^*(n)$$

nazywamy *kosztem organizacji 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:

$$(1.6) \quad C_p(n) = \Theta(T^*(n))$$

1.5.4 Efektywność

Definicja 1.25 (Efektywność[9]). Niech $T_p(n)$ będzie pesymistyczną złożonością czasową algorytmu R dla p procesorów i problemu R o rozmiarze n . Wówczas mamy

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

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 równoległe, $T_1^r(n)$. Inaczej:

$$(1.8) \quad T_1(n) = T_1^s(n) + T_1^r(n)$$

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

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

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 równoległe. 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:

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

gdzie s – część obliczeń w algorytmie które muszą być wykonane sekwencyjnie; p – liczba procesorów.

Otrzymany wzór (1.10) nazywamy **prawem Amdahla**.

Definicja 1.26 (Prawo Amdahla[10]). Niech s będzie częścią operacji w algorytmie R , która musi być wykonana sekwencyjnie, taką że $0 \leq s \leq 1$. Wówczas maksymalne przyspieszenie Ψ osiągalne przez komputer równoległy z p procesorami wykonujący algorytm R spełnia nierówność:

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

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:

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

Powinniśmy się zatem spodziewać przyspieszenia o wartości co najwyżej 4.

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

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

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:

$$(1.12) \quad \Psi_p(n) \leq \frac{\sigma + p\rho}{\sigma + \rho} = \sigma + p\rho = \sigma + p(1 - \sigma) = p + (1 - p)\sigma$$

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

Definicja 1.27 (Prawo Gustafsona i Barsisa[10]). 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:

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

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 $\sigma + \rho = 1$ [9]).

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:

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

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:

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

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

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

Z (1.13) i (1.14):

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

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

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

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

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

Przykład 1.6. Powiedzmy, że testując algorytm równoległy na $1, 2, \dots, 8$ 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.

Tabela 1.1: Dane dla przykładu 1.6.

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

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 $1, 2, \dots, 8$ 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.

Tabela 1.2: Dane dla przykładu 1.7.

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

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.

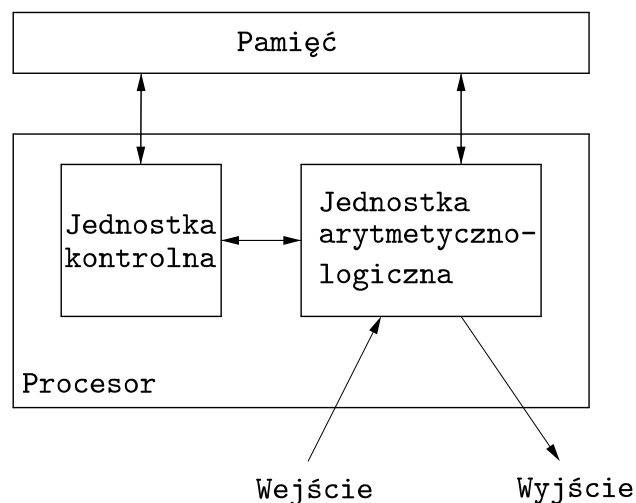
1.6 Teoretyczne modele obliczeń

1.6.1 Model RAM

Nim przejdziemy do omówienia modeli obliczeń równoległych zajmiemy się omówieniem modelu odpowiadającego maszynom sekwencyjnym.

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 (tabela 1.3).
 - (b) pewną liczbę rejestrów arytmetycznych procesora R_1, R_2, \dots, 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 , $i = 1, 2, 3, \dots$ (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[9]

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

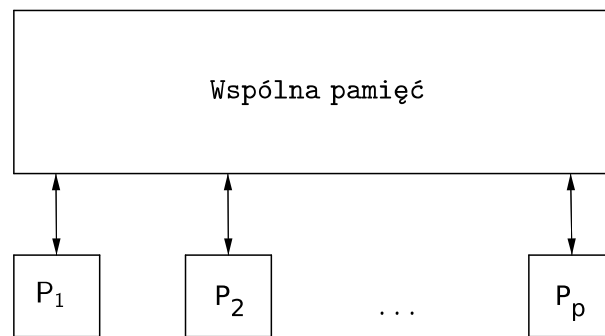
Model wspólnej pamięci PRAM (*Parallel Random Access Machine*) składa się z pewnej liczby procesorów, z których każdy posiada własną pamięć i może lokalnie wykonywać programy. Wszystkie procesory mogą komunikować się za pomocą wspólnej globalnej pamięci (rys. 1.6).

Definicja 1.29 (Złożoność komunikacyjna[9]). Maksymalny rozmiar danych przesłanych w trakcie wykonywania algorytmu PRAM między pamięcią wspólną, a pamięcią lokalną dowolnego procesora nazywamy *złożonością komunikacyjną*.

Każdemu procesorowi przyporządkowana 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 dostępie 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.

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 pamięci.
- **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[11]:

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 wysłać i odbierać dane od innych węzłów sieci w postaci *komunikatów*. Pamięć współdzielona przez węzły (rys. 1.7) nie istnieje w modelu sieciowym.

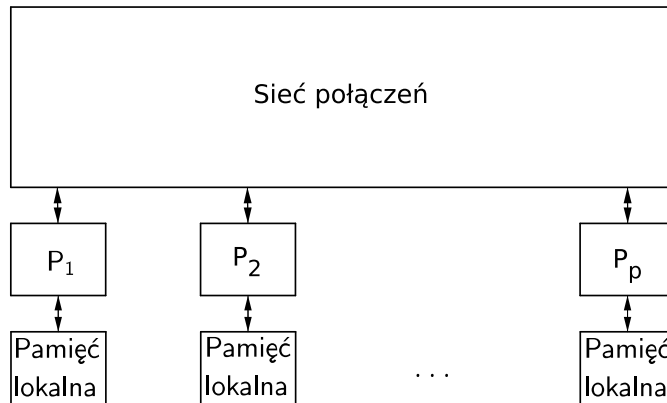
Do każdego węzła sieci przypisany jest numer identyfikacyjny (*pid*). μ -ty węzeł oznaczamy przez P_μ . Powiemy, że P_λ jest *sąsiadem* P_μ , jeśli istnieje bezpośrednie fizyczne połączenie między nimi.[1]

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:

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



Rysunek 1.7: Model sieciowy

2. maksymalny stopień wierzchołka – maksymalna liczba łącz 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 kontynuowane⁵. 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

⁵Dla interfejsu MPI odpowiednikiem jest nieblokująca funkcja `MPI_Isend`.

odebrany⁶. Jeśli odbieranie jest zakończone, komunikat jest zapisywany w pamięci lokalnej, a węzeł P_μ kontynuuje wykonywane programu.

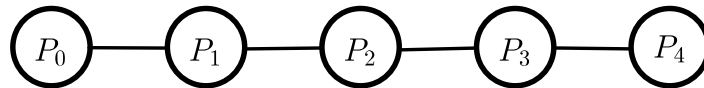
Uwaga 1.6. Zaproponowana notacja nie uwzględnia pewnych istotnych detali:

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.

Przykłady

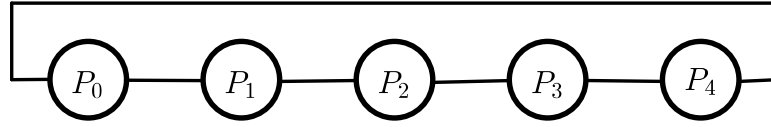


Rysunek 1.8: Siatka jednowymiarowa (liniowa)

Przykład 1.8 (Sieć liniowa). Model składa się z p węzłów P_1, P_2, \dots, 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 liniową z połączonymi końcami (rys. 1.9).

⁶W interfejsie MPI odpowiednikiem tej funkcji jest `MPI_Recv`.

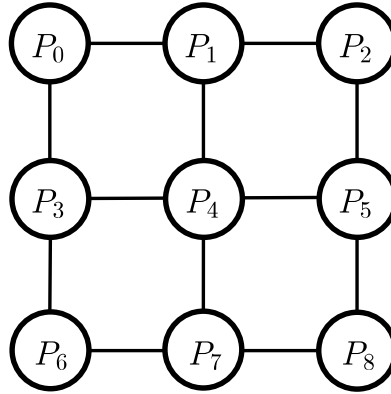


Rysunek 1.9: Torus jednowymiarowy

Tabela 1.4: Wybrane parametry charakteryzujące przykładowe sieci połączeń

Rodzaj sieci	Średnica	Maksymalny stopień	Szerokość 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	$\log p$	$\log p$	$p/2$	$\log p$	$\frac{p \log p}{2}$

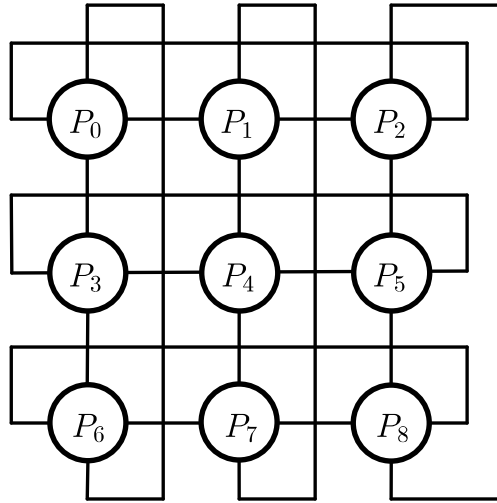
Przykład 1.10 (Siatka dwuwymiarowa). Dwuwymiarowa siatka jest dwuwymiarową wersją sieci liniowej. Składa się ona z $p = m^2$ procesorów ułożonych w siatkę $m \times m$ taką, że procesor $P_{i,j}$ jest połączony z procesorem $P_{i\pm 1,j}$ i $P_{i,j\pm 1}$. Średnica takiej sieci złożonej z $p = m^2$ procesorów wynosi \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*.

Definicja 1.30 (Kostka Boola). Niech $i_{d-1}i_{d-2}\dots i_0$, gdzie $0 \leq i \leq p - 1$ będzie binarną reprezentacją i . Wówczas węzeł i jest połączony z węzłem $P_{i(j)}$, gdzie $i^{(j)} = i_{d-1}\dots \bar{i}_j \dots i_0$ i $\bar{i}_j = 1 - i_j$. Innymi słowy, dwa węzły są ze sobą



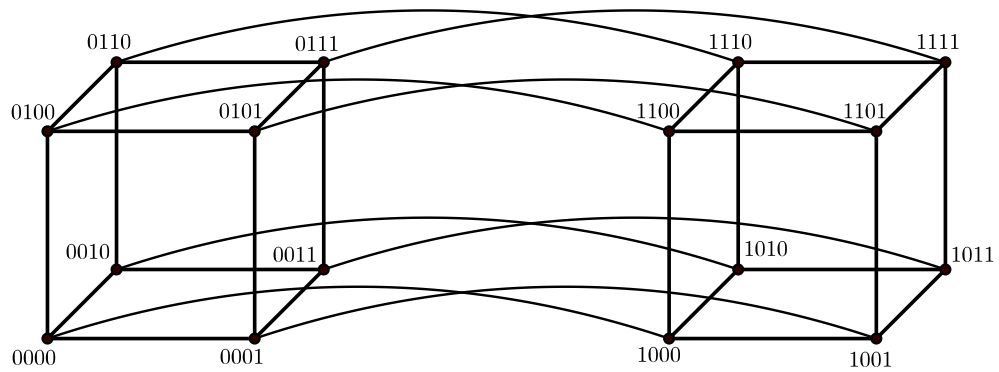
Rysunek 1.11: Torus dwuwymiarowy 3×3

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.

Hipersześcián ma strukturę 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 odległość 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

1.7 Wybrane typy danych dla obliczeń rozproszonych

Powiedzmy, że wektor $x \in \mathbb{R}^n$ jest rozdystrybuowany między pamięci lokalne sieci składającej się z p węzłów. Załóżmy wstępnie, że $n = rp$. Do reprezentacji wektora x rozdystrybuowanego w sieci stosuje się najczęściej dwa następujące podejścia: zapis kolumnowy (ang. *store-by-column*) oraz zapis wierszowy (ang. *store-by-row*).

W pierwszym z nich⁷, zapisie kolumnowym, rozpatrujemy wektor x jako macierz $r \times p$:

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

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

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

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

Każdy *wiersz* jest wówczas zapisany w odpowiednim węźle, tj. $x(\mu:p:n) \in P_\mu$. 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 $p = 4$:

$$(1.18) \quad x^r = [\underbrace{x_1 x_2 x_3 x_4}_{P_0} \mid \underbrace{x_5 x_6 x_7 x_8}_{P_1} \mid \underbrace{x_9 x_{10} x_{11}}_{P_2} \mid \underbrace{x_{12} x_{13} x_{14}}_{P_3}]$$

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

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

⁷Używamy tutaj notacji programu Matlab; więcej informacji można znaleźć w [12]

Tabela 1.5: Sposoby reprezentacji macierzy w sieci z q węzłami

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

Metody dla macierzy blokowych są analogiczne do tych z tabeli 1.5.

Rozdział 2

Mnożenie macierzy

2.1 Przegląd algorytmów klasycznych

W swojej pracy „*Gaussian Elimination is not Optimal*” z 1969 roku VOLKER STRASSEN pokazał rekursywny algorytm mnożenia macierzy kwadratowych wymiaru $m2^k$ o złożoności $\mathcal{O}(n^{2,81})$ [2]. 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[3][13] do najczęściej implementowanej dzisiaj postaci algorytmu STRASSENA-WINOGRADA (implementacja zawiera się na przykład w bibliotece GEMMW[14]). W przypadku macierzy 2×2 wykonuje on 7 operacji mnożenia i 15 dodawań[5].

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  $i \leftarrow 1, n$  do
  for  $j \leftarrow 1, n$  do
    for  $k \leftarrow 1, n$  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:

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

gdzie

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

(2.1) możemy wyrazić inaczej:

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

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:      $C_{11} \leftarrow A_{11}B_{11}$ 
4:   else
5:      $C_{11} = \text{MULTIPLY}(A_{11}B_{11}) + \text{MULTIPLY}(A_{12}B_{21})$ 
6:      $C_{12} = \text{MULTIPLY}(A_{11}B_{12}) + \text{MULTIPLY}(A_{12}B_{22})$ 
7:      $C_{21} = \text{MULTIPLY}(A_{21}B_{11}) + \text{MULTIPLY}(A_{22}B_{21})$ 
8:      $C_{22} = \text{MULTIPLY}(A_{21}B_{12}) + \text{MULTIPLY}(A_{22}B_{22})$ 
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

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

Algorytm Strassena-Winograda

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

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

Następnie ustalmy odpowiednio po siedem kombinacji liniowych T_i, S_i , $i \in \{1, 2, \dots, 7\}$ 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

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

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

W praktyce stosuje się tylko kilka kroków algorytmu Strassena-Winegrada[5]. Złożoność obliczeniową $O(n^{w_0})$ algorytmu oznacza, że jego wykonanie zatrzymuje się po osiągnięciu macierzy wymiaru 1×1 .

2.2 Algorytmy w modelu z pamięcią wspólną

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 $P_{i,j,l}$, $1 \leq i, j, l \leq n$ maszyny CREW PRAM.

Algorytm 3 Algorytm mnożenia macierzy dla n^3 procesorów.[9] (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** $P_{i,j,k}$, $1 \leq i, j, k \leq n$ **do**
 - 2: $T_{i,j,k} \leftarrow A_{i,k} B_{k,j}$ \triangleright Obliczanie składowych iloczynów skalarnych
 - 3: **end parfor**
 - 4: **for** $l \leftarrow 1, \log n$ **do** \triangleright Sumowanie składowych iloczynów skalarnych
 - 5: **parfor** $P_{i,j,k}$, $1 \leq i, j \leq n$, $1 \leq k \leq n/2^l$ **do**
-

Algorytm 3 Algorytm mnożenia macierzy dla n^3 procesorów.[9] (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  $P_{i,j,k}$ ,  $1 \leq i, j \leq n$ ,  $k = 1$  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 $P_{i,1,l}, P_{i,2,l}, \dots, P_{i,n,l}$ 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[9]:

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

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 później, jego koszt jest optymalny.

Algorytm 4 Algorytm mnożenia macierzy dla n procesorów.[9] (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  $P_i$ ,  $1 \leq i \leq n$  do
2:   for  $j \leftarrow 1, n$  do
3:     for  $k \leftarrow 1, n$  do
```

Algorytm 5 Algorytm mnożenia macierzy dla n procesorów.[9] (cz. II)

```
4:       $T_{i,j,k} \leftarrow A_{i,k}B_{k,j}$   $\triangleright$  Obliczanie składowych iloczynów skalarnych
5:      end for
6:  end for
7: end parfor
8: for  $l \leftarrow 1, \log n$  do  $\triangleright$  Sumowanie składowych iloczynów skalarnych
9:   parfor  $P_i, 1 \leq i \leq n$  do
10:    for  $j \leftarrow 1, n$  do
11:     for  $k \leftarrow 1, n/2^l$  do
12:       $T_{i,j,k} \leftarrow T_{i,j,2k-1} + T_{i,j,2k}$ 
13:    end for
14:  end for
15: end parfor
16: end for
17: parfor  $P_i, 1 \leq i \leq n$  do
18:   for  $j \leftarrow 1, n$  do
19:     $C_{i,j} \leftarrow T_{i,j,1}$ 
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 $P_i, (1 \leq i \leq n)$ 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 $2n^2(1 - \frac{1}{2^r})$ 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 $n + n^2 + n^2 + 3(2n^2(1 - \frac{1}{2^r})) + 2n = \mathcal{O}(n^2)$.

2.3 Algorytm Cannona

Wprowadzenie

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

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

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

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

Założmy, 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 uwzględnieniem 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} przesuwane 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 na zatrzymanie wykonania programu dopóki macierz A_{loc} nie zostanie wysłana¹.

Rozważymy teraz aktywność węzłów P_{12} , P_{13} , P_{21} , P_{31} . W dotychczasowych rozważaniach 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.

¹W interfejsie MPI obydwa algorytmy traktowane literalnie wywołują zająknięcie (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}	$t = 1$
–	B_{31}	–	B_{12}	–	B_{23}	
–	B_{11}	–	B_{22}	–	B_{33}	

A_{13}	B_{31}	A_{11}	B_{12}	A_{12}	B_{23}	$t = 2$
–	B_{11}	–	B_{22}	–	B_{33}	
–	B_{21}	–	B_{32}	–	B_{13}	

A_{11}	B_{11}	A_{12}	B_{22}	A_{13}	B_{33}	$t = 3$
–	B_{21}	–	B_{32}	–	B_{13}	
–	B_{31}	–	B_{12}	–	B_{23}	

Rysunek 2.5: Rozmieszczenie danych dla trzech kroków metody Cannona z uwzględnieniem 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}	$t = 1$
A_{23}	B_{31}	A_{23}	B_{12}	A_{22}	B_{23}	
A_{31}	B_{11}	A_{32}	B_{22}	A_{33}	B_{33}	
A_{13}	B_{31}	A_{11}	B_{12}	A_{12}	B_{23}	$t = 2$
A_{21}	B_{11}	A_{22}	B_{22}	A_{23}	B_{33}	
A_{32}	B_{21}	A_{33}	B_{32}	A_{31}	B_{13}	
A_{11}	B_{11}	A_{12}	B_{22}	A_{13}	B_{33}	$t = 3$
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 2.7: Rozmieszczenie danych dla trzech kroków metody Cannona w dwuwymiarowym torusie 3×3 .

Teraz jesteśmy gotowi do przedstawienia ogólnej wersji algorytmu Cannona.

Uogólnienie

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 $i - 1$ pozycji². Podobnie, j -tą kolumnę węzłów przesuwamy w górę o $j - 1$ pozycji. Powyższe rozważania prowadzą do sformułowania algorytmu 8.

Algorytm 8 Algorytm Cannona[1] (cz. I)

Dane pomocnicze: $p_1, (\mu, \lambda)$ (współrzędne węzła w sieci), $gra, d, lewo, prawo$ (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** \triangleright Wstępne przesunięcie $A_{\mu j}$ i $B_{i\lambda}$.
 - 2: SEND(A_{loc} , lewo)
 - 3: RCV(A_{loc} , prawo)
-

²Numerację elementów zaczynamy od 1.

Algorytm 8 Algorytm Cannona[1] (cz. II)

```
4: end for
5: for  $k \leftarrow 1, \lambda - 1$  do
6:   SEND( $B_{loc}$ , góra)
7:   RECV( $B_{loc}$ , dół)
8: end for
9: for  $k \leftarrow 1, p_1$  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  $k \leftarrow 1, \mu - 1$  do                                ▷ Doprowadzanie rozkładu podmacierzy
17:  SEND( $A_{loc}$ , prawo)                                          ▷ do stanu początkowego
18:  RECV( $A_{loc}$ , lewo)
19: end for
20: for  $k \leftarrow 1, \lambda - 1$  do
21:  SEND( $B_{loc}$ , dół)
22:  RECV( $B_{loc}$ , góra)
23: end for
```

Analiza algorytmu

Powiedzmy, że A, B i C są macierzami wymiaru $n \times n$ oraz n jest wielokrotnością \sqrt{p} . Wówczas każdy proces odpowiada za obliczenie części macierzy wynikowej C o wymiarach $(n/\sqrt{p}) \times (n/\sqrt{p})$.

W trakcie każdej interakcji algorytmu każdy proces mnoży blok wymiaru $(n/\sqrt{p}) \times (n/\sqrt{p})$ macierzy A z blokiem $(n/\sqrt{p}) \times (n/\sqrt{p})$ macierzy B . Wynik takiego mnożenia jest dodawany jako iloczyn częściowy do bloku wynikowego macierzy C . Jeśli przez χ oznaczymy czas potrzebny do wykonania jednej operacji dodawania i mnożenia w jednym węźle, wówczas złożoność czasowa każdej iteracji wynosi:

$$\chi(n/\sqrt{p})^3 = \chi n^3/p^{3/2}$$

Algorytm kończy się po p iteracjach. Stąd czas wykonania powinien wynosić:

$$(2.6) \quad T_p(n) = \sqrt{p}\chi n^3/p^{3/2} = \chi n^3/p$$

Jeśli $p = n^2$, wówczas algorytm wykonuje się w czasie liniowym.

Przed każdą iteracją proces musi wysłać blok macierzy A i blok macierzy B do odpowiedniego procesu i odebrać blok macierzy A i B . Oznaczmy przez $1/\beta$ czas potrzebny do przesłania jednego elementu macierzy. Czas potrzebny do wstępnej dystrybucji macierzy w sieci możemy oszacować przez:

$$(2.7) \quad 2 \left(\lambda + \frac{n^2}{p\beta} \right)$$

Poprzez \sqrt{p} iteracji każdy proces musi wysłać swoje bloki macierzy A i B oraz odebrać nowe bloki, które później pomnoży. Całkowity czas potrzebny do komunikacji między procesami w trakcie wykonywania algorytmu możemy oszacować przez:

$$(2.8) \quad 2\sqrt{p} \left(\lambda + \frac{n^2}{p\beta} \right)$$

Dodając do siebie wyrażenia (2.6), (2.7), (2.8) możemy oszacować całkowity czas wykonania algorytmu Cannona:

$$\chi n^3/p^{3/2} + 2(\sqrt{p} + 1) \left(\lambda + \frac{n^2}{p\beta} \right)$$

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 .

Podsumowując: złożoność, przyspieszenie, koszt i efektywność algorytmu 8 są następujące[9]:

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

Zgodnie z def. 1.24 algorytm jest optymalny względem kosztu.

Przykład 2.1 wygenerowano za pomocą własnej implementacji algorytmu Cannona wchodzącego w skład programu `pmm`, który przybliżamy w części A.2.

Przykład 2.1. Pokażmy iloczyn dwóch macierzy wymiaru 6×6 metodą Cannona w torusie 3×3 procesorów. Niech:

$$A = \begin{pmatrix} 8 & 5 & 6 & 1 & 2 & 3 \\ 3 & 3 & 1 & 5 & 3 & 9 \\ 9 & 2 & 9 & 0 & 4 & 9 \\ 2 & 0 & 8 & 8 & 3 & 4 \\ 6 & 7 & 6 & 7 & 5 & 0 \\ 2 & 5 & 7 & 8 & 7 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 9 & 2 & 6 & 8 & 8 \\ 1 & 6 & 0 & 1 & 6 & 7 \\ 2 & 2 & 4 & 9 & 6 & 1 \\ 6 & 8 & 5 & 4 & 4 & 5 \\ 7 & 2 & 3 & 1 & 0 & 9 \\ 1 & 8 & 0 & 6 & 6 & 8 \end{pmatrix}.$$

Poszczególne kroki algorytmu pokazane są na rysunku 2.8.

$t = 1$

$\begin{pmatrix} 5 & 9 \\ 1 & 6 \end{pmatrix}$	$\begin{pmatrix} 8 & 5 \\ 3 & 3 \end{pmatrix}$	$\begin{pmatrix} 67 & 52 \\ 26 & 23 \end{pmatrix}$	$\begin{pmatrix} 2 & 6 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 9 & 0 \\ 8 & 8 \end{pmatrix}$	$\begin{pmatrix} 66 & 48 \\ 8 & 8 \end{pmatrix}$	$\begin{pmatrix} 8 & 8 \\ 6 & 7 \end{pmatrix}$	$\begin{pmatrix} 5 & 0 \\ 7 & 1 \end{pmatrix}$	$\begin{pmatrix} 96 & 8 \\ 79 & 7 \end{pmatrix}$
$\begin{pmatrix} 4 & 9 \\ 5 & 4 \end{pmatrix}$	$\begin{pmatrix} 9 & 2 \\ 2 & 0 \end{pmatrix}$	$\begin{pmatrix} 54 & 8 \\ 53 & 10 \end{pmatrix}$	$\begin{pmatrix} 6 & 1 \\ 4 & 5 \end{pmatrix}$	$\begin{pmatrix} 6 & 9 \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 43 & 50 \\ 59 & 68 \end{pmatrix}$	$\begin{pmatrix} 2 & 2 \\ 6 & 8 \end{pmatrix}$	$\begin{pmatrix} 2 & 3 \\ 3 & 9 \end{pmatrix}$	$\begin{pmatrix} 10 & 24 \\ 36 & 90 \end{pmatrix}$
$\begin{pmatrix} 0 & 9 \\ 6 & 8 \end{pmatrix}$	$\begin{pmatrix} 6 & 7 \\ 2 & 5 \end{pmatrix}$	$\begin{pmatrix} 18 & 45 \\ 52 & 82 \end{pmatrix}$	$\begin{pmatrix} 7 & 2 \\ 1 & 8 \end{pmatrix}$	$\begin{pmatrix} 6 & 1 \\ 1 & 5 \end{pmatrix}$	$\begin{pmatrix} 44 & 17 \\ 14 & 41 \end{pmatrix}$	$\begin{pmatrix} 3 & 1 \\ 0 & 6 \end{pmatrix}$	$\begin{pmatrix} 4 & 9 \\ 3 & 4 \end{pmatrix}$	$\begin{pmatrix} 15 & 31 \\ 18 & 24 \end{pmatrix}$

$t = 2$

$\begin{pmatrix} 2 & 6 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 9 & 2 \\ 2 & 0 \end{pmatrix}$	$\begin{pmatrix} 97 & 56 \\ 28 & 23 \end{pmatrix}$	$\begin{pmatrix} 8 & 8 \\ 6 & 7 \end{pmatrix}$	$\begin{pmatrix} 6 & 7 \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 170 & 168 \\ 93 & 106 \end{pmatrix}$	$\begin{pmatrix} 5 & 9 \\ 1 & 6 \end{pmatrix}$	$\begin{pmatrix} 2 & 3 \\ 3 & 9 \end{pmatrix}$	$\begin{pmatrix} 133 & 104 \\ 99 & 64 \end{pmatrix}$
$\begin{pmatrix} 6 & 1 \\ 4 & 5 \end{pmatrix}$	$\begin{pmatrix} 6 & 7 \\ 2 & 5 \end{pmatrix}$	$\begin{pmatrix} 92 & 55 \\ 87 & 63 \end{pmatrix}$	$\begin{pmatrix} 2 & 2 \\ 6 & 8 \end{pmatrix}$	$\begin{pmatrix} 6 & 1 \\ 1 & 5 \end{pmatrix}$	$\begin{pmatrix} 57 & 62 \\ 103 & 114 \end{pmatrix}$	$\begin{pmatrix} 4 & 9 \\ 5 & 4 \end{pmatrix}$	$\begin{pmatrix} 4 & 9 \\ 3 & 4 \end{pmatrix}$	$\begin{pmatrix} 53 & 96 \\ 68 & 151 \end{pmatrix}$
$\begin{pmatrix} 7 & 2 \\ 1 & 8 \end{pmatrix}$	$\begin{pmatrix} 8 & 5 \\ 3 & 3 \end{pmatrix}$	$\begin{pmatrix} 80 & 86 \\ 84 & 111 \end{pmatrix}$	$\begin{pmatrix} 3 & 1 \\ 0 & 6 \end{pmatrix}$	$\begin{pmatrix} 9 & 0 \\ 8 & 8 \end{pmatrix}$	$\begin{pmatrix} 79 & 25 \\ 62 & 89 \end{pmatrix}$	$\begin{pmatrix} 0 & 9 \\ 6 & 8 \end{pmatrix}$	$\begin{pmatrix} 5 & 0 \\ 7 & 1 \end{pmatrix}$	$\begin{pmatrix} 78 & 40 \\ 104 & 32 \end{pmatrix}$

$t = 3$

$\begin{pmatrix} 8 & 8 \\ 6 & 7 \end{pmatrix}$	$\begin{pmatrix} 6 & 7 \\ 2 & 5 \end{pmatrix}$	$\begin{pmatrix} 161 & 152 \\ 78 & 100 \end{pmatrix}$	$\begin{pmatrix} 5 & 9 \\ 1 & 6 \end{pmatrix}$	$\begin{pmatrix} 6 & 1 \\ 1 & 5 \end{pmatrix}$	$\begin{pmatrix} 209 & 218 \\ 105 & 137 \end{pmatrix}$	$\begin{pmatrix} 2 & 6 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 4 & 9 \\ 3 & 4 \end{pmatrix}$	$\begin{pmatrix} 159 & 146 \\ 102 & 68 \end{pmatrix}$
$\begin{pmatrix} 2 & 2 \\ 6 & 8 \end{pmatrix}$	$\begin{pmatrix} 8 & 5 \\ 3 & 3 \end{pmatrix}$	$\begin{pmatrix} 114 & 71 \\ 159 & 117 \end{pmatrix}$	$\begin{pmatrix} 4 & 9 \\ 5 & 4 \end{pmatrix}$	$\begin{pmatrix} 9 & 0 \\ 8 & 8 \end{pmatrix}$	$\begin{pmatrix} 165 & 134 \\ 180 & 146 \end{pmatrix}$	$\begin{pmatrix} 6 & 1 \\ 4 & 5 \end{pmatrix}$	$\begin{pmatrix} 5 & 0 \\ 7 & 1 \end{pmatrix}$	$\begin{pmatrix} 90 & 97 \\ 123 & 156 \end{pmatrix}$
$\begin{pmatrix} 3 & 1 \\ 0 & 6 \end{pmatrix}$	$\begin{pmatrix} 9 & 2 \\ 2 & 0 \end{pmatrix}$	$\begin{pmatrix} 109 & 92 \\ 96 & 111 \end{pmatrix}$	$\begin{pmatrix} 0 & 9 \\ 6 & 8 \end{pmatrix}$	$\begin{pmatrix} 6 & 7 \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 142 & 97 \\ 154 & 195 \end{pmatrix}$	$\begin{pmatrix} 7 & 2 \\ 1 & 8 \end{pmatrix}$	$\begin{pmatrix} 2 & 3 \\ 3 & 9 \end{pmatrix}$	$\begin{pmatrix} 98 & 79 \\ 130 & 107 \end{pmatrix}$

Rysunek 2.8: Rozmieszczenie danych dla trzech kroków metody Cannona w dwuwymiarowym torusie 3×3 . W każdej komórce pokazano zawartość zmiennych lokalnych odpowiednio A_{loc} , B_{loc} i C_{loc} .

Rozdział 3

Rezultaty doświadczalne

Dane pomiarowe uzyskano na klastrze obliczeniowym Solaris dzięki uprzejmości Instytutu Matematyki UMCS. Specyfikacja techniczna węzłów zawarta została w tabeli 3.1. W trakcie testów używano tylko jednego procesora przypadającego na węzeł. Do budowy programu zawierającego implementacje wybranych algorytmów użyto kompilatora `mpiicc` z bibliotekami `mk1` i `openmp`. Wszystkie rozważane w tym rozdziale przyspieszenia są obliczane względem naiwnego algorytmu sekwencyjnego o złożoności $\mathcal{O}(n^3)$.

Wykresy 3.1, 3.2, 3.3, 3.4 prezentują przyspieszenia wariantów algorytmu Cannona dla $p = 2^k$, $k = 2, 4, 6, 8$ procesów uzyskane przy użyciu 22 węzłów i macierzy wymiarów (2048×2048) , (4096×4096) , (8192×8192) .

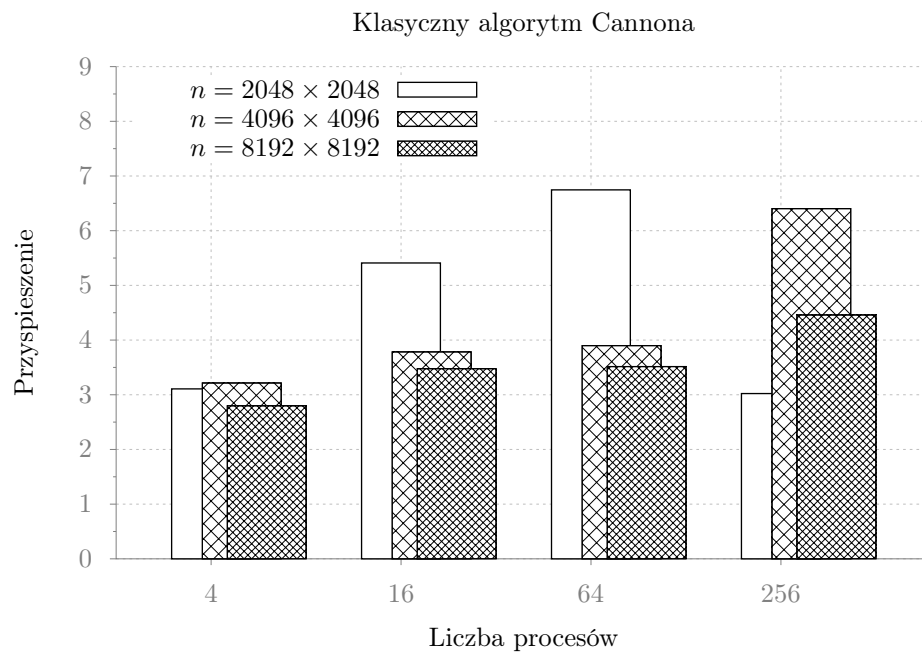
Wykres 3.5 przedstawia przyspieszenia uzyskane w zależności od liczby wątków OpenMP dla jednego procesu MPI wykonanego na 22 fizycznych węzłach sieci dla procedury `cblas_dgemm` z biblioteki MKL oraz dyrektywy `parallel for` interfejsu OpenMP.

Wykresy 3.6 i 3.7 przedstawiają przyspieszenia uzyskane dla obliczeń w topologii torusa (2×2) , (3×3) , (4×4) , ..., (16×16) . Ze względu na specyfikę algorytmu Cannona macierze wejściowe są wstępnie skalowane¹ przed rozesłaniem do węzłów sieci. Zmienia to początkowy rozmiar problemu n do $\left(\lfloor \sqrt{\frac{n}{p}} \rfloor \sqrt{p} + \sqrt{n} \bmod \sqrt{p}\right)^2$.

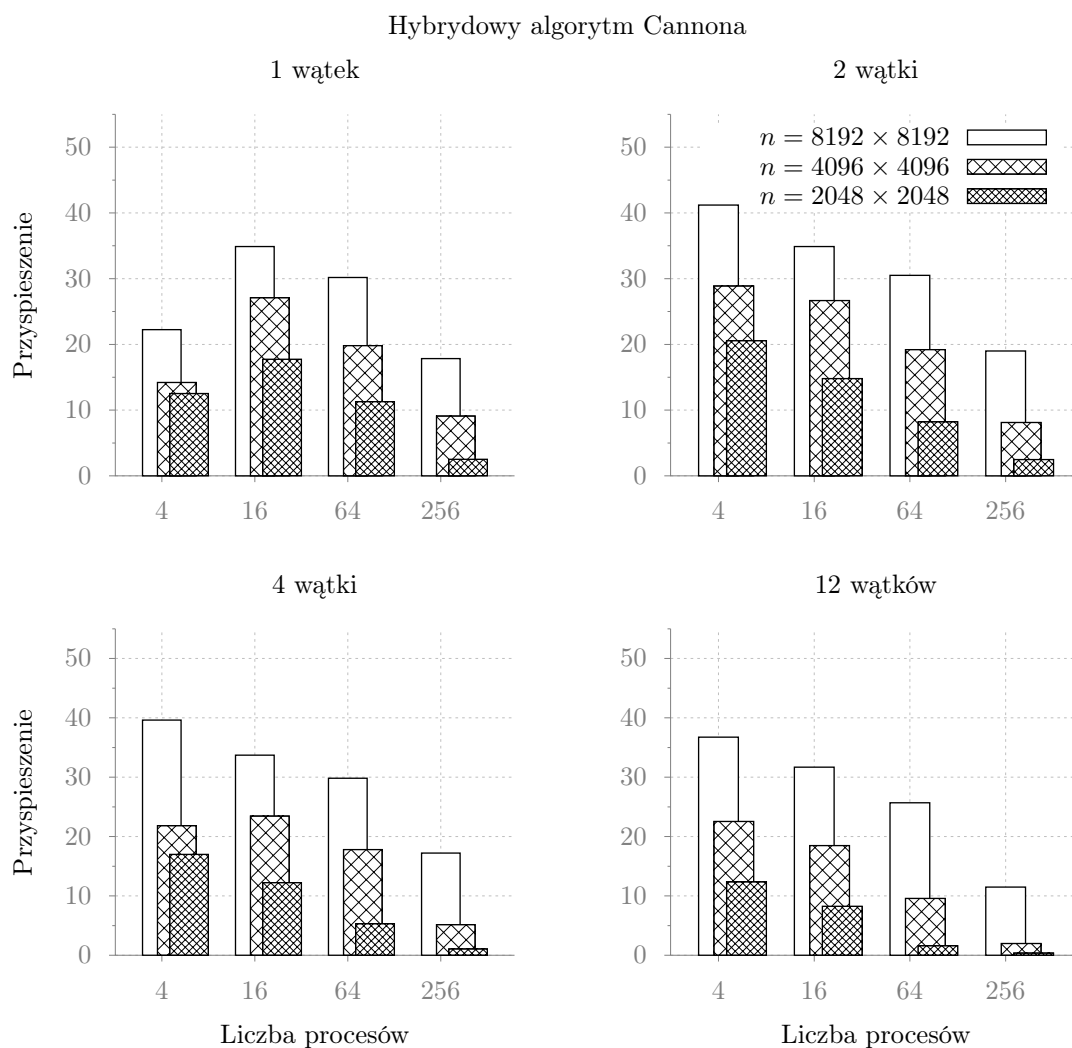
¹Ich rozmiar jest powiększany do wielokrotności \sqrt{p} (gdzie p to liczba procesów) przez uzupełnienie macierzy zerami.

Tabela 3.1: Specyfikacja techniczna węzłów klastra Solaris

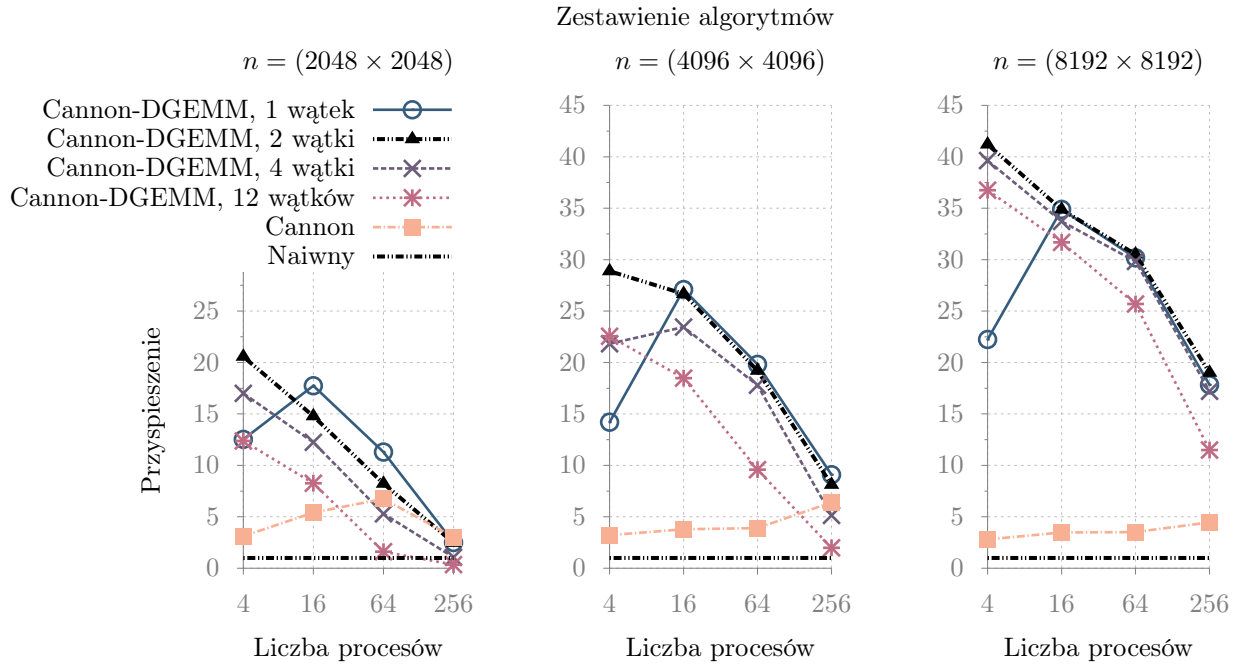
Procesor	2x Intel Xeon X5650 (2,66 GHz, 6 rdzeni, 12 wątków)
Pamięć lokalna	48GB DDR3
System operacyjny	Debian 3.2.39-2 (Linux 3.2.0-4-amd64 x86_64 kernel)



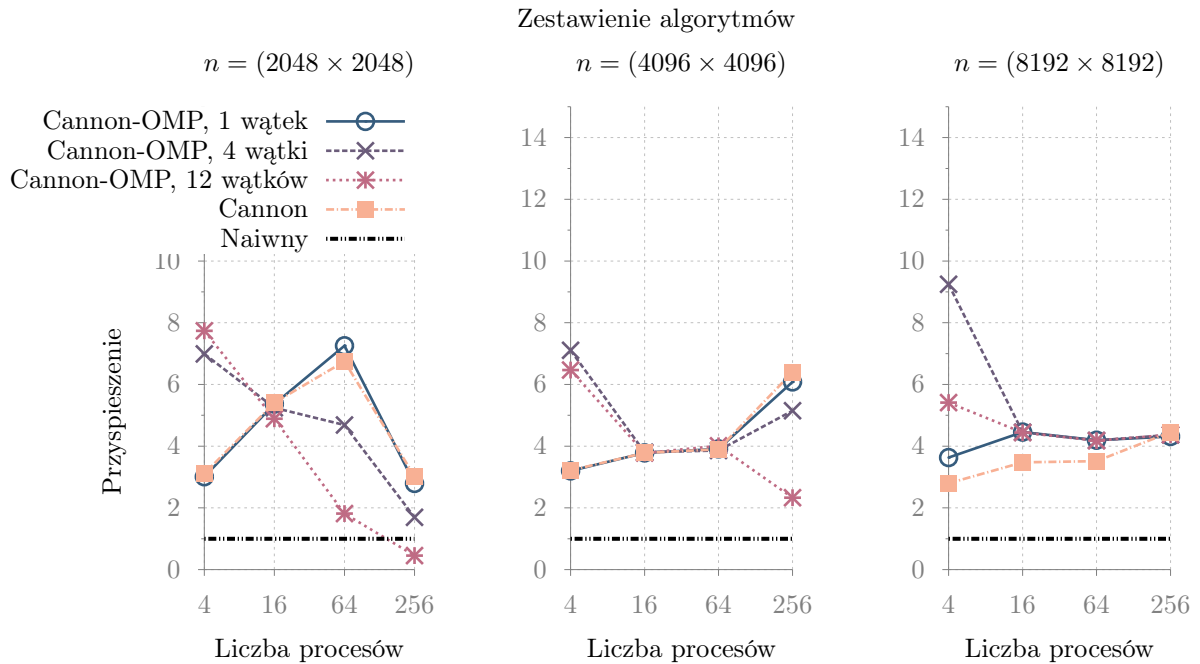
Rysunek 3.1: Przyspieszenie algorytmu Cannona na systemie równoległym składającym się z 22 fizycznych węzłów sieci. Wykres uwzględnia testy przeprowadzone na trzech zestawach danych rozmiaru (2048×2048) , (4096×4096) , (8192×8192) .



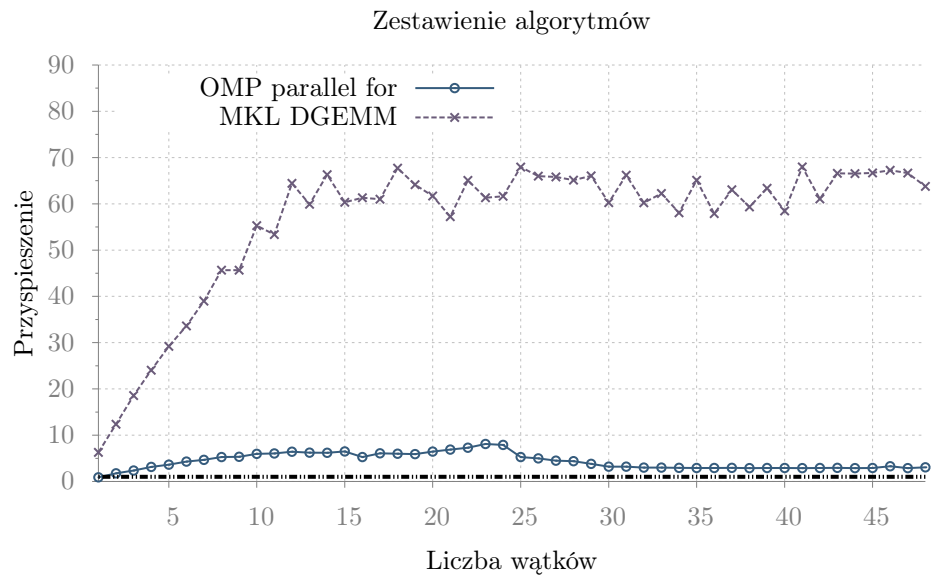
Rysunek 3.2: Przyspieszenie względne hybrydowego algorytmu Cannona z procedurą `cblas_dgemm` uzyskane na systemie równoległym składającym się z 22 fizycznych węzłów sieci dla trzech zestawów danych różnego rozmiaru.



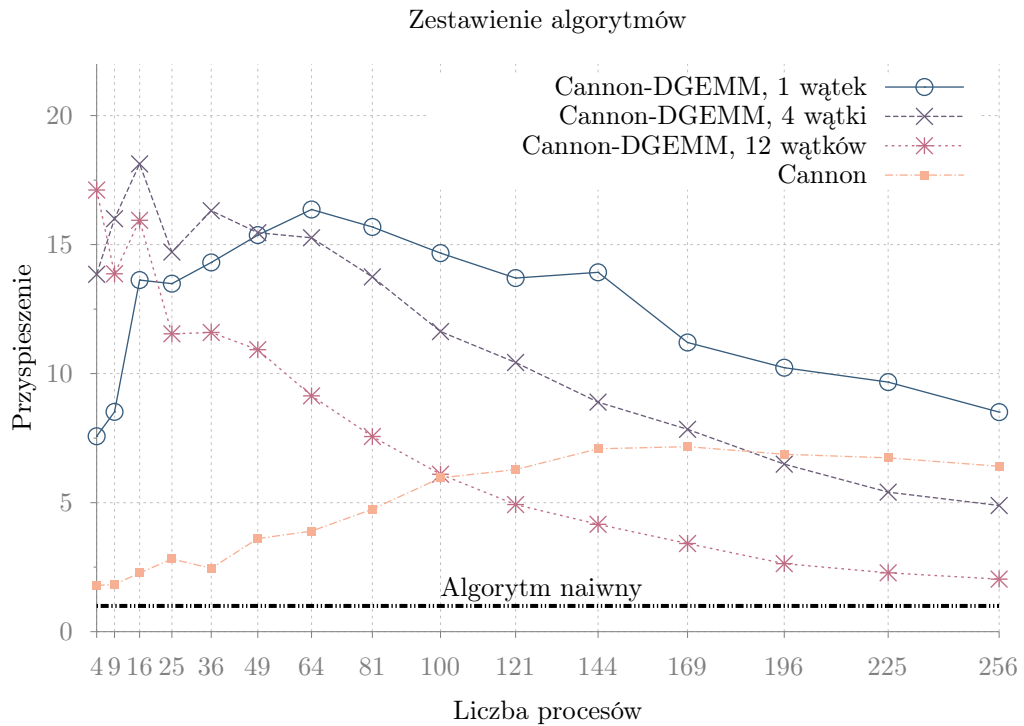
Rysunek 3.3: Zestawienie porównawcze przyspieszeń z wykresów 3.1 i 3.2.



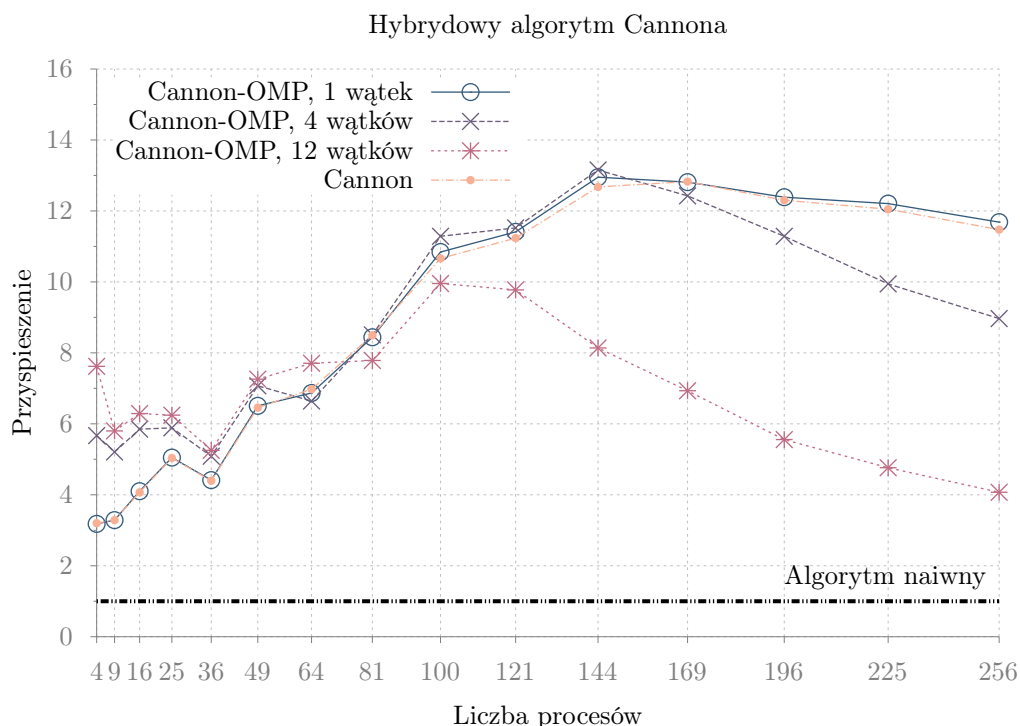
Rysunek 3.4: Porównanie wydajności klasycznego algorytmu Cannona oraz wersji hybrydowej z dyrektywą `parallel for` dla różnej liczby wątków w systemie równoległym składającym się z 22 fizycznych węzłów.



Rysunek 3.5: Porównanie wydajności procedury `cblas_dgemm` biblioteki MKL do mnożenia z pomocą dyrektywy `parallel for` interfejsu OpenMP. Wyniki uzyskano na systemie równoległym składającym się z 22 fizycznych węzłów.



Rysunek 3.6: Przyspieszenie hybrydowego algorytmu Cannona z operacją `cblas_dgemm` (na wykresie Cannon-DGEMM) dla macierzy 4096×4096 wykonany na systemie równoległym złożonym z 8 węzłów.



Rysunek 3.7: Przyspieszenie hybrydowego algorytmu Cannona z wykonaniem mnożenia z dyrektywą `parallel for` interfejsu OpenMP na poziomie procesu (na wykresie Cannon-OMP) dla macierzy 4096×4096 wykonany na systemie równoległym złożonym z 8 węzłów. Rozmiar problemu ze względu na wstępne skalowanie macierzy jest zmienny i wynosi $\left(\lfloor \sqrt{\frac{n}{p}} \rfloor \sqrt{p} + \sqrt{n} \bmod \sqrt{p}\right)^2$.

Wnioski

Przy wykonywaniu obliczenia iloczynu macierzy (2048×2048) klasyczną metodą Cannona (rys. 3.1) z uruchomionymi 64 procesami na 22 węzłach osiągamy optymalne (aczkolwiek stosunkowo niskie) dla tej implementacji przyspieszenie. Wiedząc, że każdy z 22 użytych węzłów dysponuje procesorem sześciordzeniowym widzimy, że dla liczby procesów $p > 132$ obliczenia nakładają się na siebie i nie wykonują równolegle. Stąd obserwujemy znaczny spadek przyspieszenia dla liczby procesów $p = 256$ (rys. 3.1). Dla macierzy większych wymiarów widzimy, że wykonywanie współbieżne z liczbą procesów $p > 132$, w którym duża część procesów nie jest wykonywana równolegle, ma korzystny wpływ na przyspieszenie.

Lepiej pod względem przyspieszenia zachowują się algorytmy przedstawione na wykresach 3.2, 3.3. Widzimy, że dla obliczeń macierzy (8192×8192) najoptymalniejsze jest przeprowadzenie ich w konfiguracji po dwa wątki na

cztery procesy. Wyraźnie widoczny jest też spadek przyspieszenia przy przeprowadzaniu obliczeń na zbyt dużej liczbie procesów i wątków.

Na wykresach 3.4 widzimy, że dla metody Cannona z obliczeniami z dyrektywą `parallel for` na poziomie procesu, wydajność obliczeń jest najwyższa przy konfiguracji czterech wątków na cztery procesy. W pozostałych przypadkach (pomijając te, w których ponosimy koszty nakładania się obliczeń przez uruchomienie zbyt wielkiej liczby procesów lub wątków) algorytm jest nieznacznie szybszy od klasycznej metody Cannona.

Z wykresu 3.5 widzimy, że dla 22 wątków procedura oparta o dyrektywę `parallel for` osiąga maksymalne przyspieszenie. Procedura `cblas_dgemm` od dziesięciu wątków w górę osiąga stosunkowo stałe przyspieszenie.

Najszybszą metodą, zgodnie z oczekiwaniami, okazało się wykonanie obliczeń procedurą `cblas_dgemm` z biblioteki Intel MKL przy wykorzystaniu wielowątkowości biblioteki OpenMP. Osiągała ona przyspieszenia dwukrotnie większe niż hybrydowy algorytm Cannona korzystający z tej procedury w obliczeniach na poziomie procesu. Uzyskany słaby wynik metody Cannona należy tłumaczyć tutaj przeznaczeniem tej metody do obliczeń na dużych zbiorach danych (obliczeń typu *big data*).

Dodatek A

Implementacja

A.1 Opis programu

A.1.1 Wprowadzenie

W ramach pracy licencjackiej zaimplementowano w języku C prosty system na potrzeby testowania wybranych algorytmów obliczania iloczynu macierzowego. Program wykorzystuje interfejsy programowania równoległego MPI i OpenMP oraz bibliotekę matematyczną Intel MKL. Użyto interfejsu `argp` do parsingu argumentów wejściowych oraz wyświetlania opcji `--help` i `--version` w stylu GNU (listing 1, listing 2). Możliwe jest wykonanie:

1. Naiwnego algorytmu sekwencyjnego (opcja `--method=SEQUENTIAL`),
2. Mnożenia z operacją `parfor` (opcja `--method=OMP`),
3. Mnożenia z wykorzystaniem operacji `cblas_dgemm` z biblioteki Intel MKL (opcja `--method=MKL`)
4. Klasycznego algorytmu Cannona (opcja `--method=CANNON`),
5. Algorytmów hybrydowych:
 - (a) Algorytmu Cannona z drobnoziarnistą operacją `parfor` (opcja `--method=CANNON_OMP`),
 - (b) Algorytmu Cannona z drobnoziarnistą operacją `cblas_dgemm` z biblioteki Intel MKL, (opcja `--method=CANNON_DGEMM`).

Program `gen` powstał na potrzeby szybkiego generowania dużych macierzy gęstych w zapisie wierszowym (patrz 1.7). Wygenerowane przez program pliki

składają się z ciągów liczb pseudolosowych o pewnych zadanych własnościach (patrz listing 2). Implementacja algorytmu Cannona, chociaż sam algorytm w oryginalnej wersji pracuje tylko na macierzach kwadratowych o rozmiarze wielokrotności szerokości lub długości siatki procesów, pozwala pracować na macierzach dowolnych rozmiarów. Jeśli zachodzi taka potrzeba, program przed wykonaniem obliczeń skaluje macierze do wymiarów wymaganych przez wybrany algorytm¹

Program rozwijany jest w serwisie github pod adresem <http://github.com/rszczers/ParallelMultiplication>.

A.1.2 Użytkowanie

Na potrzeby pracy proces testowania został całkowicie zautomatyzowany ze względu przeprowadzenia dużej liczby testów dla różnych grup danych wejściowych i na potrzeby graficznej prezentacji wydajności obliczeń dla różnej ilości procesów i wątków. Całość zarządzania jest zestawem domyślnie zdefiniowanych celów zdefiniowanych w pliku `Makefile` i realizowanych przez narzędzie `Make`. Najważniejsze z nich to:

<code>make</code>	Odbudowuje drzewo katalogów i kompiluje programy do katalogu <code>./build/</code> .
<code>make data</code>	Generuje przykładowe macierze kwadratowe <code>\$(PATH_A)</code> i <code>\$(PATH_B)</code> o szerokości <code>\$(SIZE)</code> .
<code>make seq</code>	Wykonanie mnożenia algorytmem sekwencyjnym.
<code>make omp</code>	Wykonanie mnożenia z operacją <code>parfor</code> .
<code>make mkl</code>	Wykonanie mnożenia z operacją <code>cblas_dgemm</code> .
<code>make cannon</code>	Wykonanie algorytm Cannona.
<code>make cannon_omp</code>	Wykonanie algorytm Cannona z operacją <code>parfor</code> .
<code>make cannon_dgemm</code>	wykonuje algorytm Cannona z operacją MKL <code>cblas_dgemm</code>

Dla obliczeń równoległych istotne są dwie zmienne: `NPROCS` oraz `OMP_THREADS`. Pierwsza z nich określa liczbę procesów do uruchomienia, druga — liczbę wątków. Tabela A.1 zawiera zestawienie najistotniejszych zdefiniowanych w pliku `Makefile` parametrów wywołania instrukcji `make`.

Przy domyślnych ustawieniach po wykonaniu każdego zadania w katalogu `./debug/` tworzony jest plik `debug_X`, gdzie `X` to data wykonania zadania w

¹O ile $n > \sqrt{p}$, gdzie n to rozmiar problemu, a p – liczba procesów.

czasie uniksowym. Zawiera on dane o czasie wykonania programu, jego części sekwencyjnej, rozmiarze danych, liczbie procesów i wątków. Opcja `-s` pozwala na zapis danych częściowych każdego z procesów na każdym etapie wykonywania algorytmu. Pliki zapisywane są w czytelnej postaci do katalogu `./debug/` pod nazwą `XXXX_YYYY_Z`, gdzie `XXXX` identyfikuje proces z którego zapisano dane, `YYYY` — krok algorytmu, `Z` — macierz A , B lub C , której fragment zapisano. Pozwala to łatwo prześledzić wszystkie etapy obliczeń.

Tabela A.1: Krótkie zestawienie zmiennych w pliku `Makefile`

<code>SIZE</code>	Rozmiar macierzy testowej macierzy kwadratowej.
<code>NPROC</code>	Liczba procesów do uruchomienia.
<code>DEBUG_DIR</code>	Ścieżka do katalogu z danymi do testowania algorytmów.
<code>RESOURCES_DIR</code>	Ścieżka do katalogu z danymi wejściowymi.
<code>BUILD_DIR</code>	Ścieżka do katalogu wyjściowego kompilacji.
<code>OMP_THREADS</code>	Liczba wątków do uruchomienia.
<code>OUTPUT_SRUN</code>	Ścieżka do pliku wyjściowego zawierającego macierz C dla algorytmu sekwencyjnego.
<code>OUTPUT_CRUN</code>	Ścieżka do pliku wyjściowego zawierającego macierz C dla algorytmu Cannona.
<code>OUTPUT_MRUN</code>	Ścieżka do pliku wyjściowego zawierającego macierz C dla procedury MKL <code>cblas_dgemm</code> .
<code>OUTPUT_OMP</code>	Ścieżka do pliku wyjściowego zawierającego macierz C dla algorytmu z procedurą <code>parfor</code> .
<code>OUTPUT_CANNON_OMP</code>	Ścieżka do pliku wyjściowego zawierającego macierz C dla algorytmu Cannona z procedurą <code>parfor</code> .
<code>OUTPUT_CANNON_MKL</code>	Ścieżka do pliku wyjściowego zawierającego macierz C dla algorytmu Cannona z metodą <code>cblas_dgemm</code> .
<code>PATH_A</code>	Ścieżka do pliku wejściowego zawierającego macierz A w zapisie wierszowym.
<code>PATH_B</code>	Ścieżka do pliku wejściowego zawierającego macierz B w zapisie wierszowym.
<code>MIN</code>	Kres dolny elementów macierzy do wygenerowania.
<code>MAX</code>	Kres górny elementów macierzy do wygenerowania.

Listing 1 Ekran pomocy programu pmm.

```
rszczers@solaris:~/ParallelMultiplication/build$ mpirun -np 1 ./pmm --help
Usage: pmm [OPTION...] -A matrixA_path -B matrixB_path -m NUM -k NUM -n NUM
Parallel Matrix Multiplication.
```

-a, --method=METHOD	Algorithm used
-A, --inputA=FILE	Path to input FILE containing matrix A data
-B, --inputB=FILE	Path to input FILE containing matrix B data
-d, --debug[=DIR]	Path to debug directory
-k NUM	Number of rows of B
-l, --list	Show list of available algorithms
-m NUM	Number of rows of A
-n NUM	Number of columns of B
-o, -C, --output[=FILE]	Path to output FILE containing matrix C=A*B data
-q, --quiet	Do not show any computations
-s, --steps	Dump data from each node for every step
-t, --time	Show elapsed time
-v, --verbose	Show all computations
-, --help	Give this help list
--usage	Give a short usage message
-V, --version	Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

Report bugs to <rafal.szczerski@gmail.com>.

Listing 2 Ekran pomocy programu gen.

```
rszczers@solaris:~/ParallelMultiplication/build$ ./gen --help
Usage: gen [OPTION...] -l NUM
```

-f, --float	Generate floating point numbers
-l, --length=NUM	Length of array to generate
-m, --min=NUM	Lower boundary of generated elements
-M, --max=NUM	Upper boundary of generated elements
-p, --path[=FILE]	Output file
-v, --verbose	Verbose mode
-, --help	Give this help list
--usage	Give a short usage message

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

A.1.3 Przebieg testów na klastrze Solaris

Wyniki przedstawione w pracy można odtworzyć za pomocą zadań obliczeniowych systemu kolejkowego Torque zdefiniowanych w plikach `job.sh` i `job_seq.sh` (patrz listing 3, listing 4). Poniżej pokażemy przykładową sesję pracy z programem. Zaczniemy od procesu kompilacji i skończymy na generowaniu wykresów.

Aktualną wersję programu pobieramy z repozytorium i kompilujemy źródła:

```
$ git clone git@github.com:rszczers/ParallelMultiplication.git
$ make all
```

W katalogu `./build/` powstały pliki `pmm` i `gen`. Pierwszy z nich implementuje wybrane algorytmy i przeprowadza ich testy, drugi służy do generowania przykładowych macierzy na potrzeby testów.

Aby dodać zdefiniowane zadania obliczeniowe do systemu kolejkowania wykonujemy komendy:

```
$ qsub job.sh
$ qsub job_seq.sh
```

Aby ocenić status wykonania naszego zadania używamy komendy `qstat`. Szacowany czas wykonania możemy odczytać dzięki wywołaniu komendy `showstart id`, gdzie `id` to identyfikator zadania nadany przez system kolejkowy.

W katalogu `./debug/` po wykonaniu znajdują się wszystkie mierzone parametry wykonania algorytmów dla każdego wywołania zdefiniowanego w zadaniu. Przeanalizowane dane na potrzeby wykresów umieszczane są w katalogu `./gnuplot/data/`, gdzie ulegają dalszej obróbce. Wynik końcowy w postaci wykresów umieszczany jest w katalogu `./paper/includes/plots/`, skąd są importowane do niniejszej pracy.

Listing 3 Plik `job.sh`

```
#!/bin/bash
#PBS -N pmm
#PBS -l nodes=32:ppn=1
#PBS -l walltime=72:00:00
#PBS -q default
#PBS -m abe
#PBS -M rafal.szczerski@gmail.com
#PBS -o pmm.log

cd $PBS_O_WORKDIR
for s in 2048 4096 8192; do
    make data SIZE=$s
    echo "make data SIZE=$s"
    make test SIZE=$s
done
```

Listing 4 Plik `job_seq.sh`

```
#!/bin/bash
#PBS -N pmm_seqpart
#PBS -l nodes=8
#PBS -l walltime=04:00:00
#PBS -q default
#PBS -m abe
#PBS -M rafal.szczerski@gmail.com
#PBS -o pmm.log

cd $PBS_O_WORKDIR
make data SIZE=4096
echo "make data SIZE=4096"
./makeplots_seq.sh 384 4096
mv ./debug/debug_* ./debug/seq/
```

A.2 Omówienie kodu źródłowego

Rozwiązania dostarczone z niniejszą pracą stanowią konglomerat kilku technologii programistycznych, których celem jest dostarczenie narzędzia służącego przeprowadzaniu równoległych operacji szybkiego mnożenia macierzy dla dużych danych wejściowych oraz prezentacji wydajności takich operacji w dokumentach L^AT_EX (rozdział 3).

Program pmm

Trzon projektu stanowi program `pmm`. Dostarcza on wygodny interfejs użytkownika pozwalający na przeprowadzenie operacji mnożenia macierzy na dowolnych dwóch plikach reprezentujących macierz w zapisie wierszowym. W zależności od wyboru metody mnożenia operacje przebiegają sekwencyjne lub równoległe.

Interesująca z punktu widzenia obliczeń rozproszonych jest implementacja algorytmów Cannona². Z racji, że ich implementacja jest analogiczna (z dokładnością do obliczeń drobnoziarnistych i rezerowania większej ilości pamięci na ich cele) omówimy tylko implementację klasycznego algorytmu. Ich działanie zasadza się na wyborze jednego wyróżnionego węzła nadzorcy (dalej nazywanego po prostu *nadzorcą*), którego zadaniem jest inicjalizacja obliczeń oraz zebranie wyników z wszystkich węzłów sieci po ich zakończeniu. Proces inicjalizowania polega na wczytaniu do pamięci lokalnej węzła danych wejściowych³ (plików zawierających elementy macierzy).

Listing 5 Plik `main.c`; definiowanie bloków macierzy

```
365         max = arguments.m;
366         (max < arguments.k) && (max = arguments.k);
367         (max < arguments.n) && (max = arguments.n);
368
369         int sz = max/dims[0]; // row length per block
370         if(max > sz * dims[0]) {
371             sz += 1;
372             max = sz * dims[0];
373         }
374         int blockSz = sz * sz;
375
376         MPI_Type_contiguous(blockSz, MPI_DOUBLE, &MPI_SUBMATRIX);
377         MPI_Type_commit(&MPI_SUBMATRIX);
```

Na listingu 5 zaprezentowany jest fragment kodu odpowiedzialny za wyznaczanie rozmiaru bloków macierzy wejściowych do rozesłania do innych węzłów sieci. W liniach (365 – 367) określany jest największy spośród wymiarów

²Odnosząc się do listingu 1 mowa o algorytmach: `cannon`, `cannon_dgemm`, `cannon_omp`

³Rozwiązanie to ma swoje wady i swoje zalety. Przede wszystkim wymaga wiele pamięci lokalnej w węźle. Rozwiązaniem alternatywnym byłoby mapowanie plików (`mmap`) i rozsyłanie ich po fragmencie do innych węzłów bezpośrednio po zbuforowaniu. Wadą tego rozwiązania jest utrudnione wstępne przetwarzanie (skalowanie macierzy, interpretacja szerokości i długości) przy zachowaniu wierszowego zapisu macierzy.

macierzy podanych przez użytkownika⁴. W liniijkach (369 – 374) określana jest szerokość bloku macierzy (zmienna `sz`), liczba elementów w bloku (zmienna `blockSz`) oraz szerokość całej macierzy po uwzględnieniu skalowania (zmienna `max`).

Nadzorca decyduje o skalowaniu macierzy w zależności od wymiarów torusa⁵, na którym przeprowadzane są obliczenia. Zadanie węzła nadzorującego kończy się na tym etapie zdefiniowaniem typu `MPI_SUBMATRIX`, który w dalszej części będzie służył do wysyłania komunikatów w sieci zawierających odpowiednie podmacierze.

Wszystkie argumenty dostarczone przez użytkownika przechowywane są w strukturze `arguments`. Wspomniane skalowanie macierzy wejściowych ma miejsce wówczas, gdy wymiary macierzy dostarczone przez użytkownika nie są równe (macierze nie są kwadratowe) oraz ich szerokość nie jest wielokrotnością szerokości sieci w topologii torusa. Decydują o tym instrukcje zamieszczone w listingu 6.

Listing 6 Plik `main.c`; warunek skalowania macierzy

```
408         bool resize = true;
409         if(arguments.m == max &&
410            arguments.n == max &&
411            arguments.k == max) {
412             resize = false;
413         }
414
415         load_matrix(arguments.pathA, A, arguments.m, arguments.k,
416                   max, resize);
417         load_matrix(arguments.pathB, B, arguments.k, arguments.n,
418                   max, resize);
```

Skalowanie odbywa się w trakcie wczytywania macierzy przez nadzorcę w procedurze `load_matrix`. Decyduje o tym ostatni ostatni z argumentów formalnych. Procedura polega na uzupełnieniu brakujących elementów zerami, aż do uzyskania macierzy o wymiarach wymaganych przez algorytm Cannona.

Na tym etapie rozpoczynamy właściwą część algorytmu polegającą na rozdstrybuowaniu macierzy blokowej do węzłów sieci. Na potrzebę objaśnienia dalszej części implementacji powiedzmy, że chcemy obliczyć iloczyn pewnych macierzy kwadratowych A , B składających się z dziewięciu bloków A_{ij} B_{ij} o

⁴Wykorzystuje się tutaj fakt, że operator przypisania „=” zwraca wartość przypisania.

⁵Szerokość i wysokość torusa muszą być takie same; w przeciwnym wypadku obliczenie jest przerywane.

równych wymiarach. Klasyczna wersja algorytmu omawiana w [15], [9], [1] zakłada wstępnie rozdystrybuowanie podmacierzy w sposób przedstawiony na rysunku A.1 i następnie przeprowadzenie odpowiednich przesunięć podmacierzy między węzłami sieci do stanu przedstawionego na rysunku A.2.

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

Rysunek A.1: Klasyczne wejściowe rozmieszczenie macierzy blokowych A_{ij} i B_{ij}

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

Rysunek A.2: Proponowane wejściowe rozmieszczenie macierzy blokowych A_{ij} i B_{ij}

Operację tę można uprościć przechodząc bezpośrednio do rozmieszczenia podmacierzy w sieci takiego jak na rys. A.2. Rozpatrzmy kod zamieszczony na listingu 7. Zawarte w nim instrukcje wykonywane są tylko przez nadzorcę i dotyczą wyłącznie dystrybucji macierzy A .

Aby zrozumieć ideę stojącą za tą częścią kodu sprowadźmy dla przykładu problem do obliczeń na torusie 3×3 i macierzy kwadratowej A złożonych z dziewięciu bloków A_{ij} równego wymiaru. Jeśli tablicę `proclA` zinterpretujemy jako macierz 3×3 w zapisie wierszowym, wówczas obrazowo przedstawia ją rysunek A.3.

0	1	2
5	3	4
7	8	6

Rysunek A.3: Tablica pomocnicza `proclA`

Listing 7 Plik main.c; wstępne rozmieszczanie macierzy

```
443     int proclA[numprocs];
444     for (int i = 0; i < dims[0]; i++) {
445         for (int j = 0; j < dims[1]; j++) {
446             if (j < dims[1] - i) {
447                 proclA[i * dims[1] + j + i] = i * dims[1] + j;
448             } else {
449                 proclA[i * dims[1] + j - (dims[1] - i)] =
450                     i * dims[1] + j;
451             }
452         }
453     }
454
455     seq_t1 = MPI_Wtime();
456     seq_elap += seq_t1 - seq_t0;
457
458     for (int proc = numprocs - 1; proc >= 0; proc--) {
459         seq_t0 = MPI_Wtime();
460
461         int displacements[sz];
462         int start = (proc % dims[1]) * sz +
463             (proc / dims[1]) * (dims[1] * blockSz);
464         displacements[0] = start;
465         for (int k = 1; k < sz; k++) {
466             displacements[k] = displacements[k-1] +
467                 sz * dims[1];
468         }
469
470         int k = 0;
471         for (int i = 0; i < sz; i++) {
472             for (int j = 0; j < sz; j++) {
473                 pA[k] = A[displacements[i] + j];
474                 k++;
475             }
476         }
477
478         seq_t1 = MPI_Wtime();
479         seq_elap += seq_t1 - seq_t0;
480
481         if(proc != ROOT) {
482             MPI_Send(pA, 1, MPI_SUBMATRIX, proclA[proc],
483                 DISTRIBUTION_A, cartcom);
484         }
485     }
```

Matematycznie rzecz ujmując zdefiniowaliśmy w ten sposób permutację między indeksami tabeli, a zbiorem $\{0, 1, \dots, 8\}$:

$$f: \begin{cases} 0 \mapsto 0, & 1 \mapsto 1, & 2 \mapsto 2, \\ 3 \mapsto 5, & 4 \mapsto 3, & 5 \mapsto 4, \\ 6 \mapsto 7, & 7 \mapsto 8, & 8 \mapsto 6. \end{cases}$$

inaczej $f = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 1 & 2 & 5 & 3 & 4 & 7 & 8 & 6 \end{pmatrix}$.

Aby zrozumieć pętlę **for** w liniach (458–485) ponumerujmy podmacierze A_{ij} :

$$g: \begin{cases} A_{11} \mapsto 0, & A_{12} \mapsto 1, & A_{13} \mapsto 2, \\ A_{21} \mapsto 3, & A_{22} \mapsto 4, & A_{23} \mapsto 5, \\ A_{31} \mapsto 6, & A_{32} \mapsto 7, & A_{33} \mapsto 8. \end{cases}$$

Permutacja f jest nieprzypadkowa, gdyż jeśli rozważymy teraz wartość $f(g(A_{ij}))$, otrzymamy identyfikator procesu, do którego należy wysłać macierz A_{ij} . Innymi słowy odwzorowanie f jest odpowiednością między częściami macierzy (wartościami funkcji g), a identyfikatorami procesów.

Pętla **for** zawarta w liniach (458-485) iteruje się po zmiennej **proc**, która odpowiada wyżej zdefiniowanym wartościom funkcji g . Należy rozumieć przez to, że dla **proc** = i przetwarzamy do wysłania blok $A_{ij} = g^{-1}(i)$.

Tablica **displacements** definiowana w ciele pętli (linie 462-468) zawiera indeksy pierwszych elementów każdego wiersza podmacierzy tablicy **A**⁶, które z kolei służą do przepisywania odpowiednich elementów do bufora **pA** (linie 471-476). Ostatecznie bufor **pA** zawierający blok $g^{-1}(\text{proc})$ jest wysyłany do węzła $f(g(A_{ij}))$ (procedura **MPI_Send**, linia 482). Pętla jest skonstruowana tak, że po jej wykonaniu nadzorca posiada przypadający mu blok macierzy bez zbędnego wysyłania go do siebie (warunek w lini 481).

Równolegle z przygotowywaniem komunikatów przez nadzorcę, pozostałe procesory przygotowują się do odebrania przeznaczonych im komunikatów zawierających odpowiednie bloki A_{ij} . Tym sposobem otrzymujemy rozkład macierzy A przedstawiony na rysunku A.2.

Aby zapobiec blokowaniu się komunikacji, na tym etapie procesy są synchronizowane (procedura **MPI_Barrier**) i w analogiczny sposób⁷ dystrybuowana jest macierz B .

Kolejny etap obliczeń polega na przesyłaniu sobie przez procesy swoich bloków A_{ij} i wykonywaniu obliczeń iloczynów częściowych w myśl algorytmu

⁶Tablica **A** przechowuje wszystkie elementy macierzy A w węźle nadzorczym.

⁷Z dokładnością do permutacji

omówionego w części 2.3. Procesowi temu odpowiadają instrukcje pokazane na listingu 8

Listing 8 Plik `main.c`; przesyłanie podmacierzy

```
576     int top, bottom, left, right;
577     MPI_Cart_shift(cartcom, 1, 1, &left, &right);
578     MPI_Cart_shift(cartcom, 0, 1, &top, &bottom);
579
580     for (int i = 1; i < dims[0]; i++) { /* if dims[0] == dims[1] */
581         MPI_Sendrecv_replace(pA, 1, MPI_SUBMATRIX, left,
582                               SKEW_LEFTRIGHT, right,
583                               SKEW_LEFTRIGHT, cartcom, &status);
584         MPI_Sendrecv_replace(pB, 1, MPI_SUBMATRIX, top,
585                               SKEW_BOTTOMUP, bottom,
586                               SKEW_BOTTOMUP, cartcom, &status);
587
588         for (int ii = 0; ii < sz; ii++) {
589             for (int j = 0; j < sz; j++) {
590                 for (int l = 0; l < sz; l++) {
591                     pC[ii * sz + j] +=
592                         pA[ii * sz + l] * pB[l * sz + j];
593                 }
594             }
595         }
```

W liniach (576-578) każdy proces równolegle znajduje identyfikatory swoich sąsiadów. W pętli zawartej między liniami (580-595) procesy wymieniają się zawartościami buforów `pA` i `pB` w komunikacji nieblokującej⁸ i przeprowadzają obliczenia częściowe⁹ (linie 588-595). Po wykonaniu odpowiedniej ilości iteracji wyniki częściowe znajdujące się w pamięciach lokalnych każdego z węzłów są zbierane do nadzorcy. Proces ten pokazany jest na listingu 9.

⁸Procedura `MPI_Sendrecv_replace`.

⁹Algorytmy hybrydowe w tej części korzystają z równoległości drobnoziarnistej używając innych metod obliczania iloczynów częściowych

Listing 9 Plik main.c; odbieranie wyników częściowych

```
644     for(int proc = 1; proc < numprocs; proc++) {
645         if(pid == proc) {
646             MPI_Send(pC, 1, MPI_SUBMATRIX, ROOT,
647                     COLLECTING, cartcom);
648         }
649
650         if (pid == ROOT) {
651             MPI_Recv(pC, 1, MPI_SUBMATRIX, proc,
652                     COLLECTING, cartcom, &status);
653
654             seq_t0 = MPI_Wtime();
655
656             // przesunięcia wierszy dla pozostałych procesów
657             int displacements[sz];
658             /* proc % dims[1] * sz - first column */
659             /* of first line of submatrix stored in p-process */
660             /* (proc / dims[1])... - horizontal shifts */
661             int start = (proc % dims[1]) * sz +
662                 (proc / dims[1]) * (dims[1] * blockSz);
663             displacements[0] = start;
664
665             for(int k = 1; k < sz; k++) {
666                 displacements[k] =
667                     displacements[k-1] + sz * dims[1];
668             }
669
670             for(int i = 0; i < sz; i++) {
671                 for(int j = 0; j < sz; j++) {
672                     C[displacements[i] + j] = pC[i * sz + j];
673                 }
674             }
675
676             seq_t1 = MPI_Wtime();
677             seq_elap += seq_t1 - seq_t0;
678         }
679     }
```

W liniach (657-668) obliczane są indeksy pierwszych elementów każdego wiersza bloku wynikowego macierzy C odbieranego kolejno od każdego z procesorów. Bloki są niejako „wsadzane” na swoje miejsce w macierzy C w pętli między liniami (670-674). Macierz wynikowa C znajduje się w pamięci nadzorcy. Wszystkie bufory pomocnicze są zwalniane, a macierz wynikowa ewentual-

nie zapisywana do pliku¹⁰ lub drukowana na standardowym wyjściu.¹¹

Dane o wydajności algorytmów

W kodzie wybrano szereg punktów pomiarowych, w których z pomocą procedury `MPI_Wtime` mierzy się całkowity czas wykonania obliczeń (włączając w to czas wczytywania plików do węzła nadzorującego i czas na komunikację między węzłami), czas wykonywania samego algorytmu do chwili otrzymania wyników częściowych na każdym z węzłów oraz czas wykonania części sekwencyjnych. Tak uzyskane dane mogą zostać przez użytkownika zapisane w dowolnym wybranym miejscu celem przeprowadzenia późniejszej analizy¹². Przykładowy plik wynikowy został zamieszczony w listingu 10. Dane umieszczone w pliku kolejno oznaczają: nazwę metody obliczeń, czas wykonania algorytmu równoległego, rozmiary m , k , n podane przez użytkownika, liczbę procesów, liczbę wątków, czas wykonania części sekwencyjnej, czas całkowity obliczenia.

Listing 10 Przykładowy plik z danymi pomiarowymi

```
METHOD:      CANNON
ETA:          1.012851
ARGS:         m=2048, k=2048, n=2048, nprocs=64, threads=1,
↪ seq_elap=0.023992, total=4.427899
```

Za zapisywanie danych pomiarowych odpowiada procedura `save_info`.

Przetwarzanie danych pomiarowych

Przetwarzanie danych pomiarowych polega na wyciąganiu sekwencji interesujących danych potrzebnych później w procesie generowania wykresów programem `gnuplot`. Dane programu `gnuplot` są w formacie kolumn danych oddzielonych tabulacją. Przetwarzaniem danych wynikowych programu `pmm` do postaci danych `gnuplot` zajmuje się skrypt `data.sh`. Jest on w istocie ciągiem instrukcji sortujących i selekcjonujących dane (listing 11) otrzymane dzięki skryptowi `parse.pl`. Skrypt ten wyciąga wszystkie ciągi liczbowe z danych pomiarowych i układa je w linię danych oddzielonych separatorem (listing 12).

¹⁰Tylko o ile użytkownik określił parametr opcji `-C`, którym powinna być ścieżka do pliku wyjściowego.

¹¹Odpowiada za to opcja `-v`.

¹²Odpowiada za to opcja `-d`.

Listing 11 Fragment skryptu data.sh

```
4 ./parse.pl cannon_omp ./debug/seq | sort -t$'\t' -k5 -n >
  ↪ gnuplot/data/scal_cannon_omp.dat
5 ./parse.pl cannon_dgemm ./debug/seq | sort -t$'\t' -k5 -n >
  ↪ gnuplot/data/scal_cannon_dgemm.dat
6
7 cat gnuplot/data/scal_cannon_dgemm.dat | awk '$6==1' >
  ↪ gnuplot/data/scal_cannon_dgemm_1.dat
8 cat gnuplot/data/scal_cannon_dgemm.dat | awk '$6==2' >
  ↪ gnuplot/data/scal_cannon_dgemm_2.dat
9 cat gnuplot/data/scal_cannon_dgemm.dat | awk '$6==4' >
  ↪ gnuplot/data/scal_cannon_dgemm_4.dat
```

Listing 12 Skrypt parse.pl

```
#!/usr/bin/perl
use strict;
use warnings;
use English;
use feature "switch";
$ARGV[0] = "CANNON" unless defined $ARGV[0];
my $method = shift;

my $dir = shift;
#my $dir = './debug';

foreach my $fp (glob("$dir/debug_*")) {
    open (my $fh, "<", $fp) or die "cannot open '$fp': $OS_ERROR";
    if(<$fh> =~ /\s$method$/gim) {
        my @data;
        while (<$fh>) {
            my @tmp = /(\d+\.\d+)|(\d+)/g;
            foreach(@tmp) {
                if((defined $_) and !($_ =~ /\$/)){
                    push @data, $_;
                }
            }
        }
        print join("\t", @data);
        print "\n";
    }
    close $fh or die "cannot close '$fp': $OS_ERROR";
}
```

Generowanie wykresów

Za generowanie wykresów odpowiadają pliki `*.p` w ramach których zdefiniowane są ciągi instrukcji programu `gnuplot`. Generuje on w katalogu `./gnuplot/` dane w formacie `.eps` zawierające dane graficzne oraz `.tex` zawierające wszystkie napisy pojawiające się na wykresie. Wywołaniu skryptu `data.sh` towarzyszy uruchomienie każdego ze zdefiniowanych skryptów oraz przeniesienie plików `.eps` i `.tex` do katalogu `./paper/includes/plots/`.

Program gen

Jako efekt uboczny pracy nad programem `pmm` powstał program do generowania długich ciągów liczb pseudolosowych za pomocą funkcji `rand`. W początkowych etapach nad pracą wykorzystywany był prosty skrypt `bash` generujący ciągi liczb w pętli i strumieniujący je do pliku z każdym powtórzeniem refrenu pętli. Rozwiązanie to okazało się wolne i praktycznie uniemożliwiało generowanie macierzy o wymiarach większych niż 256×256 . W ten sposób pojawiła się konieczność zaimplementowania nowego rozwiązania. Jest nim program `gen` z zaimplementowanym zapisem buforowanym i szeregiem opcji (listing 2) ułatwiających uzyskanie macierzy o zadanych własnościach.

Dodatek B

Załącznik

Płyta CD zawiera:

- Pliki źródłowe pracy licencjackiej, w tym dokumenty systemu L^AT_EX oraz pliki graficzne programu Geogebra (katalog `./paper/`),
- Skompilowana wersja pracy licencjackiej w formacie pdf (plik `./paper/parallel_matrix_multiplication.pdf`; kompilację można łatwo powtórzyć poleceniem `make`),
- Kod źródłowy programów użytych do pomiarów wydajności wybranych algorytmów (katalog `./src/`),
- Skrypty perl i bash automatyzujące testowanie algorytmów, analizę danych i kolejkovanie zadań w systemie Torque (pliki `parse.pl`, `data.sh`, `makeplots.sh`, `makeplots_seq.sh`, `job.sh`, `job_seq.sh` w katalogu głównym),
- Skrypty programu Gnuplot służące do generowania wykresów zamieszczonych w pracy (pliki `./gnuplot/*.p`),
- Dane pomiarowe wygenerowane na potrzeby niniejszej pracy (katalog `./debug/` oraz `./gnutplot/data/`).

Bibliografia

- [1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.
- [2] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
- [3] Shmuel Winograd. On multiplication of 2×2 matrices. *Linear Algebra and its Applications*, 4:381–388, 1971.
- [4] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.
- [5] 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.
- [6] Ching-Tien Ho, S.L. Johnsson, and A. Edelman. Matrix Multiplication on Hypercubes Using Full Bandwidth and Constant Storage. In *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, pages 447–451. IEEE, 1991.
- [7] Ed J. Radatz. *Standards Coordinating Committee 10, Terms and Definitions. The IEEE Standard Dictionary of Electrical and Electronics Terms*. IEEE, 1996.
- [8] Fayez Gebali. *Algorithms and Parallel Computing*. Wiley Publishing, 1st edition, 2011.
- [9] Zbigniew Czech. *Wprowadzenie do obliczeń równoległych*. Wydawnictwo Naukowe PWN, Warszawa, 2013.
- [10] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

- [11] Cormen Thomas H., Leiserson Charles E., Rivest Roland L., and Stein Clifford. *Wprowadzenie do algorytmów*. WNT, 2001.
- [12] Matrix indexing in MATLAB. <http://www.mathworks.com/company/newsletters/articles/matrix-indexing-in-matlab.html>. Dostęp: 08/09/15.
- [13] 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.
- [14] Craig C. Douglas, Michael Heroux, Gordon Slishman, Roger M. Smith, and Roger M. Gemm: A portable level 3 blas winograd variant of strassen's matrix-matrix multiply algorithm, 1994.
- [15] Przemysław Stpicznyński and Marcin Brzuszek. *Podstawy programowania obliczeń równoległych*. UMCS, 2011.