Podstawy programowania obliczeń równoległych

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UMCS

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Podstawy programowania obliczeń równoległych

Przemysław Stpiczyński
Marcin Brzuszek

UMCS
UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ

Lublin 2011

Instytut Informatyki UMCS
Lublin 2011

Przemysław Stpiczyński (Instytut Matematyki UMCS)
Marcin Brzuszek

Podstawy programowania obliczeń równoległych

Recenzent: Marcin Paprzycki

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
ul. Ratajczaka 26/8
61-815 Poznań
www: www.esus.pl

ISBN: 978-83-62773-15-2

Spis treści

Przedmowa

Konstrukcja komputerów oraz klastrów komputerowych o dużej mocy obliczeniowej wiąże się z istnieniem problemów obliczeniowych, które wymagają rozwiązania w akceptowalnym czasie. Pojawianie się kolejnych typów architektur wieloprocesorowych oraz procesorów zawierających mechanizmy wewnętrznej równoległości stanowi wyzwanie dla twórców oprogramowania. Zwykle kompilatory optymalizujące nie są w stanie wygenerować kodu maszynowego, który w zadowalającym stopniu wykorzystywałby teoretyczną maksymalną wydajność skomplikowanych architektur wieloprocesorowych. Stąd potrzeba ciągłego doskonalenia metod obliczeniowych, które mogłyby być efektywnie implementowane na współczesnych architekturach wieloprocesorowych i możliwie dobrze wykorzystywać moc oferowaną przez konkretne maszyny. Trzeba tutaj podkreślić, że w ostatnich latach nastąpiło upowszechnienie architektur wieloprocesorowych za sprawą procesorów wielordzeniowych, a zatem konstrukcja algorytmów równoległych stała się jednym z ważniejszych kierunków badań nad nowymi algorytmami. Pojawiają się nawet głosy, że powinno się utożsamiać programowanie komputerów z programowaniem równoległym[1].

Niniejsza książka powstała na bazie wcześniejszych publikacji autorów, w szczególności prac [54, 60, 64] oraz przygotowywanych materiałów do zajęć z przedmiotu *Programowanie równoległe*. Stanowi ona wprowadzenie do programowania obliczeń (głównie numerycznych) na komputerach równoległych z procesorami ogólnego przeznaczenia (CPU). Nie omawia ona zagadnień związanych z programowaniem z wykorzystaniem procesorów kart graficznych, gdyż będzie to tematem odrębnej publikacji. Zawiera ona przegląd współczesnych komputerowych architektur wieloprocesorowych, omawia metody teoretycznej analizy wydajności komputerów oraz prezentuje szczegółowo programowanie z wykorzystaniem standardów OpenMP i MPI.

[1] Justin R. Rattner, wiceprezes firmy Intel, dyrektor *Corporate Technology Group* oraz *Intel Chief Technology Officer*, `http://www.computerworld.pl/news/134247_1.html`

Poruszone jest również zagadnienie programowania komputerów równoległych przy pomocy bibliotek podprogramów realizujących ważne algorytmy numeryczne.

Książka stanowi podręcznik do przedmiotu *Programowanie równoległe* prowadzonego dla studentów kierunków *matematyka* oraz *informatyka* na Wydziale Matematyki, Fizyki i Informatyki Uniwersytetu Marii Curie-Skłodowskiej w Lublinie, choć może być ona również przydatna studentom innych kierunków studiów oraz wszystkim zainteresowanym tematyką programowania komputerów wieloprocesorowych. Materiał wprowadzany na wykładzie odpowiada poszczególnym rozdziałom podręcznika. Każdy rozdział kończą zadania do samodzielnego zaprogramowania w ramach laboratorium oraz prac domowych.

Rozdział 1

Przegląd architektur komputerów równoległych

W pierwszym rozdziale przedstawimy krótki przegląd zagadnień związanych ze współczesnymi równoległymi architekturami komputerowymi wykorzystywanymi do obliczeń naukowych oraz omówimy najważniejsze problemy związane z dostosowaniem kodu źródłowego programów w celu efektywnego wykorzystania możliwości oferowanych przez współczesne komputery wektorowe, równoległe oraz klastry komputerowe. Więcej informacji dotyczących omawianych zagadnień można znaleźć w książkach [21,25,38,40,55].

1.1. Równoległość wewnątrz procesora i obliczenia wektorowe

Jednym z podstawowych mechanizmów stosowanych przy konstrukcji szybkich procesorów jest *potokowość*. Opiera się on na prostym spostrzeżeniu. W klasycznym modelu von Neumanna, procesor wykonuje kolejne rozkazy w cyklu *pobierz–wykonaj*. Każdy cykl jest realizowany w kilku etapach. Rozkaz jest pobierany z pamięci oraz dekodowany. Następnie pobierane są potrzebne argumenty rozkazu, jest on wykonywany, po czym wynik jest umieszczany w pamięci lub rejestrze. Następnie w podobny sposób przetwarzany jest kolejny rozkaz. W mechanizmie potokowości każdy taki etap jest wykonywany przez oddzielny układ (segment), który działa równolegle z pozostałymi układami, odpowiedzialnymi za realizację innych etapów. Wspólny zegar synchronizuje przekazywanie danych między poszczególnymi segmentami, dostosowując częstotliwość do czasu działania najwolniejszego segmentu [40]. Zakładając, że nie ma bezpośredniej zależności między kolejnymi rozkazami, gdy pierwszy rozkaz jest dekodowany, w tym samym czasie może być pobrany z pamięci następny rozkaz. Następnie, gdy realizowane jest pobieranie argumentów pierwszego, jednocześnie trwa dekodowanie drugiego i pobieranie kolejnego rozkazu. W ten sposób, jeśli liczba etapów wykonania pojedynczego rozkazu wynosi k oraz za jednostkę czasu przyjmiemy czas wykonania jednego etapu, wówczas potokowe wykonanie n rozkazów zajmie $n + k - 1$ zamiast $k \cdot n$, jak miałoby to miejsce w klasycznym modelu von Neumanna. Gdy istnieje bezpośrednia zależność między rozkazami (na przykład w postaci instrukcji skoku warunkowego), wówczas jest wybierana najbardziej prawdopodobna gałąź selekcji (mechanizm *branch prediction* [45]).

Idea potokowości została dalej rozszerzona w kierunku *mechanizmu wektorowości*. W obliczeniach naukowych większość działań wykonywanych jest na wektorach i macierzach. Zaprojektowano zatem specjalne potoki dla realizacji identycznych obliczeń na całych wektorach oraz zastosowano mechanizm *łańcuchowania* (ang. *chaining*) potoków, po raz pierwszy w komputerze

Cray-1. Przykładowo, gdy wykonywana jest operacja postaci

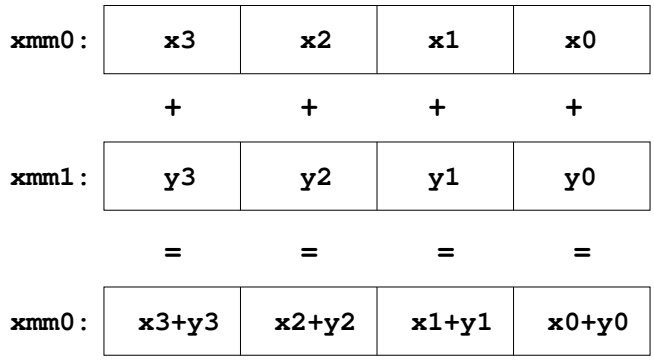
$$\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}, \tag{1.1}$$

wówczas jeden potok realizuje mnożenie wektora $\mathbf{x}$ przez liczbę α, drugi zaś dodaje wynik tego mnożenia do wektora $\mathbf{y}$, bez konieczności oczekiwania na zakończenie obliczania pośredniego wyniku $\alpha\mathbf{x}$ [15]. Co więcej, lista rozkazów procesorów zawiera rozkazy operujące na danych zapisanych w specjalnych rejestrach, zawierających pewną liczbę słów maszynowych stanowiących elementy wektorów, a wykonanie takich rozkazów odbywa się przy użyciu mechanizmów potokowości i łańcuchowania. Takie procesory określa się mianem wektorowych [15]. Zwykle są one wyposażone w pewną liczbę jednostek wektorowych oraz jednostkę skalarną realizującą obliczenia, które nie mogą być wykonane w sposób wektorowy.

Realizując idee równoległości wewnątrz pojedynczego procesora na poziomie wykonywanych równolegle rozkazów (ang. *instruction-level parallelism*) powstała koncepcja budowy procesorów superskalarnych [41, 43], wyposażonych w kilka jednostek arytmetyczno-logicznych (ALU) oraz jedną lub więcej jednostek realizujących działania zmiennopozycyjne (FPU). Jednostki obliczeniowe otrzymują w tym samym cyklu do wykonania instrukcje pochodzące zwykle z pojedynczego strumienia. Zależność między poszczególnymi instrukcjami jest sprawdzana dynamicznie w trakcie wykonania programu przez odpowiednie układy procesora. Przykładem procesora, w którym zrealizowano superskalarność, jest PowerPC 970 firmy IBM.

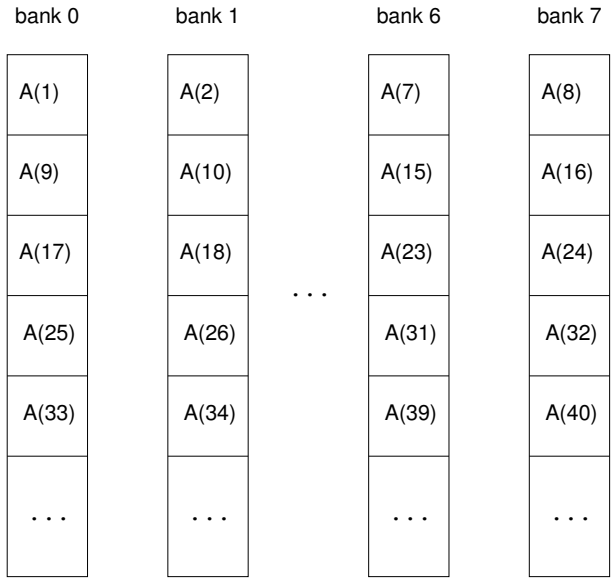
Ciekawym pomysłem łączącym ideę wektorowości z użyciem szybkich procesorów skalarnych jest architektura ViVA (ang. *Virtual Vector Architecture* [46]) opracowana przez IBM. Zakłada ona połączenie ośmiu skalarnych procesorów IBM Power5 w taki sposób, aby mogły działać jak pojedynczy procesor wektorowy o teoretycznej maksymalnej wydajności na poziomie 60-80 Gflops[1]. Architektura ViVA została wykorzystana przy budowie superkomputera ASC Purple zainstalowanego w Lawrence Livermore National Laboratory, który w listopadzie 2006 uplasował się na czwartym miejscu listy rankingowej Top500 systemów komputerowych o największej mocy obliczeniowej na świecie [11]. Warto wspomnieć, że podobną ideę zastosowano przy budowie superkomputera Cray X1, gdzie połączono cztery procesory SSP (ang. *single-streaming processor*) w procesor MSP (ang. *multi-streaming processor*).

[1] 1 Gflops (= 1000 Mflops) jest miarą wydajności komputerówi oznacza 10^9 operacji zmiennopozycyjnych na sekundę. Szczegółowo wyjaśniamy to pojęcie w rozdziale 2.



Rysunek 1.1. Dodawanie wektorów przy użyciu rozkazu `addps xmm0,xmm1`

Idea wektorowości została wykorzystana w popularnych procesorach Intela, które począwszy od modelu Pentium III zostały wyposażone w mechanizm SSE (ang. *streaming SIMD extensions* [32, 33]), umożliwiający działanie na czteroelementowych wektorach liczb zmiennopozycyjnych pojedynczej precyzji, przechowywanych w specjalnych 128-bitowych rejestrach (ang. *128-bit packed single-precision floating-point*) za pomocą pojedynczych rozkazów, co stanowi realizację koncepcji SIMD (ang. *single instruction stream, multiple data stream*) z klasyfikacji maszyn cyfrowych według Flynna [22]. Rysunek 1.1 pokazuje sposób realizacji operacji dodawania dwóch wektorów czteroelementowych za pomocą rozkazów SSE. W przypadku działania na dłuższych wektorach stosowana jest technika dzielenia wektorów na części czteroelementowe, które są przetwarzane przy użyciu rozkazów SSE. Wprowadzono również rozkazy umożliwiające wskazywanie procesorowi konieczności załadowania do pamięci podręcznej potrzebnych danych (ang. *prefetching*). Mechanizm SSE2, wprowadzony w procesorach Pentium 4 oraz procesorach Athlon 64 firmy AMD, daje możliwość operowania na wektorach liczb zmiennopozycyjnych podwójnej precyzji oraz liczb całkowitych przechowywanych również w 128-bitowych rejestrach. Dalsze rozszerzenia SSE3 i SSE4 [34, 35] wprowadzone odpowiednio w procesorach Pentium 4 Prescot oraz Core 2 Duo poszerzają zestaw operacji o arytmetykę na wektorach liczb zespolonych i nowe rozkazy do przetwarzania multimediów, wspierające przykładowo obróbkę formatów wideo. Użycie rozkazów z repertuaru SSE na ogół znacznie przyspiesza działanie programu, gdyż zmniejsza się liczba wykonywanych rozkazów w stosunku do liczby przetworzonych danych.



Rysunek 1.2. Rozmieszczenie składowych tablicy w ośmiu bankach pamięci

1.2. Wykorzystanie pamięci komputera

Kolejnym ważnym elementem architektury komputerów, który w znacznym stopniu decyduje o szybkości obliczeń, jest system pamięci, obejmujący zarówno pamięć operacyjną, zewnętrzną oraz, mającą kluczowe znaczenie dla osiągnięcia wysokiej wydajności obliczeń, pamięć podręczną.

1.2.1. Podział pamięci na banki

Aby zapewnić szybką współpracę procesora z pamięcią, jest ona zwykle dzielona na banki, których liczba jest potęgą dwójki. Po każdym odwołaniu do pamięci (odczyt lub zapis) bank pamięci musi odczekać pewną liczbę cykli zegara, zanim będzie gotowy do obsługi następnego odwołania. Jeśli dane są pobierane z pamięci w ten sposób, że kolejne ich elementy znajdują się w kolejnych bankach pamięci, wówczas pamięć jest wykorzystywana optymalnie, co oczywiście wiąże się z osiąganiem pożądanej dużej efektywności wykonania programu.

Kompilatory języków programowania zwykle organizują rozmieszczenie danych w pamięci w ten sposób (ang. *memory interleaving*), że kolejne elementy danych (najczęściej składowe tablic) są alokowane w kolejnych bankach pamięci (rysunek 1.2). Niewłaściwa organizacja przetwarzania danych

umieszczonych w pamięci w ten właśnie sposób może spowodować znaczne spowolnienie działania programu. Rozważmy przykładowo następującą konstrukcję iteracyjną.

```
1 for ( i =1; i<=n ; i+=k ) {
2     A[ i ]++;
3 }
```

Jeśli składowe tablicy `A` przetwarzane są kolejno (`K=1`), wówczas nie występuje oczekiwanie procesora na pamięć, gdyż aktualnie przetwarzane składowe znajdują się w kolejnych bankach pamięci. Jeśli zaś przykładowo `K=4`, wówczas będą przetwarzane kolejno składowe `A(1)`, `A(5)`, `A(9)`, `A(11)` itd. Zatem co druga składowa będzie się znajdować w tym samym banku. Spowoduje to konflikt w dostępie do banków pamięci, procesor będzie musiał czekać na pamięć, co w konsekwencji znacznie spowolni obliczenia. W praktyce, konflikty w dostępie do banków pamięci mogą spowodować nawet siedmiokrotny wzrost czasu obliczeń [17, 52, 53]. Należy zatem unikać sytuacji, gdy wartość zmiennej `K` będzie wielokrotnością potęgi liczby dwa.

1.2.2. Pamięć podręczna

Kolejnym elementem architektury komputera, który ma ogromny wpływ na szybkość wykonywania obliczeń, jest *pamięć podręczna* (ang. *cache memory*). Jest to na ogół niewielka rozmiarowo pamięć umieszczana między procesorem a główną pamięcią operacyjną, charakteryzująca się znacznie większą niż ona szybkością działania. W pamięci podręcznej składowane są zarówno rozkazy, jak i dane, których wykorzystanie przewidują odpowiednie mechanizmy procesora [57]. Nowoczesne systemy komputerowe mają przynajmniej dwa poziomy pamięci podręcznej. Rejestry procesora, poszczególne poziomy pamięci podręcznej, pamięć operacyjna i pamięć zewnętrzna tworzą hierarchię pamięci komputera. Ogólna zasada jest następująca: *im dalej od procesora, tym pamięć ma większą pojemność, ale jest wolniejsza.* Aby efektywnie wykorzystać hierarchię pamięci, algorytmy powinny realizować koncepcję *lokalności danych* (ang. *data locality*). Pewna porcja danych powinna być pobierana „w stronę procesora", czyli do mniejszej, ale szybszej pamięci. Następnie, gdy dane znajdują się w pamięci podręcznej najbliżej procesora, powinny być realizowane na nich wszystkie konieczne i możliwe do wykonania na danym etapie działania programu. W optymalnym przypadku algorytm nie powinien więcej odwoływać się do tych danych. Koncepcję lokalności danych najpełniej wykorzystano przy projektowaniu blokowych wersji podstawowych algorytmów algebry liniowej w projekcie ATLAS [65]. Pobieranie danych do pamięci podręcznej „bliżej procesora" jest zwykle realizowane automatycznie przez odpowiednie układy procesora,

choć lista rozkazów procesora może być wyposażona w odpowiednie rozkazy „powiadamiania" procesora o konieczności przesłania określonego obszaru pamięci w stronę procesora, jak to ma miejsce w przypadku rozszerzeń SSE [33]. Dzięki temu, gdy potrzebne dane znajdują się w pamięci podręcznej pierwszego poziomu, dany rozkaz będzie mógł być wykonany bez opóźnienia. W przypadku konieczności ładowania danych z pamięci operacyjnej oczekiwanie może trwać od kilkudziesięciu do kilkuset cykli [32, rozdział 6].

W przypadku obliczeń na macierzach rzeczą naturalną wydaje się użycie tablic dwuwymiarowych. Poszczególne składowe mogą być rozmieszczane wierszami (języki C/C++) albo kolumnami (język Fortran), jak przedstawiono na rysunku 1.3. Rozważmy przykładowo macierz

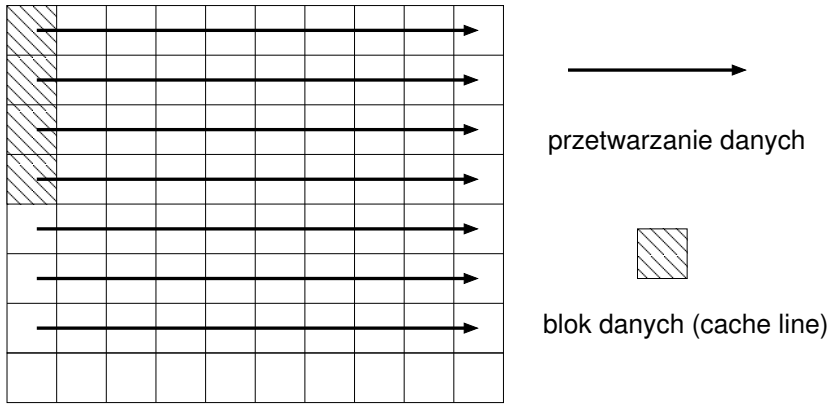
$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}. \tag{1.2}$$

Przy rozmieszczeniu elementów kolumnami istotny jest parametr `LDA` (ang. *leading dimension of array*), określający liczbę wierszy tablicy dwuwymiarowej, w której przechowywana jest macierz (1.2). Zwykle przyjmuje się `LDA` $= m$, choć w pewnych przypadkach z uwagi na możliwe lepsze wykorzystanie pamięci podręcznej, korzystniej jest zwiększyć wiodący rozmiar tablicy (ang. *leading dimension padding*), przyjmując za `LDA` liczbę nieparzystą większą niż m [39], co oczywiście wiąże się z koniecznością alokacji większej ilości pamięci dla tablic przechowujących dane programu.

W pewnych przypadkach użycie tablic dwuwymiarowych może się wiązać z występowaniem zjawiska braku potrzebnych danych w pamięci podręcznej (ang. *cache miss*). Ilustruje to rysunek 1.4. Przypuśćmy, że elementy tablicy dwuwymiarowej rozmieszczane są kolumnami (ang. *column major storage*), a w pewnym algorytmie elementy macierzy są przetwarzane wierszami. Gdy program odwołuje się do pierwszej składowej w pierwszym wierszu, wówczas do pamięci podręcznej ładowany jest blok kolejnych słów z pamięci operacyjnej (ang. *cache line*), zawierający potrzebny element. Niestety, gdy następnie program odwołuje się do drugiej składowej w tym wierszu, nie znajduje się ona w pamięci podręcznej. Gdy rozmiar bloku ładowanego do pamięci podręcznej jest mniejszy od liczby wierszy, przetwarzanie tablicy może wiązać się ze słabym wykorzystaniem pamięci podręcznej (duża liczba *cache miss*). Łatwo zauważyć, że zmiana porządku przetwarzania tablicy na kolumnowy znacznie poprawi efektywność, gdyż większość potrzebnych składowych tablicy będzie się znajdować w odpowiednim momencie w pamięci podręcznej (ang. *cache hit*). Trzeba jednak zaznaczyć, że taka zmiana porządku przetwarzania składowych tablicy (ang. *loop interchange*) nie zawsze jest możliwa.

```
      1  9 17 25 33 41 49 57 65 73
      2 10 18 26 34 42 50 58 66 74
      3 11 19 27 35 43 51 59 67 75
      4 12 20 28 36 44 52 60 68 76
A =   5 13 21 29 37 45 53 61 69 77
      6 14 22 30 38 46 54 62 70 78
      7 15 23 31 39 47 55 63 71 79
      *  *  *  *  *  *  *  *  *  *
```

Rysunek 1.3. Kolumnowe rozmieszczenie składowych tablicy dwuwymiarowej 7×10 dla `LDA=8`



Rysunek 1.4. Zjawisko *cache miss* przy rozmieszczeniu kolumnowym

1.2.3. Alternatywne sposoby reprezentacji macierzy

W celu ograniczenia opisanych w poprzednim punkcie niekorzystnych zjawisk, związanych ze stosowaniem tablic dwuwymiarowych, zaproponowano alternatywne sposoby reprezentacji macierzy [28, 29], które prowadzą do konstrukcji bardzo szybkich algorytmów [20]. Podstawową ideą jest podział macierzy na bloki według następującego schematu

$$A = \begin{pmatrix} A_{11} & \dots & A_{1n_g} \\ \vdots & & \vdots \\ A_{m_g1} & \dots & A_{m_gn_g} \end{pmatrix} \in \mathbb{R}^{m\times n}. \tag{1.3}$$

Każdy blok A_{ij} jest składowany w postaci kwadratowego bloku o rozmiarze $n_b \times n_b$, w ten sposób, aby zajmował zwarty obszar pamięci operacyjnej,

```
      1  5  9 13 | 33 37 41 45 | 65 69  *  *
      2  6 10 14 | 34 38 42 46 | 66 70  *  *
      3  7 11 15 | 35 39 43 47 | 67 71  *  *
      4  8 12 16 | 36 40 44 48 | 68 72  *  *
A =  ------------------------------------------
     17 21 25 29 | 49 53 57 61 | 73 77  *  *
     18 22 26 30 | 50 54 58 62 | 74 78  *  *
     19 23 27 31 | 51 55 59 63 | 75 79  *  *
      *  *  *  * |  *  *  *  * |  *  *  *  *
```

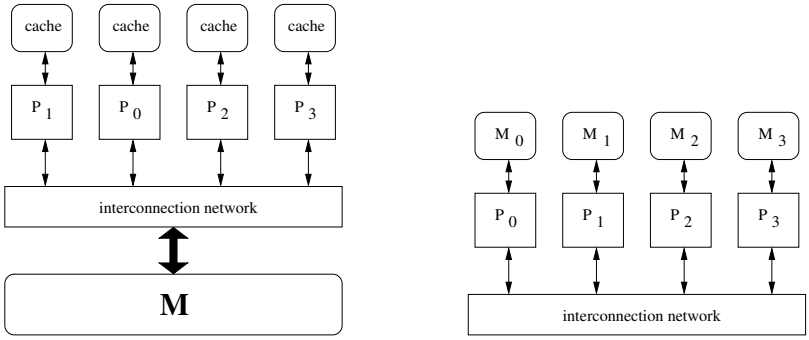
Rysunek 1.5. Nowy blokowy sposób reprezentacji macierzy

co pokazuje rysunek 1.5. Oczywiście w przypadku, gdy liczby wierszy i kolumn nie dzieli się przez n_b, wówczas dolne i prawe skrajne bloki macierzy nie są kwadratowe. Rozmiar bloku n_b powinien być tak dobrany, aby cały blok mógł zmieścić się w pamięci podręcznej pierwszego poziomu. Dzięki temu pamięć podręczna może być wykorzystana znacznie bardziej efektywnie, oczywiście pod warunkiem, że algorytm jest ukierunkowany na przetwarzanie poszczególnych bloków macierzy. Wymaga to odpowiedniej konstrukcji algorytmu, ale pozwala na bardzo dobre wykorzystanie mocy obliczeniowej procesora [28]. Postuluje się również implementację wsparcia nowych sposobów reprezentacji na poziomie kompilatora, co znacznie ułatwiłoby konstrukcję efektywnych i szybkich algorytmów [20].

W pracy [61] przedstawiono zastosowanie nowego sposobu reprezentacji macierzy dla numerycznego rozwiązywania równań różniczkowych zwyczajnych, zaś w pracy [62] podano nowy format reprezentacji macierzy wykorzystujący bloki prostokątne.

1.3. Komputery równoległe i klastry

Istnieje wiele klasyfikacji komputerów równoległych (wyposażonych w więcej niż jeden procesor). W naszych rozważaniach będziemy zajmować się maszynami pasującymi do modelu MIMD (ang. *multiple instruction stream, multiple data stream*) według klasyfikacji Flynna [22], który to model obejmuje większość współczesnych komputerów wieloprocesorowych. Z punktu widzenia programisty najistotniejszy będzie jednak dalszy podział wynikający z typu zastosowanej pamięci (rysunek 1.6). Będziemy zatem zajmować się komputerami wieloprocesorowymi wyposażonymi we wspólną pamięć (ang.



Rysunek 1.6. Komputery klasy MIMD z pamięcią wspólną i rozproszoną

shared memory), gdzie każdy procesor będzie mógł adresować dowolny fragment pamięci, oraz komputerami z pamięcią rozproszoną, które charakteryzują się brakiem realizowanej fizycznie wspólnej przestrzeni adresowej.

1.3.1. Komputery z pamięcią wspólną

W tym modelu liczba procesorów będzie na ogół niewielka[2], przy czym poszczególne procesory mogą być wektorowe. Systemy takie charakteryzują się jednolitym (ang. *uniform memory access*, UMA) i szybkim dostępem procesorów do pamięci i w konsekwencji krótkim czasem synchronizacji i komunikacji między procesorami, choć próba jednoczesnego dostępu procesorów do modułów pamięci może spowodować ograniczenie tej szybkości. Aby zminimalizować to niekorzystne zjawisko, procesory uzyskują dostęp do modułów pamięci poprzez statyczną lub dynamiczną sieć połączeń (ang. *interconnection network*). Może mieć ona postać magistrali (ang. *shared bus*) lub przełącznicy krzyżowej (ang. *crossbar switch*). Możliwa jest też konstrukcja układów logicznych przełącznicy we wnętrzu modułów pamięci (pamięć wieloportowa, ang. *multiport memory*) bądź też budowa wielostopniowych sieci połączeń (ang. *multistage networks*). Więcej informacji na ten temat można znaleźć w książce [40].

Trzeba podkreślić, że w tym modelu kluczowe dla efektywności staje się właściwe wykorzystanie pamięci podręcznej. Dzięki temu procesor, odwołując się do modułu pamięci zawierającego potrzebne dane, pobierze większą ich ilość do pamięci podręcznej, a następnie będzie mógł przetwarzać je bez konieczności odwoływania się do pamięci operacyjnej. Wiąże się to również

[2] Pod pojęciem „niewielka" w obecnej chwili należy rozumieć „rzędu kilku", a maksymalnie kilkudziesięciu.

z koniecznością zapewnienia spójności pamięci podręcznej (ang. *cache coherence*), gdyż pewne procesory mogą jednocześnie modyfikować te same obszary pamięci operacyjnej przechowywane w swoich pamięciach podręcznych, co wymaga użycia odpowiedniego protokołu uzgadniania zawartości. Zwykle jest to realizowane sprzętowo [25, podrozdział 2.4.6]. Zadanie możliwie równomiernego obciążenia procesorów pracą jest jednym z zadań systemu operacyjnego i może być realizowane poprzez mechanizmy wielowątkowości, co jest określane mianem symetrycznego wieloprzetwarzania [38] (ang. *symmetric multiprocessing* – SMP).

1.3.2. Komputery z pamięcią rozproszoną

Drugim rodzajem maszyn wieloprocesorowych będą komputery z *pamięcią fizycznie rozproszoną* (ang. *distributed memory*), charakteryzujące się brakiem realizowanej fizycznie wspólnej przestrzeni adresowej. W tym przypadku procesory będą wyposażone w system pamięci lokalnej (obejmujący również pamięć podręczną) oraz połączone ze sobą za pomocą sieci połączeń. Najbardziej powszechnymi topologiami takiej sieci są pierścień, siatka, drzewo oraz hipersześcian (ang. *n-cube*), szczególnie ważny z uwagi na możliwość zanurzenia w nim innych wykorzystywanych topologii sieci połączeń. Do tego modelu będziemy również zaliczać klastry budowane z różnych komputerów (niekoniecznie identycznych) połączonych siecią (np. Ethernet, Myrinet, InfiniBand).

Komputery wieloprocesorowe budowane obecnie zawierają często oba rodzaje pamięci. Przykładem jest Cray X1 [49] składający się z węzłów obliczeniowych zawierających cztery procesory MSP, które mają dostęp do pamięci wspólnej. Poszczególne węzły są ze sobą połączone szybką magistralą i nie występuje wspólna dla wszystkich procesorów, realizowana fizycznie, przestrzeń adresowa. Podobną budowę mają klastry wyposażone w wieloprocesorowe węzły SMP, gdzie najczęściej każdy węzeł jest dwuprocesorowym lub czteroprocesorowym komputerem.

Systemy komputerowe z pamięcią rozproszoną mogą udostępniać użytkownikom logicznie spójną przestrzeń adresową, podzieloną na pamięci lokalne poszczególnych procesorów, implementowaną sprzętowo bądź programowo. Każdy procesor może uzyskiwać dostęp do fragmentu wspólnej przestrzeni adresowej, który jest alokowany w jego pamięci lokalnej, znacznie szybciej niż do pamięci, która fizycznie znajduje się na innym procesorze. Architektury tego typu określa się mianem NUMA (ang. *non-uniform memory access*). Bardziej złożonym mechanizmem jest cc-NUMA (ang. *cache coherent NUMA*), gdzie stosuje się protokoły uzgadniania zawartości pamięci podręcznej poszczególnych procesorów [25, 38].

1.3.3. Procesory wielordzeniowe

W ostatnich latach ogromną popularność zdobyły procesory wielordzeniowe, których pojawienie się stanowi wyzwanie dla twórców oprogramowania [5,42]. Konstrukcja takich procesorów polega na umieszczaniu w ramach pojedynczego pakietu, mającego postać układu scalonego, więcej niż jednego rdzenia (ang. *core*), logicznie stanowiącego oddzielny procesor. Aktualnie (zima 2010/11) dominują procesory dwurdzeniowe (ang. *dual-core*) oraz czterordzeniowe (ang. *quad-core*) konstrukcji firmy Intel oraz AMD, choć na rynku dostępne są również procesory ośmiordzeniowe (procesor Cell zaprojektowany wspólnie przez firmy Sony, Toshiba i IBM). Poszczególne rdzenie mają własną pamięć podręczną pierwszego poziomu, ale mogą mieć wspólną pamięć podręczną poziomu drugiego (Intel Core 2 Duo, Cell). Dzięki takiej filozofii konstrukcji procesory charakteryzują się znacznie efektywniejszym wykorzystaniem pamięci podręcznej i szybszym zapewnianiem jej spójności w ramach procesora wielordzeniowego. Dodatkowym atutem procesorów *multicore* jest mniejszy pobór energii niż w przypadku identycznej liczby procesorów „tradycyjnych".

Efektywne wykorzystanie procesorów wielordzeniowych wiąże się zatem z koniecznością opracowania algorytmów równoległych, szczególnie dobrze wykorzystujących pamięć podręczną. W pracy [30] wykazano, że w przypadku obliczeń z zakresu algebry liniowej szczególnie dobre wyniki daje wykorzystanie nowych sposobów reprezentacji macierzy opisanych w podrozdziale 1.2.3.

1.4. Optymalizacja programów uwzględniająca różne aspekty architektur

Wykorzystanie mechanizmów oferowanych przez współczesne komputery możliwe jest dzięki zastosowaniu kompilatorów optymalizujących kod pod kątem własności danej architektury. Przedstawimy teraz skrótowo rodzaje takiej optymalizacji. Trzeba jednak podkreślić, że zadowalająco dobre wykorzystanie własności architektur komputerowych jest możliwe po uwzględnieniu tak zwanego fundamentalnego trójkąta *algorytmy–sprzęt–kompilatory* (ang. *algorithms–hardware–compilers* [20]), co w praktyce oznacza konieczność opracowania odpowiednich algorytmów.

1.4.1. Optymalizacja maszynowa i skalarna

Podstawowymi rodzajami optymalizacji kodu oferowanymi przez kompilatory jest zależna od architektury komputera optymalizacja maszynowa

oraz niezależna sprzętowo optymalizacja skalarna [1,2]. Pierwszy rodzaj dotyczy właściwego wykorzystania architektury oraz specyficznej listy rozkazów procesora. W ramach optymalizacji skalarnej zwykle rozróżnia się dwa typy: optymalizację lokalną w ramach bloków składających się wyłącznie z instrukcji prostych bez instrukcji warunkowych oraz optymalizację globalną obejmującą kod całego podprogramu. Optymalizacja lokalna wykorzystuje techniki, takie jak eliminacja nadmiarowych podstawień, propagacja stałych, eliminacja wspólnych części kodu oraz nadmiarowych wyrażeń, upraszczanie wyrażeń. Optymalizacja globalna wykorzystuje podobne techniki, ale w obrębie całych podprogramów. Dodatkowo ważną techniką jest przemieszczanie fragmentów kodu. Przykładowo rozważmy następującą instrukcję iteracyjną.

```
1 for(i=0;i<n;i++){
2     a[i]=i*(1+b)*(c+d);
3 }
```

Wyrażenie `(1+b)*(c+d)` jest obliczane przy każdej iteracji pętli, dając za każdym razem identyczny wynik. Kompilator zastosuje przemieszczenie fragmentu kodu przed pętlę, co da następującą postać powyższej instrukcji iteracyjnej.

```
1 float temp1=(1+b)*(c+d);
2 for(i=0;i<n;i++){
3     a[i]=i*temp1;
4 }
```

Zatem, optymalizacja skalarna będzie redukować liczbę odwołań do pamięci i zmniejszać liczbę i czas wykonywania operacji, co powinno spowodować szybsze działanie programu.

1.4.2. Optymalizacja wektorowa i równoległa

W przypadku procesorów wektorowych oraz procesorów oferujących podobne rozszerzenia (na przykład SSE w popularnych procesorach firmy Intel) największy przyrost wydajności uzyskuje się dzięki optymalizacji wektorowej oraz równoległej (zorientowanej na wykorzystanie wielu procesorów), o ile tylko postać kodu źródłowego na to pozwala. Konstrukcjami, które są bardzo dobrze wektoryzowane, to pętle realizujące przetwarzanie tablic w ten sposób, że poszczególne itaracje pętli są od siebie niezależne. W prostych przypadkach uniemożliwiających bezpośrednią wektoryzację stosowane są odpowiednie techniki przekształcania kodu źródłowego [68]. Należy do nich usuwanie instrukcji warunkowych z wnętrza pętli, ich rozdzielanie, czy też przenoszenie poza pętlę przypadków skrajnych. Niestety, w wielu

przypadkach nie istnieją bezpośrednie proste metody przekształcania kodu źródłowego w ten sposób, by możliwa była wektoryzacja pętli. Jako przykład rozważmy następujący prosty fragment kodu źródłowego.

```
1 for ( i =0; i <n-1; i++){
2       a[ i +1]=a[ i ]+b[ i ];
3 }
```

Do obliczenia wartości każdej następnej składowej tablicy jest wykorzystywana obliczona wcześniej wartość poprzedniej składowej. Taka pętla nie może być automatycznie zwektoryzowana i tym bardziej zrównoleglona. Zatem wykonanie konstrukcji tego typu będzie się odbywało bez udziału jednostek wektorowych i na ogół przebiegało z bardzo niewielką wydajnością obliczeń. W przypadku popularnych procesorów będzie to zaledwie około 10% maksymalnej wydajności, w przypadku zaś procesorów wektorowych znacznie mniej. Z drugiej strony, fragmenty programów wykonywane z niewielką wydajnością znacznie obniżają wypadkową wydajność obliczeń. W pracy [49] zawarto nawet sugestię, że w przypadku programów z dominującymi fragmentami skalarnymi należy rozważyć rezygnację z użycia superkomputera Cray X1, gdyż takie programy będą w stanie wykorzystać jedynie znikomy ułamek maksymalnej teoretycznej wydajności pojedynczego procesora.

Dzięki automatycznej optymalizacji równoległej możliwe jest wykorzystanie wielu procesorów w komputerze. Instrukcje programu są dzielone na wątki (ciągi instrukcji wykonywanych na pojedynczych procesorach), które mogą być wykonywane równolegle (jednocześnie). Zwykle na wątki dzielona jest pula iteracji pętli. Optymalizacja równoległa jest często stosowana w połączeniu z optymalizacją wektorową. Pętle wewnętrzne są wektoryzowane, zewnętrzne zaś zrównoleglane.

1.5. Programowanie równoległe

Istnieje kilka ukierunkowanych na możliwie pełne wykorzystanie oferowanej mocy obliczeniowej metodologii tworzenia oprogramowania na komputery równoległe. Do ważniejszych należy zaliczyć zastosowanie kompilatorów optymalizujących [2, 66, 68], które mogą dokonać optymalizacji kodu na poziomie języka maszynowego (optymalizacja maszynowa) oraz użytego języka programowania wysokiego poziomu (optymalizacja skalarna, wektorowa i równoległa). Niestety, taka automatyczna optymalizacja pod kątem wieloprocesorowości zastosowana do typowych programów, które implementują klasyczne algorytmy sekwencyjne, na ogół nie daje zadowalających rezultatów.

Zwykle konieczne staje się rozważenie czterech aspektów tworzenia efektywnych programów na komputery równoległe [21, rozdział 3.2], [25].

1. *Identyfikacja równoległości obliczeń* polegająca na wskazaniu fragmentów algorytmu lub kodu, które mogą być wykonywane równolegle, dając przy każdym wykonaniu programu ten sam wynik obliczeń jak w przypadku programu sekwencyjnego.
2. *Wybór strategii dekompozycji* programu na części wykonywane równolegle. Możliwe są dwie zasadnicze strategie: *równoległość zadań* (ang. *task parallelism*), gdzie podstawę analizy stanowi graf zależności między poszczególnymi (na ogół) różnymi funkcjonalnie zadaniami obliczeniowymi oraz *równoległość danych* (ang. *data parallelism*), gdzie poszczególne wykonywane równolegle zadania obliczeniowe dotyczą podobnych operacji wykonywanych na różnych danych.
3. *Wybór modelu programowania*, który determinuje wybór konkretnego języka programowania wspierającego równoległość obliczeń oraz środowiska wykonania programu. Jest on dokonywany w zależności od konkretnej architektury komputerowej, która ma być użyta do obliczeń. Zatem, możliwe są dwa główne modele: pierwszy wykorzystujący pamięć wspólną oraz drugi, oparty na wymianie komunikatów (ang. *message-passing*), gdzie nie zakłada się istnienia wspólnej pamięci dostępnej dla wszystkich procesorów.
4. *Styl implementacji równoległości* w programie, wynikający z przyjętej wcześniej strategii dekompozycji oraz modelu programowania (przykładowo zrównoleglanie pętli, programowanie zadań rekursywnych bądź model SPMD).

Przy programowaniu komputerów równoległych z pamięcią wspólną wykorzystuje się najczęściej języki programowania Fortran i C/C++, ze wsparciem dla OpenMP [6]. Jest to standard definiujący zestaw dyrektyw oraz kilku funkcji bibliotecznych, umożliwiający specyfikację równoległości wykonania poszczególnych fragmentów programu oraz synchronizację wielu wątków działających równolegle. Zrównoleglanie możliwe jest na poziomie pętli oraz poszczególnych fragmentów kodu (sekcji). Komunikacja między wątkami odbywa się poprzez wspólną pamięć. Istnieje również możliwość specyfikowania operacji atomowych. Program rozpoczyna działanie jako pojedynczy wątek. W miejscu specyfikacji równoległości (za pomocą odpowiednich dyrektyw definiujących region równoległy) następuje rozdzielenie wątku głównego na grupę wątków działających równolegle aż do miejsca złączenia. W programie może wystąpić wiele regionów równoległych. Programowanie w OpenMP omawiamy szczegółowo w rozdziale 4.

Architektury wieloprocesorowe z pamięcią rozproszoną programuje się zwykle wykorzystując środowisko MPI (ang. *Message Passing Interface* [51]), wspierające model programu typu SPMD (ang. *Single Program, Multiple*

Data) lub powoli wychodzące z użycia środowisko PVM (ang. *Parallel Virtual Machine* [19]). Program SPMD w MPI zakłada wykonanie pojedynczej instancji programu (procesu) na każdym procesorze biorącym udział w obliczeniach [36]. Standard MPI definiuje zbiór funkcji umożliwiających programowanie równoległe za pomocą wymiany komunikatów (ang. *message-passing*). Oprócz funkcji ogólnego przeznaczenia inicjujących i kończących pracę procesu w środowisku równoległym, najważniejszą grupę stanowią funkcje przesyłania danych między procesami. Wyróżnia się tu komunikację między dwoma procesami (ang. *point-to-point*) oraz komunikację strukturalną w ramach grupy wątków (tak zwaną komunikację kolektywną oraz operacje redukcyjne). Programowanie w MPI omawiamy szczegółowo w rozdziałach 5 oraz 6

Innymi mniej popularnymi narzędziami programowania komputerów z pamięcią rozproszoną są języki Co-array Fortran (w skrócie CAF [21, podrozdział 12.4]) oraz Unified Parallel C (w skrócie UPC [8]). Oba rozszerzają standardowe języki Fortran i C o mechanizmy umożliwiające programowanie równoległe. Podobnie jak MPI, CAF zakłada wykonanie programu w wielu kopiach (obrazach, ang. *images*) posiadających swoje własne dane. Poszczególne obrazy mogą się odwoływać do danych innych obrazów, bez konieczności jawnego użycia operacji przesyłania komunikatów. Oczywiście CAF posiada również mechanizmy synchronizacji działania obrazów. Język UPC, podobnie jak MPI oraz CAF, zakłada wykonanie programu, opierając się na modelu SPMD. Rozszerza standard C o mechanizmy definiowania danych we wspólnej przestrzeni adresowej, która fizycznie jest alokowana porcjami w pamięciach lokalnych poszczególnych procesów i umożliwia dostęp do nich bez konieczności jawnego użycia funkcji przesyłania komunikatów.

Rozdział 2

Modele realizacji obliczeń równoległych

W niniejszym rozdziale przedstawimy wybrane modele realizacji obliczeń na współczesnych komputerach wektorowych oraz równoległych.

2.1. Przyspieszenie

Podstawową miarą charakteryzującą wykonanie programu na komputerze jest czas obliczeń. Stosowane techniki optymalizacji „ręcznej", gdzie dostosowuje się program do danej architektury komputera poprzez wprowadzenie zmian w kodzie źródłowym, bądź też opracowanie nowego algorytmu dla danego typu architektury komputera mają na celu skrócenie czasu działania programu. W konsekwencji możliwe jest rozwiązywanie w pewnym, akceptowalnym dla użytkownika czasie problemów o większych rozmiarach lub też w przypadku obliczeń numerycznych uzyskiwanie większej dokładności wyników, na przykład poprzez zagęszczenie podziału siatki lub wykonanie większej liczby iteracji danej metody. Jednakże operowanie bezwzględnymi czasami wykonania poszczególnych programów bądź też ich fragmentów realizujących konkretne algorytmy może nie odzwierciedlać w wystarczającym stopniu zysku czasowego, jaki uzyskuje się dzięki optymalizacji. Stąd wygodniej jest posługiwać się terminem przyspieszenie (ang. *speedup*), pokazującym, ile razy szybciej działa program (lub jego fragment realizujący konkretny algorytm) zoptymalizowany na konkretną architekturę komputera względem pewnego programu uznanego za punkt odniesienia. Podamy teraz za książkami [38], [50] oraz [15] najważniejsze pojęcia z tym związane.

Definicja 2.1. ([38]) *Przyspieszeniem bezwzględnym algorytmu równoległego nazywamy wielkość*

$$s_p^\star = \frac{t_1^\star}{t_p}, \tag{2.1}$$

gdzie $t_1^\star$ jest czasem wykonania najlepszej realizacji algorytmu sekwencyjnego, a t_p czasem działania algorytmu równoległego na p procesorach.

Często wielkość $t_1^\star$ nie jest znana, a zatem w praktyce używa się również innej definicji przyspieszenia.

Definicja 2.2. ([38]) *Przyspieszeniem względnym algorytmu równoległego nazywamy wielkość*

$$s_p = \frac{t_1}{t_p}, \tag{2.2}$$

gdzie t_1 jest czasem wykonania algorytmu na jednym procesorze, a t_p czasem działania tego algorytmu na p procesorach.

W przypadku analizy programów wektorowych przyspieszenie uzyskane dzięki wektoryzacji definiuje się podobnie, jako zysk czasowy osiągany względem najszybszego algorytmu skalarnego.

Definicja 2.3. ([50]) *Przyspieszeniem algorytmu wektorowego względem najszybszego algorytmu skalarnego nazywamy wielkość*

$$s_v^\star = \frac{t_s^\star}{t_v}, \tag{2.3}$$

gdzie $t_s^\star$ jest czasem działania najlepszej realizacji algorytmu skalarnego, a t_v czasem działania algorytmu wektorowego.

Algorytmy wektorowe często charakteryzują się większą liczbą operacji arytmetycznych w porównaniu do najlepszych (najszybszych) algorytmów skalarnych. W praktyce użyteczne są algorytmy, które charakteryzują się ograniczonym wzrostem złożoności obliczeniowej (traktowanej jako liczba operacji zmiennopozycyjnych w algorytmie) w stosunku do liczby operacji wykonywanych przez najszybszy algorytm skalarny, co intuicyjnie oznacza, że zysk wynikający z użycia wektorowości nie będzie „pochłaniany" przez znaczący wzrost liczby operacji algorytmu wektorowego dla większych rozmiarów problemu. Algorytmy o tej własności nazywamy zgodnymi (ang. *consistent*) z najlepszym algorytmem skalarnym rozwiązującym dany problem. Formalnie precyzuje to następująca definicja.

Definicja 2.4. ([50]) *Algorytm wektorowy rozwiązujący problem o rozmiarze n nazywamy zgodnym z najszybszym algorytmem skalarnym, gdy*

$$\lim_{n\to\infty} \frac{V(n)}{S(n)} = C < +\infty \tag{2.4}$$

gdzie $V(n)$ oraz $S(n)$ oznaczają odpowiednio liczbę operacji zmiennopozycyjnych algorytmu wektorowego i najszybszego algorytmu skalarnego.

Jak zaznaczyliśmy w podrozdziale 1.1, większość współczesnych procesorów implementuje równoległość na wielu poziomach dla osiągnięcia dużej wydajności obliczeń, co oczywiście wymaga użycia odpowiednich algorytmów, które będą w stanie wykorzystać możliwości oferowane przez architekturę konkretnego procesora. Użycie komputerów wieloprocesorowych wiąże się dodatkowo z koniecznością uwzględnienia równoległości obliczeń również na poziomie grupy oddzielnych procesorów. Zatem opracowywanie nowych algorytmów powinno uwzględniać możliwości optymalizacji kodu źródłowego pod kątem użycia równoległości na wielu poziomach oraz właściwego wykorzystania hierarchii pamięci, dzięki czemu czas obliczeń może skrócić się znacząco.

Definicje 2.2 i 2.3 uwzględniające jedynie równoległość na poziomie grupy procesorów oraz wektorowości nie oddają w pełni zysku czasowego, jaki otrzymuje się poprzez zastosowanie algorytmu opracowanego z myślą o konkretnym sprzęcie komputerowym, gdzie równoległość może być zaimplementowana na wielu poziomach. Zatem podobnie jak w książce [21, podrozdział

8.8], będziemy definiować przyspieszenie jako miarę tego, jak zmienia się czas obliczeń dzięki zastosowanej optymalizacji dla danej architektury komputerowej. Otrzymamy w ten sposób następującą definicję.

Definicja 2.5. ([50]) *Przyspieszeniem algorytmu A względem algorytmu B nazywamy wielkość*

$$s = \frac{t_B}{t_A}, \tag{2.5}$$

gdzie t_A jest czasem działania algorytmu A, a t_B czasem działania algorytmu B na danym systemie komputerowym dla takiego samego rozmiaru problemu.

Wydajność obliczeniową współczesnych systemów komputerowych [1] (ang. *computational speed of modern computer architectures* inaczej *performance of computer programs on modern computer architectures* [15]) będziemy podawać w milionach operacji zmiennopozycyjnych na sekundę (Mflops) i definiować jako

$$r = \frac{N}{t} \quad \text{Mflops}, \tag{2.6}$$

gdzie N oznacza liczbę operacji zmiennopozycyjnych wykonanych w czasie t mikrosekund. Producenci sprzętu podają opierając się na wzorze (2.6) teoretyczną maksymalną wydajność obliczeniową r_{peak} (ang. *peak performance*). Na ogół dla konkretnych programów wykonywanych na danym sprzęcie zachodzi $r < r_{peak}$. Oczywiście, im wartość obliczona ze wzoru (2.6) jest większa, tym lepsze wykorzystanie możliwości danej architektury komputerowej. Oczywiście, różne programy rozwiązujące dany problem obliczeniowy różnymi metodami mogą się charakteryzować różnymi liczbami wykonywanych operacji, stąd wydajność będziemy traktować jako pomocniczą charakterystykę jakości algorytmu wykonywanego na konkretnym sprzęcie. W przypadku gdy różne algorytmy charakteryzują się identyczną liczbą operacji, wielkość (2.6) stanowi ważne kryterium porównania algorytmów przy jednoczesnym wskazaniu na stopień wykorzystania możliwości sprzętu.

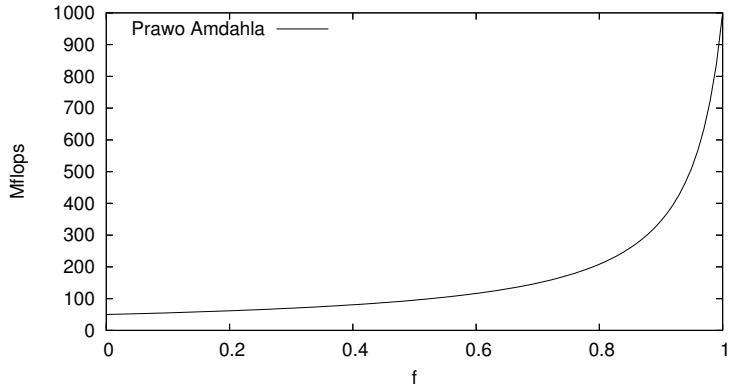
Przekształcając wzór (2.6), wnioskujemy, że czas wykonania programu spełnia następującą zależność

$$t = \frac{N}{r} \quad \mu\text{s}, \tag{2.7}$$

co jest równoważne

$$t = \frac{N}{10^6 r} \quad \text{s}. \tag{2.8}$$

[1] Innym polskim tłumaczeniem tego terminu jest *szybkość komputerów w zakresie obliczeń numerycznych* [38].



Rysunek 2.1. Prawo Amdahla dla $V = 1000$ oraz $S = 50$ Mflops

Dla poszczególnych fragmentów programu może być osiągnięta różna wydajność, a zatem wzór (2.6) opisuje tylko średnią wydajność obliczeniową danej architektury. Poniżej przedstawimy najważniejsze modele, które znacznie lepiej oddają specyfikę wykonania programów na współczesnych komputerach.

2.2. Prawo Amdahla

Niech f będzie częścią programu składającego się z N operacji zmiennopozycyjnych, dla którego osiągnięto wydajność V, a $1 - f$ częścią wykonywaną przy wydajności S, przy czym $V \gg S$. Wówczas korzystając z (2.7), otrzymujemy łączny czas wykonywania obliczeń obu części programu

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

oraz uwzględniając (2.6) otrzymujemy wydajność obliczeniową komputera, który wykonuje dany program

$$r = \frac{1}{\frac{f}{V} + \frac{(1-f)}{S}} \quad \text{Mflops.} \tag{2.9}$$

Wzór (2.9) nosi nazwę *prawo Amdahla* [15,16] i opisuje wpływ optymalizacji fragmentu programu na wydajność obliczeniową danej architektury.

Jako przykład rozważmy sytuację, gdy $V = 1000$ oraz $S = 50$ Mflops. Rysunek 2.1 pokazuje osiągniętą wydajność (Mflops) w zależności od wartości f. Możemy zaobserwować, że relatywnie duża wartość $f = 0.8$, dla

której osiągnięta jest maksymalna wydajność, skutkuje wydajnością wykonania całego programu równą 200 Mflops, a zatem cały program wykorzystuje zaledwie 20% teoretycznej maksymalnej wydajności. Oznacza to, że aby uzyskać zadowalająco krótki czas wykonania programu, należy zadbać o zoptymalizowanie jego najwolniejszych części. Zauważmy też, że w omawianym przypadku największy wzrost wydajności obliczeniowej danej architektury uzyskujemy przy zmianie wartości f od 0.9 do 1.0. Zwykle jednak fragmenty programu, dla których jest osiągnięta niewielka wydajność, to obliczenia w postaci rekurencji bądź też realizujące dostęp do pamięci w sposób nieoptymalny, które wymagają zastosowania specjalnych algorytmów dla osiągnięcia zadowalającego czasu wykonania.

2.3. Model Hockneya-Jesshope'a

Innym modelem, który dokładniej charakteryzuje obliczenia wektorowe jest model Hockneya - Jesshope'a obliczeń wektorowych [15,31], pokazujący wydajność komputera wykonującego obliczenia w postaci pętli. Może on być również przydatny przy analizie i predykcji czasu działania programów wykonywanych na komputerach z popularnymi procesorami wyposażonymi w rozszerzenia wektorowe (np. SSE). Rozważmy pętlę o N iteracjach. Wydajność, jaką osiąga komputer wykonując takie obliczenia, wyraża się wzorem

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \text{ Mflops}, \tag{2.10}$$

gdzie r_∞ oznacza wydajność komputera (Mflops) wykonującego „nieskończoną" pętlę (bardzo długą), $n_{1/2}$ zaś jest długością (liczbą iteracji) pętli, dla której osiągnięta jest wydajność około $r_\infty/2$. Przykładowo, operacja DOT wyznaczenia iloczynu skalarnego wektorów $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$

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

ma postać następującej pętli o liczbie iteracji równej N.

```
1 dot=0.0;
2 for(i=0;i<n;i++){
3     dot+=y[i]*x[i];
4 }
```

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

$$T_{DOT}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N). \tag{2.11}$$

Podobnie zdefiniowana wzorem (1.1) operacja AXPY może być w najprostszej postaci[2] zaprogramowana jako następująca konstrukcja iteracyjna.

```
1 for ( i=0;i<n ; i++){
2     y[ i]+=alpha*x[ i ];
3 }
```

Na każdą iterację pętli przypadają dwie operacje arytmetyczne, stąd czas jej wykonania wyraża się wzorem

$$T_{AXPY}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N). \tag{2.12}$$

Oczywiście, wielkości r_∞ oraz $n_{1/2}$ występujące odpowiednio we wzorach (2.11) i (2.12) są na ogół różne, nawet dla tego samego procesora. Wadą modelu jest to, że nie uwzględnia on zagadnień związanych z organizacją pamięci w komputerze. Może się zdarzyć, że taka sama pętla, operująca na różnych zestawach danych alokowanych w pamięci operacyjnej w odmienny sposób, będzie w każdym przypadku wykonywana przy bardzo różnych wydajnościach. Może to być spowodowane konfliktami w dostępie do banków pamięci bądź też innym schematem wykorzystania pamięci podręcznej.

2.3.1. Przykład zastosowania

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

$$L\mathbf{x} = \mathbf{b}, \tag{2.13}$$

gdzie $\mathbf{x}$, $\mathbf{b} \in \mathbb{R}^N$ oraz

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

Układ może być rozwiązany za pomocą następującego algorytmu [59]:

$$\begin{cases} x_1 = b_1/a_{11} \\ x_i = \left(b_i - \sum_{k=1}^{i-1} a_{ik}x_k\right)/a_{ii} \quad \text{dla } i = 2, \ldots, N. \end{cases} \tag{2.14}$$

[2] W rzeczywistości kod źródłowy operacji DOT i AXPY w bibliotece BLAS jest bardziej skomplikowany. Składowe wektorów nie muszą być kolejnymi składowymi tablic oraz zaimplementowany jest mechanizm rozwijania pętli w sekwencje instrukcji (ang. *loop unrolling*).

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

$$\begin{aligned} T_1(N) &= \sum_{k=1}^{N-1} T_{DOT}(k) = \frac{2\cdot 10^{-6}}{r_\infty}\left(n_{1/2}(N-1) + \sum_{k=1}^{N-1} k\right) \\ &= \frac{2\cdot 10^{-6}}{r_\infty}(N-1)\left(n_{1/2} + \frac{N}{2}\right). \end{aligned} \tag{2.15}$$

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

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

Oczywiście zachodzi

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

Stąd otrzymujemy następujący wzór [58]:

$$\begin{cases} \mathbf{y}_0 = \mathbf{b} \\ \mathbf{y}_i = L_i^{-1}\mathbf{y}_{i-1} \quad \text{dla } i = 1, \ldots, N \\ \mathbf{x} = \mathbf{y}_N \end{cases} \tag{2.16}$$

Operacja mnożenia macierzy L_i^{-1} przez wektor $\mathbf{y}_{i-1}$ nie wymaga jawnego wyznaczania postaci macierzy. Istotnie, rozpisując wzór (2.16) otrzymujemy

$$\mathbf{y}_i = \begin{pmatrix} y_1^{(i)} \\ y_2^{(i)} \\ \vdots \\ y_i^{(i)} \\ \vdots \\ y_{N-1}^{(i)} \\ y_N^{(i)} \end{pmatrix} = L_i^{-1} \begin{pmatrix} y_1^{(i-1)} \\ y_2^{(i-1)} \\ \vdots \\ y_i^{(i-1)} \\ \vdots \\ y_{N-1}^{(i-1)} \\ y_N^{(i-1)} \end{pmatrix} = \begin{pmatrix} y_1^{(i-1)} \\ \vdots \\ y_{i-1}^{(i-1)} \\ y_i^{(i-1)}/a_{ii} \\ y_{i+1}^{(i-1)} - a_{i+1,i}y_i^{(i-1)}/a_{ii} \\ \vdots \\ y_N^{(i-1)} - a_{N,i}y_i^{(i-1)}/a_{ii} \end{pmatrix} \tag{2.17}$$

W algorytmie opartym na wzorach (2.16) i (2.17) nie trzeba składować wszystkich wyznaczanych wektorów. Każdy kolejny wektor $\mathbf{y}_i$ będzie składowany na miejscu poprzedniego, to znaczy $\mathbf{y}_{i-1}$. Stąd operacja aktualizacji wektora we wzorze (2.17) przyjmie postać sekwencji operacji skalarnej

$$y_i \leftarrow y_i/a_{ii}, \tag{2.18}$$

a następnie wektorowej

$$\begin{pmatrix} y_{i+1} \\ \vdots \\ y_N \end{pmatrix} \leftarrow \begin{pmatrix} y_{i+1} \\ \vdots \\ y_N \end{pmatrix} - y_i \begin{pmatrix} a_{i+1,i} \\ \vdots \\ a_{N,i} \end{pmatrix}. \tag{2.19}$$

Zauważmy, że (2.19) to właśnie operacja AXPY. Stąd podobnie jak w przypadku poprzedniego algorytmu, pomijając czas potrzebny do wykonania dzielenia (2.18), otrzymujemy

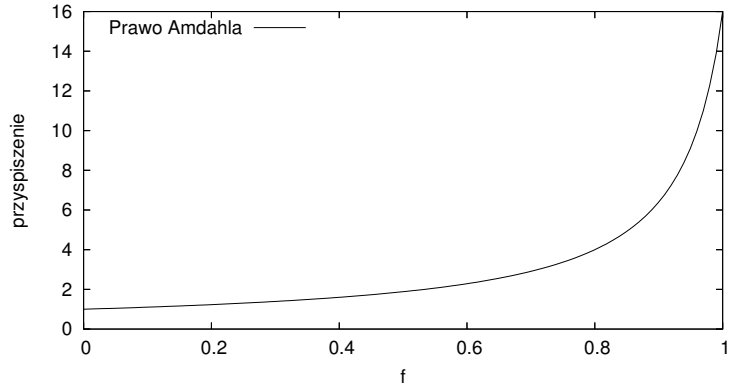
$$\begin{aligned} T_2(N) &= \sum_{k=1}^{N-1} T_{AXPY}(N-k) = \frac{2\cdot 10^{-6}}{r_\infty}\left(n_{1/2}(N-1) + \sum_{k=1}^{N-1}(N-k)\right) \\ &= \frac{2\cdot 10^{-6}}{r_\infty}(N-1)\left(n_{1/2} + \frac{N}{2}\right). \end{aligned} \tag{2.20}$$

Zatem, dla obu algorytmów czas wykonania operacji wektorowych wyraża się podobnie w postaci funkcji zależnych od parametrów r_∞, $n_{1/2}$, właściwych dla operacji DOT i AXPY. Dla komputera wektorowego C3210 wartości r_∞ oraz $n_{1/2}$ dla operacji DOT wynoszą odpowiednio $r_\infty = 18$, $n_{1/2} = 36$, zaś dla operacji AXPY mamy $r_\infty = 16$, $n_{1/2} = 26$. Zatem $T_1(1000) = 0.059$ oraz $T_2(1000) = 0.066$. Zauważmy, że mimo jednakowej, wynoszącej w przypadku obu algorytmów, liczby operacji arytmetycznych N^2, wyznaczony czas działania każdego algorytmu jest inny.

W pracy [63] przedstawiono inne zastosowanie omówionego wyżej modelu Hockney'a-Jesshope'a dla wyznaczania optymalnych wartości parametrów metody *divide and conquer* wyznaczania rozwiązania liniowych równań rekurencyjnych o stałych współczynnikach.

2.4. Prawo Amdahla dla obliczeń równoległych

Prawo Amdahla ma swój odpowiednik również dla obliczeń równoległych [15]. Przypuśćmy, że czas wykonania programu na jednym procesorze wynosi t_1. Niech f oznacza część programu, która może być idealnie zrównoleglona na p procesorach. Pozostała sekwencyjna część programu $(1-f)$



Rysunek 2.2. Prawo Amdahla dla obliczeń równoległych, $p = 16$

będzie wykonywana na jednym procesorze. Łączny czas wykonania programu równoległego przy użyciu p procesorów wynosi

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

Stąd przyspieszenie w sensie definicji 2.2 wyraża się wzorem [15]:

$$s_p = \frac{t_1}{t_p} = \frac{p}{f + (1-f)p}. \tag{2.21}$$

Rysunek 2.2 pokazuje wpływ zrównoleglonej części f na przyspieszenie względne programu. Można zaobserwować bardzo duży negatywny wpływ części sekwencyjnej (niezrównoleglonej) na osiągnięte przyspieszenie – podobnie jak w przypadku podstawowej wersji prawa Amdahla określonego wzorem (2.9).

2.5. Model Gustafsona

Prawo Amdahla zakłada stały rozmiar rozwiązywanego problemu. Tymczasem zwykle wraz ze wzrostem dostępnych zasobów (zwykle procesorów) zwiększa się również rozmiar problemu. W pracy [26] Gustafson zaproponował model alternatywny dla prawa Amdahla. Głównym jego założeniem jest fakt, że w miarę wzrostu posiadanych zasobów obliczeniowych (liczba procesorów, wydajność komputera) zwiększa się rozmiar rozwiązywanych problemów. Zatem w tym modelu przyjmuje się za stały czas obliczeń.

Niech t_s oraz t_p oznaczają odpowiednio czas obliczeń sekwencyjnych (na jednym procesorze) oraz czas obliczeń na komputerze równoległym o p procesorach. Niech dalej f będzie częścią czasu t_p wykonywaną na w sposób równoległy, zaś $1 - f$ częścią sekwencyjną (wykonywaną na jednym procesorze). Dla uproszczenia przyjmijmy, że $t_p = 1$. Wówczas czas wykonania tego programu na jednym procesorze wyrazi się wzorem

$$t_s = pf + 1 - f.$$

Przyspieszenie[3] programu równoległego wyraża się wzorem [15]

$$s_{p,f} = 1 + f(1 - p) = p + (1 - p)(1 - f).$$

Użyteczność tego modelu została wykazana w pracy [27] przy optymalizacji programów na komputer NCube [15]. Model uwzględnia również typowe podejście stosowane obecnie w praktyce. Gdy dysponujemy większymi zasobami obliczeniowymi (np. liczbą procesorów), zwiększamy rozmiary rozwiązywanych problemów i przyjmujemy, że czas obliczeń, który jest dla nas akceptowalny, pozostaje bez zmian.

2.6. Zadania

Poniżej zamieściliśmy kilka zadań do wykonania samodzielnego. Ich celem jest utrwalenie wiadomości przedstawionych w tym rozdziale.

Zadanie 2.1.
System wieloprocesorowy składa się ze 100 procesorów, z których każdy może pracować z wydajnością równą 2 Gflopy. Jaka będzie wydajność systemu (mierzona w Gflopach) jeśli 10% kodu jest sekwencyjna, a pozostałe 90% można zrównoleglić?

Zadanie 2.2.
Prawo Amdahla dla obliczeń równoległych mówi jakie będzie przyspieszenie programu na p procesorach, w przypadku gdy tylko pewna część f czasu obliczeń może zostać zrównoleglona.

1. Wyprowadź wzór na to przyspieszenie.
2. Udowodnij, że maksymalne przyspieszenie na procesorach, w przypadku gdy niezrównoleglona pozostanie część f programu, wynosi $1/(f)$.

[3] Przyspieszenie wyznaczane przy użyciu modelu Gustafsona nazywa się czasami *przyspieszeniem skalowalnym* (ang. *scaled speedup*) [67].

Zadanie 2.3.

Korzystając z prawa Amdahla dla obliczeń równoległych wyznaczyć przyspieszenie programu, w którym 80% czasu obliczeń wykonywana jest na czterech procesorach, zaś pozostałe 20% stanowią obliczenia sekwencyjne. Zakładamy, że ten algorytm wykonywany na jednym procesorze jest optymalnym algorytmem sekwencyjnym rozwiązującym dany problem.

Zadanie 2.4.

Niech będzie dany program, w którym 80% czasu obliczeń może być zrównoleglona (wykonywana na p procesorach), zaś pozostałe 20% stanowią obliczenia sekwencyjne. Zakładamy, że ten program wykonywany na jednym procesorze jest realizacją optymalnego algorytmu sekwencyjnego rozwiązującego dany problem. Wyznaczyć minimalną liczbę procesorów, dla której zostanie osiągnięte przyspieszenie równe 4.

Zadanie 2.5.

Niech będzie dany program, w którym część f czasu obliczeń może być zrównoleglona (wykonywana na 10 procesorach), zaś pozostałe $1 - f$ to obliczenia sekwencyjne. Zakładamy, że ten program wykonywany na jednym procesorze jest realizacją optymalnego algorytmu sekwencyjnego rozwiązującego dany problem. Ile musi wynosić (co najmniej) wartość f aby efektywność programu nie była mniejsza od 0.5 ?

Zadanie 2.6.

Niech $A \in \mathbb{R}^{m \times n}$, $\mathbf{y} \in \mathbb{R}^m$ oraz $\mathbf{x} \in \mathbb{R}^n$. Rozważ dwa algorytmy mnożenia macierzy przez wektor postaci $\mathbf{y} \leftarrow \mathbf{y} + A\mathbf{x}$:

1. **zwykły** – iloczyn skalarny wierszy przez wektor,

$$y_i \leftarrow y_i + \sum_{j=1}^{n} a_{ij} x_j, \quad \text{dla} \quad i = 1, \ldots, m,$$

2. **wektorowy** – suma kolumn macierzy przemnożonych przez składowe wektora,

$$\mathbf{y} \leftarrow \mathbf{y} + \sum_{j=1}^{n} x_j A_{*,j}.$$

Wyznacz czas działania obu algorytmów na tym samym komputerze (przy danych r i dla obu pętli: AXPY i DOT).

Zadanie 2.7.

Spróbuj opracować wzór na przyspieszenie dla równoległego algorytmu szukającego zadanej wartości w tablicy. Niech t_s to będzie czas sekwencyjny.

W wersji równoległej przeszukiwaną tablicę elementów dzielimy na p części. Załóż, że szukany element został znaleziony po czasie Δt, gdzie $\Delta t < t_s/p$. Dla jakiego układu danych w przestrzeni rozwiązań przyspieszenie to będzie najmniejsze, a dla jakiego równoległa wersja daje największe korzyści? Przyspieszenie, jakie uzyskamy w przypadku równoległego algorytmu szukającego określane jest mianem przyspieszenia superliniowego. Oznacza to, że dla algorytmu równoległego wykonywanego na p procesorach można uzyskać przyspieszenie większe niż p, czasem nawet wielokrotnie większe.

Zadanie 2.8.

Łącząc wzór na przyspieszenie wynikający z prawa Amdahla dla obliczeń równoległych z analizą na przyspieszenie superliniowe z zadania 2.7, napisz wzór na przyspieszenie dla algorytmu szukającego w przypadku gdy część f algorytmu musi wykonać się sekwencyjnie.

Rozdział 3

BLAS: podstawowe podprogramy algebry liniowej

Jedną z najważniejszych metod opracowywania bardzo szybkich algorytmów spełniających postulaty właściwego wykorzystania pamięci podręcznej, które wykorzystywałyby w dużym stopniu możliwości współczesnych procesorów, jest zastosowanie do ich konstrukcji podprogramów z biblioteki BLAS, które umożliwiają efektywne wykorzystanie hierarchii pamięci [9,37] i zapewniają przenośność kodu między różnymi architekturami [4, 18]. Podejście to zostało zastosowane przy konstrukcji biblioteki LAPACK [3], zawierającej zestaw podprogramów rozwiązujących typowe zagadnienia algebry liniowej (rozwiązywanie układów równań liniowych o macierzach pełnych i pasmowych oraz algebraiczne zagadnienie własne).

3.1. BLAS poziomów 1, 2 i 3

W roku 1979 zaproponowano standard dla podprogramów realizujących podstawowe operacje algebry liniowej (ang. *Basic Linear Algebra Subprograms* – BLAS) [44]. Twórcy oprogramowania matematycznego wykorzystali fakt, że programy realizujące metody numeryczne z dziedziny algebry liniowej składają się z pewnej liczby podstawowych operacji typu skalowanie wektora, dodawanie wektorów czy też iloczyn skalarny. Powstała kolekcja podprogramów napisanych w języku Fortran 77, które zostały użyte do konstrukcji biblioteki LINPACK [12], zawierającej podprogramy do rozwiązywania układów równań liniowych o macierzach pełnych i pasmowych. Zaowocowało to nie tylko klarownością i czytelnością kodu źródłowego programów wykorzystujących BLAS, ale również dało możliwość efektywnego przenoszenia kodu źródłowego między różnymi rodzajami architektur komputerowych, tak by w maksymalnym stopniu wykorzystać ich własności (ang. *performance portability*). Twórcy oprogramowania na konkretne komputery mogli dostarczać biblioteki podprogramów BLAS zoptymalizowane na konkretny typ procesora. Szczególnie dobrze można zoptymalizować podprogramy z biblioteki BLAS na procesory wektorowe [17].

Następnym krokiem w rozwoju standardu BLAS były prace nad biblioteką LAPACK [3], wykorzystującą algorytmy blokowe, której funkcjonalność pokryła bibliotekę LINPACK oraz EISPACK [23], zawierającą podprogramy do rozwiązywania algebraicznego zagadnienia własnego. Zdefiniowano zbiór podprogramów zawierających działania typu macierz - wektor (biblioteka BLAS poziomu drugiego [14]) oraz macierz - macierz (inaczej BLAS poziomu trzeciego [13]), wychodząc naprzeciw możliwościom oferowanym przez nowe procesory, czyli mechanizmom zaawansowanego wykorzystania hierarchii pamięci. Szczególnie poziom 3 oferował zadowalającą lokalność danych i w konsekwencji bardzo dużą efektywność. Oryginalny zestaw podprogramów BLAS przyjęto określać mianem BLAS poziomu 1.

Poniżej przedstawiamy najważniejsze operacje z poszczególnych poziomów BLAS-u. Pełny wykaz można znaleźć pod adresem `http://www.netlib.org/blas/blasqr.ps` oraz w książkach [3, 15].

Poziom 1: operacje na wektorach

— $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$, $\mathbf{x} \leftarrow \alpha\mathbf{x}$, $\mathbf{y} \leftarrow \mathbf{x}$, $\mathbf{y} \leftrightarrow \mathbf{x}$, $dot \leftarrow \mathbf{x}^T\mathbf{y}$, $nrm2 \leftarrow \|\mathbf{x}\|_2$, $asum \leftarrow \|re(\mathbf{x})\|_1 + \|im(\mathbf{x})\|_1$.

Poziom 2: operacje typu macierz-wektor

— iloczyn macierz-wektor: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$, $\mathbf{y} \leftarrow \alpha A^T\mathbf{x} + \beta\mathbf{y}$
— aktualizacje macierzy typu „rank-1": $A \leftarrow \alpha\mathbf{x}\mathbf{y}^T + A$
— aktualizacje macierzy symetrycznych typu „rank-1" oraz „rank-2": $A \leftarrow \alpha\mathbf{x}\mathbf{x}^T + A$, $A \leftarrow \alpha\mathbf{x}\mathbf{y}^T + \alpha\mathbf{y}\mathbf{x}^T + A$,
— mnożenie wektora przez macierz trójkątną: $\mathbf{x} \leftarrow T\mathbf{x}$, $\mathbf{x} \leftarrow T^T\mathbf{x}$,
— rozwiązywanie układów równań liniowych o macierzach trójkątnych: $\mathbf{x} \leftarrow T^{-1}\mathbf{x}$, $\mathbf{x} \leftarrow T^{-T}\mathbf{x}$.

Poziom 3: operacje typu macierz-macierz

— mnożenie macierzy: $C \leftarrow \alpha AB + \beta C$, $C \leftarrow \alpha A^TB + \beta C$, $C \leftarrow \alpha AB^T + \beta C$, $C \leftarrow \alpha A^TB^T + \beta C$
— aktualizacje macierzy symetrycznych typu „rank-k" oraz „rank-$2k$": $C \leftarrow \alpha AA^T + \beta C$, $C \leftarrow \alpha A^TA + \beta C$, $C \leftarrow \alpha A^TB + \alpha B^TA + \beta C$, $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$,
— mnożenie macierzy przez macierz trójkątną: $B \leftarrow \alpha TB$, $B \leftarrow \alpha T^TB$, $B \leftarrow \alpha BT$, $B \leftarrow \alpha BT^T$,
— rozwiązywanie układów równań liniowych o macierzach trójkątnych z wieloma prawymi stronami: $B \leftarrow \alpha T^{-1}B$, $B \leftarrow \alpha T^{-T}B$, $B \leftarrow \alpha BT^{-1}$, $B \leftarrow \alpha BT^{-T}$.

Tabela 3.1 pokazuje zalety użycia wyższych poziomów BLAS-u [15]. Dla reprezentatywnych operacji z poszczególnych poziomów (kolumna 1) podaje liczbę odwołań do pamięci (kolumna 2), liczbę operacji arytmetycznych (kolumna 3) oraz ich stosunek (kolumna 4), przy założeniu że $m = n = k$. Im wyższy poziom BLAS-u, tym ten stosunek jest korzystniejszy, gdyż realizowana jest większa liczba operacji arytmetycznych na danych pobieranych z pamięci.

BLAS (operacja)		pamięć	operacje	stosunek
$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$	(AXPY)	$3n$	$2n$	$3:2$
$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$	(GEMV)	$mn + n + 2m$	$2m + 2mn$	$1:2$
$C \leftarrow \alpha AB + \beta C$	(GEMM)	$2mn + mk + kn$	$2mkn + 2mn$	$2:n$

Tabela 3.1. BLAS: odwołania do pamięci, liczba operacji arytmetycznych oraz ich stosunek, przy założeniu że $n = m = k$ [15]

3.2. Mnożenie macierzy przy wykorzystaniu różnych poziomów BLAS-u

Aby zilustrować wpływ tego faktu na szybkość obliczeń, rozważmy cztery równoważne sobie algorytmy mnożenia macierzy[1] postaci

$$C = \beta C + \alpha AB. \tag{3.1}$$

Zauważmy, że każdy wykonuje identyczną liczbę działań arytmetycznych. Algorytm 3.1, to klasyczne mnożenie macierzy, a algorytmy 3.2, 3.3, 3.4 wykorzystują odpowiednie operacje z kolejnych poziomów BLAS-u.

Algorytm 3.1. Sekwencyjne (skalarne) mnożenie macierzy.

Wejście: $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $\alpha, \beta \in \mathbb{R}$
Wyjście: $C = \beta C + \alpha AB$

```
1: for j = 1 to n do
2:    for i = 1 to m do
3:       t ← 0
4:       for l = 1 to k do
5:          t ← t + a_il b_lj
6:       end for
7:       c_ij ← βc_ij + αt
8:    end for
9: end for
```

Algorytm mnożenia macierzy może być łatwo zrównoleglony na poziomie operacji blokowych. Rozważmy operację (3.1). Dla uproszczenia przyjmijmy,

[1] Przy ich opisie oraz w dalszych rozdziałach wykorzystamy następujące oznaczenia. Niech $M \in \mathbb{R}^{m\times n}$, wówczas $M_{i:j,k:l}$ oznacza macierz powstającą z M jako wspólna część wierszy od i do j oraz kolumn od k do l.

Algorytm 3.2. Mnożenie macierzy przy użyciu podprogramów BLAS 1.

Wejście: $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $\alpha, \beta \in \mathbb{R}$
Wyjście: $C = \beta C + \alpha AB$

```
1: for j = 1 to n do
2:    C_{*j} ← βC_{*j} {operacja SCAL}
3:    for i = 1 to k do
4:       C_{*j} ← C_{*j} + (αb_{ij})A_{*i} {operacja AXPY}
5:    end for
6: end for
```

Algorytm 3.3. Mnożenie macierzy przy użyciu podprogramów BLAS 2.

Wejście: $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $\alpha, \beta \in \mathbb{R}$
Wyjście: $C = \beta C + \alpha AB$

```
1: for j = 1 to n do
2:    C_{*j} ← αAB_{*j} + βC_{*j} {operacja GEMV}
3: end for
```

że A, B, C są macierzami kwadratowymi o liczbie wierszy równej n. Zakładając, że n jest liczbą parzystą, możemy zapisać operację w następującej postaci blokowej

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} + \beta \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{3.2}$$

gdzie każdy blok A_{ij}, B_{ij}, C_{ij} jest macierzą kwadratową o liczbie wierszy równej $n/2$. Stąd wyznaczenie wyniku operacji może być rozłożone na osiem operacji takiej samej postaci jak (3.1).

$$\begin{array}{ll} C_{11} \leftarrow \alpha A_{11}B_{11} + \beta C_{11} & /1/ \\ C_{11} \leftarrow \alpha A_{12}B_{21} + C_{11} & /2/ \\ C_{12} \leftarrow \alpha A_{11}B_{12} + \beta C_{12} & /3/ \\ C_{12} \leftarrow \alpha A_{12}B_{22} + C_{11} & /4/ \\ C_{21} \leftarrow \alpha A_{22}B_{21} + \beta C_{21} & /5/ \\ C_{21} \leftarrow \alpha A_{21}B_{11} + C_{21} & /6/ \\ C_{22} \leftarrow \alpha A_{22}B_{22} + \beta C_{22} & /7/ \\ C_{22} \leftarrow \alpha A_{21}B_{12} + C_{22} & /8/ \end{array} \tag{3.3}$$

Zauważmy, że najpierw można równolegle wykonać operacje o numerach nieparzystych, a następnie (również równolegle) operacje o numerach parzystych. Opisane postępowanie pozwala na wyrażenie operacji (3.1) w terminach tej samej operacji. Liczbę podziałów według schematu (3.2) można dostosować do wielkości pamięci podręcznej.

Algorytm 3.4. Mnożenie macierzy przy użyciu podprogramów BLAS 3.
Wejście: $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $\alpha, \beta \in \mathbb{R}$
Wyjście: $C = \beta C + \alpha AB$
1: $C \leftarrow \beta C + \alpha AB$ {operacja `GEMM`}

	PIII 866MHz		P4 3GHz HT		Cray X1, 1 MSP	
alg.	Mflops	sec.	Mflops	sec.	Mflops	sec.
3.1	93.98	21.28	282.49	7.08	7542.29	0.27
3.2	94.65	21.13	1162.79	1.72	587.41	3.40
3.3	342.46	5.83	1418.43	1.40	7259.48	0.28
3.4	1398.60	1.43	7692.30	0.26	16369.89	0.12

Tabela 3.2. Wydajność i czas wykonania algorytmów mnożenia macierzy na różnych procesorach dla wartości $m = n = k = 1000$

Tabela 3.2 pokazuje czas działania i wydajność osiąganą przy wykonaniu poszczególnych algorytmów na trzech różnych typach starszych komputerów, przy czym $m = n = k = 1000$. W każdym przypadku widać, że algorytm wykorzystujący wyższy poziom BLAS-u jest istotnie szybszy. Co więcej, wykonanie algorytmu 3.4 odbywa się z wydajnością bliską teoretycznej maksymalnej. W przypadku procesorów Pentium czas wykonania każdego algorytmu wykorzystującego BLAS jest mniejszy niż czas wykonania algorytmu 3.1, który wykorzystuje jedynie kilka procent wydajności procesorów. W przypadku komputera Cray X1 czas wykonania algorytmu 3.1 utrzymuje się na poziomie czasu wykonania algorytmu wykorzystującego BLAS poziomu 2, co jest wynikiem doskonałej optymalizacji prostego kodu algorytmu 3.1, realizowanej przez kompilator Cray, który jest powszechnie uznawany za jeden z najlepszych kompilatorów optymalizujących. Zauważmy, że stosunkowo słabą optymalizację kodu przeprowadza kompilator Intela, a zatem w przypadku tych procesorów użycie podprogramów z wyższych poziomów biblioteki BLAS staje się koniecznością, jeśli chcemy pełniej wykorzystać moc oferowaną przez te procesory. Tabela 3.3 pokazuje wydajność, czas działania oraz przyspieszenie w sensie definicji 2.2 dla algorytmów 3.1, 3.2, 3.3 oraz 3.4 na dwuprocesorowym komputerze Quad-Core Xeon (łącznie 8 rdzeni). Można zauważyć, że algorytmy 3.1, 3.2 i 3.3 wykorzystują niewielki procent wydajności komputera, a dla algorytmu 3.4 osiągana jest bardzo duża wydajność. Wszystkie algorytmy dają się dobrze zrównoleglić, choć najlepsze przyspieszenie względne jest osiągnięte dla algorytmów 3.1 i 3.2. Zauważmy również, że dla algorytmów 3.1, 3.2 można zaobserwować przyspieszanie ponadliniowe (większe niż liczba użytych rdzeni). Taką

sytuację obserwuje się często, gdy w przypadku obliczeń równoległych, dane przetwarzane przez poszczególne procesory (rdzenie) lepiej wykorzystują pamięć podręczną [48]. Warto również zwrócić uwagę na ogromny wzrost wydajności w porównaniu z wynikami przedstawionymi w tabeli 3.2.

Zauważmy również, że podprogramy z biblioteki BLAS dowolnego poziomu mogą być łatwo zrównoleglone, przy czym ze względu na odpowiednio duży stopień lokalności danych najlepsze efekty daje zrównoleglenie podprogramów z poziomu 3 [10]. Biblioteka PBLAS (ang. *parallel BLAS*, [7]) zawiera podprogramy realizujące podstawowe operacje algebry liniowej w środowisku rozproszonym, wykorzystuje lokalnie BLAS oraz BLACS w warstwie komunikacyjnej.

3.3. Rozkład Cholesky'ego

Przedstawiony w poprzednim podrozdziale problem mnożenia macierzy należy do takich, które raczej łatwo poddają się optymalizacji. Dla kontrastu rozważmy teraz problem efektywnego zaprogramowania rozkładu Cholesky'ego [24] przy wykorzystaniu poszczególnych poziomów BLAS-u. Będzie to jednocześnie przykład ilustrujący metodę konstrukcji algorytmów blokowych dla zagadnień algebry liniowej. Niech

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \in \mathbb{R}^{n \times n}$$

będzie macierzą symetryczną ($a_{ij} = a_{ji}$) i dodatnio określoną (dla każdego $\mathbf{x} \neq \mathbf{0}$, $\mathbf{x}^T A\mathbf{x} > 0$). Istnieje wówczas macierz dolnotrójkątna L taka, że

$$LL^T = A.$$

Algorytm 3.5 jest prostym (sekwencyjnym) algorytmem wyznaczania komponentu rozkładu L.

Zauważmy, że pętle z linii 3–5 oraz 8–10 algorytmu 3.5 mogą być zapisane w postaci wektorowej. Otrzymamy wówczas algorytm wektorowy 3.6. Instrukcja 3 może być zrealizowana w postaci wywołania operacji AXPY, zaś instrukcja 6 jako wywołanie SCAL z pierwszego poziomu BLAS-u.

Pętla z linii 2–4 algorytmu 3.6 może być zapisana w postaci w postaci wywołania operacji GEMV (mnożenie macierz-wektor) z drugiego poziomu BLAS-u. Otrzymamy w ten sposób algorytm 3.7.

	1 core		Quad-Core			2x Quad-Core		
alg.	Mflops	czas (s)	Mflops	czas (s)	s_p	Mflops	czas (s)	s_p
3.1	350.9	5.6988	1435.0	1.3937	4.09	2783.4	0.7185	7.93
3.2	1573.8	1.2708	6502.0	0.3076	4.13	12446.1	0.1607	7.90
3.3	4195.5	0.4767	13690.1	0.1461	3.26	22326.9	0.0896	5.32
3.4	16026.3	0.1248	54094.9	0.0370	3.38	86673.5	0.0231	5.41

Tabela 3.3. Wydajność, czas wykonania i przyspieszenie względne algorytmów mnożenia macierzy na dwuprocesorowym komputerze Xeon Quad-Core dla wartości $m = n = k = 1000$

Algorytm 3.5 Sekwencyjna (skalarna) metoda Cholesky'ego

Wejście: $A \in \mathbb{R}^{n\times n}$,
Wyjście: $a_{ij} = l_{ij}$, $1 \le j \le i \le n$

for $j = 1$ to n **do**
  for $k = 1$ to $j-1$ **do**
    for $i = j$ to n **do**
      $a_{ij} \leftarrow a_{ij} - a_{jk}a_{ik}$
    end for
  end for
  $a_{jj} \leftarrow \sqrt{a_{jj}}$
  for $i = j+1$ to n **do**
    $a_{ij} \leftarrow a_{ij}/a_{jj}$
  end for
end for

Algorytm 3.6 Wektorowa metoda Cholesky'ego

Wejście: $A \in \mathbb{R}^{n\times n}$,
Wyjście: $a_{ij} = l_{ij}$, $1 \le j \le i \le n$

for $j = 1$ to n **do**
  for $k = 1$ to $j-1$ **do**
    $$\begin{pmatrix} a_{jj} \\ \vdots \\ a_{nj} \end{pmatrix} \leftarrow \begin{pmatrix} a_{jj} \\ \vdots \\ a_{nj} \end{pmatrix} - a_{jk} \begin{pmatrix} a_{jk} \\ \vdots \\ a_{nk} \end{pmatrix}$$
  end for
  $a_{jj} \leftarrow \sqrt{a_{jj}}$
  $$\begin{pmatrix} a_{j+1,j} \\ \vdots \\ a_{nj} \end{pmatrix} \leftarrow \frac{1}{a_{jj}} \begin{pmatrix} a_{j+1,j} \\ \vdots \\ a_{nj} \end{pmatrix}$$
end for

Algorytm 3.7 Wektorowo-macierzowa metoda Cholesky'ego

Wejście: $A \in \mathbb{R}^{n\times n}$,

Wyjście: $a_{ij} = l_{ij}$, $1 \le j \le i \le n$

1: **for** $j = 1$ to n **do**

2: $$\begin{pmatrix} a_{jj} \\ \vdots \\ a_{nj} \end{pmatrix} \leftarrow \begin{pmatrix} a_{jj} \\ \vdots \\ a_{nj} \end{pmatrix} - \begin{pmatrix} a_{j1} & \dots & a_{j,j-1} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{n,j-1} \end{pmatrix} \times \begin{pmatrix} a_{j1} \\ \vdots \\ a_{j,j-1} \end{pmatrix}$$

3: $a_{jj} \leftarrow \sqrt{a_{jj}}$

4: $$\begin{pmatrix} a_{j+1,j} \\ \vdots \\ a_{nj} \end{pmatrix} \leftarrow \frac{1}{a_{jj}} \begin{pmatrix} a_{j+1,j} \\ \vdots \\ a_{nj} \end{pmatrix}$$

5: **end for**

Aby wyrazić rozkład Cholesky'ego w postaci wywołań operacji z trzeciego poziomu BLAS-u, zapiszmy rozkład w następującej postaci blokowej:[2]

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{pmatrix}.$$

Stąd po dokonaniu mnożenia odpowiednich bloków otrzymamy następującą postać macierzy A:

$$A = \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T & L_{11}L_{31}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T & L_{21}L_{31}^T + L_{22}L_{32}^T \\ L_{31}L_{11}^T & L_{31}L_{21}^T + L_{32}L_{22}^T & L_{31}L_{31}^T + L_{32}L_{32}^T + L_{33}L_{33}^T \end{pmatrix}.$$

Przyjmijmy, że został już zrealizowany pierwszy krok blokowej metody i powstał następujący rozkład:

$$A = \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & 0 & I \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ 0 & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{pmatrix},$$

co uzyskujemy dokonując najpierw rozkładu $A_{11} = L_{11}L_{11}^T$, a następnie rozwiązując układ równań postaci

$$\begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} L_{11}^T = \begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix}.$$

[2] Dla prostoty przyjmujemy, że macierz można zapisać w postaci dziewięciu bloków – macierzy kwadratowych o takiej samej liczbie wierszy.

W kolejnym kroku rozpoczynamy od aktualizacji pozostałych bloków przy pomocy bloków już wyznaczonych. Otrzymamy w ten sposób:

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} \leftarrow \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} L_{21}^T$$

oraz

$$\begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \leftarrow \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} L_{31}^T.$$

Łącząc powyższe operacje w jedną, otrzymujemy

$$\begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} \leftarrow \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \begin{pmatrix} L_{21}^T & L_{31}^T \end{pmatrix}.$$

Następnie dokonujemy rozkładu $A_{22} = L_{22}L_{22}^T$, po czym dokonujemy aktualizacji rozwiązując układ równań liniowych o macierzy dolnotrójkątnej

$$A_{32} = L_{32}L_{22}^T.$$

Otrzymujemy wówczas następujący rozkład

$$A = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & I \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ 0 & L_{22}^T & L_{32}^T \\ 0 & 0 & A_{33} \end{pmatrix}.$$

W kolejnych krokach w analogiczny sposób zajmujemy się rozkładem bloku A_{33}. Niech teraz macierz A będzie dana w postaci blokowej

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1p} \\ A_{21} & A_{22} & \dots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pp} \end{pmatrix},$$

gdzie $A_{ij} = A_{ji}^T$ oraz $A_{ij} \in \mathbb{R}^{n/p \times n/p}$ dla $1 \leq i, j \leq p$. Algorytm 3.8 dokonuje rozkładu Cholesky'ego przy wykorzystaniu BLAS-u poziomu trzeciego.

Trzeba podkreślić, że niewątpliwą zaletą algorytmu 3.8 jest możliwość prostego zastosowania nowych formatów reprezentacji macierzy (podrozdział 1.2.3). Istotnie, każdy blok A_{ij} może zajmować zwarty obszar pamięci operacyjnej.

Table 3.4–3.8 pokazują czas działania oraz wydajność komputera *Dual Xeon Quadcore 3.2 GHz* dla algorytmów wykorzystujących poszczególne poziomy BLAS-u (Alg. 1, 2 oraz 3) oraz algorytmu 3.8 wykorzystującego format reprezentacji macierzy przedstawiony w sekcji 1.2.3 (Alg. T). Dla algorytmów Alg. 1 oraz Alg. 2 nie zastosowano równoległości, Alg. 3 to jedno

Algorytm 3.8 Macierzowa (blokowa) metoda Cholesky'ego

Wejście: $A \in \mathbb{R}^{n\times n}$,
Wyjście: $a_{ij} = l_{ij}$, $1 \le j \le i \le n$

1: **for** $j = 1$ to p **do**
2: $A_{jj} \leftarrow \mathrm{chol}(A_{jj})$
3: **if** $j < p$ **then**
4: $$\begin{pmatrix} A_{j+1,j} \\ \vdots \\ A_{pj} \end{pmatrix} \leftarrow \begin{pmatrix} A_{j+1,j} \\ \vdots \\ A_{pj} \end{pmatrix} \times A_{jj}^{-T}$$
5: $$\begin{pmatrix} A_{j+1,j+1} & \dots & A_{j+1,p} \\ \vdots & & \vdots \\ A_{p,j+1} & \dots & A_{pp} \end{pmatrix} \leftarrow \begin{pmatrix} A_{j+1,j+1} & \dots & A_{j+1,p} \\ \vdots & & \vdots \\ A_{p,j+1} & \dots & A_{pp} \end{pmatrix} - \begin{pmatrix} A_{j+1,j} \\ \vdots \\ A_{pj} \end{pmatrix} \times \begin{pmatrix} A_{j+1,j}^T \dots A_{pj}^T \end{pmatrix}$$
6: **end if**
7: **end for**

	Quad-core		2x Quad-core	
alg.	Mflops	sec.	Mflops	sec.
Alg.1	3573.77	0.10	3573.77	0.10
Alg.2	3261.17	0.11	3261.17	0.11
Alg.3	40164.80	8.91E-3	57774.00	6.19E-3
Alg.T	43547.12	8.22E-3	47456.76	7.54E-3

Tabela 3.4. Czas działania i wydajność algorytmów dla $n = 1024$

wywołanie podprogramu `spotrf` z biblioteki Intel MKL (wersja równoległa). Alg. T został zrównoleglony przy pomocy OpenMP.[3] Dla większych rozmiarów danych ($n = 16384$, $n = 32768$) podano wyniki jedynie dla Alg. 3 oraz Alg. T. Czas wykonania pozostałych jest rzędu kilku godzin, co w praktyce czyni je bezużytecznymi. Zauważmy, że dla większych rozmiarów problemu użycie nowych sposobów reprezentacji macierzy w zauważalny sposób skraca czas obliczeń.

[3] Wykorzystano standard OpenMP, który omawiamy szczegółowo w rozdziale 4.

	Quad-core		2x Quad-core	
alg.	Mflops	sec.	Mflops	sec.
Alg.1	1466.16	15.62	1466.16	15.62
Alg.2	1364.54	16.78	1364.54	16.78
Alg.3	67010.55	0.34	125835.50	0.18
Alg.T	72725.28	0.32	128566.94	0.20

Tabela 3.5. Czas działania i wydajność algorytmów dla $n = 4096$

	Quad-core		2x Quad-core	
alg.	Mflops	sec.	Mflops	sec.
Alg.3	72800.40	2.51	134151.73	1.36
Alg.T	81660.70	2.24	147904.53	1.23

Tabela 3.6. Czas działania i wydajność algorytmów dla $n = 8192$

	Quad-core		2x Quad-core	
alg.	Mflops	sec.	Mflops	sec.
Alg.3	77349.85	18.95	145749.74	10.05
Alg.T	84852.68	17.27	163312.97	8.97

Tabela 3.7. Czas działania i wydajność algorytmów dla $n = 16384$

	Quad-core		2x Quad-core	
alg.	Mflops	sec.	Mflops	sec.
Alg.3	80094.68	146.43	146194.12	80.19
Alg.T	85876.31	136.57	168920.31	69.40

Tabela 3.8. Czas działania i wydajność algorytmów dla $n = 32768$

3.4. Praktyczne użycie biblioteki BLAS

Biblioteka BLAS została zaprojektowana z myślą o języku Fortran. Nagłówki przykładowych podprogramów z poszczególnych poszczególnych poziomów przedstawia listing 3.1. Na poziomie pierwszym (działania na wektorach) poszczególne operacje są opisane przy pomocy fortranowskich podprogramów (czyli `SUBROUTINE`) oraz funkcji (czyli `FUNCTION`). W nazwach pierwszy znak określa typ danych, na których działa operacja (`S`, `D` dla obliczeń na liczbach rzeczywistych odpowiednio pojedynczej i podwójnej precyzji oraz `C`, `Z` dla obliczeń na liczbach zespolonych pojedynczej i podwójnej precyzji). Wyjątek stanowi operacja `I_AMAX` znajdowania numeru składowej wektora, która jest największa co do modułu. W nazwie, w miejsce znaku „_", należy wstawić jedną z liter określających typ składowych wektora. Przykładowo `SAXPY` oraz `DAXPY` to podprogramy realizujące operację AXPY. Wektory są opisywane parametrami tablicowymi (przykładowo `SX` oraz `SY` w podprogramie `SAXPY` na listingu 3.1) oraz liczbami całkowitymi, które wskazują co ile danych w pamięci znajdują się kolejne składowe wektora (parametry `INCX` oraz `INCY` na listingu 3.1).

Na poziomie 2 dochodzą opisy tablic przechowujących macierze, które są przechowywane kolumnami (jak pokazano na rysunku 1.3), stąd parametr `LDA` określa liczbę wierszy w tablicy przechowującej macierz. Parametr `TRANS` równy `'T'` lub `'N'` mówi, czy należy zastosować transpozycję macierzy. Parametr `UPLO` równy `'L'` lub `'U'` określa, czy macierz jest dolnotrójkątna, czy górnotrójkątna. Parametr `DIAG` równy `'U'` lub `'N'` mówi, czy macierz ma na głównej przekątnej jedynki, czy też elementy różne od jedności. Na poziomie 3 dochodzi parametr `SIDE` równy `'L'` (left) lub `'R'` (right) oznaczający operację lewostronną lub prawostronną.

Listing 3.1. Przykładowe nagłówki w języku Fortran

```
1  ******************  Poziom 1  ****************
2  *
3        SUBROUTINE SAXPY(N,SA,SX,INCX,SY,INCY)
4  *     .. parametry skalarne ..
5        REAL SA
6        INTEGER INCX,INCY,N
7  *     ..
8  *     .. parametry tablicowe ..
9        REAL SX(*),SY(*)
10 *
11 *     Wyznacza sy, gdzie sy:=sy+sa*sx
12 *
13 ***********************************************
14       REAL FUNCTION SDOT(N,SX,INCX,SY,INCY)
15 *     .. parametry skalarne ..
16       INTEGER INCX,INCY,N
```

```
17 *      ..
18 *      .. parametry tablicowe ..
19        REAL SX(*),SY(*)
20 *
21 *      Zwraca sx'*sy  (tzw. dot product)
22 *
23 ********************  Poziom 2  ****************
24 *
25        SUBROUTINE SGEMV(TRANS,M,N,ALPHA,A,LDA,X,INCX,
26                                                BETA,Y,INCY)
27 *      .. parametry skalarne ..
28        REAL ALPHA,BETA
29        INTEGER INCX,INCY,LDA,M,N
30        CHARACTER TRANS
31 *      ..
32 *      .. parametry tablicowe ..
33        REAL A(LDA,*),X(*),Y(*)
34 *
35 *      Wyznacza y, gdzie y := alpha*A*x + beta*y
36 *                  lub    y := alpha*A'*x + beta*y
37 *
38 ***********************************************
39        SUBROUTINE STRSV(UPLO,TRANS,DIAG,N,A,LDA,X,INCX)
40 *      .. parametry skalarne ..
41        INTEGER INCX,LDA,N
42        CHARACTER DIAG,TRANS,UPLO
43 *      ..
44 *      .. parametry tablicowe ..
45        REAL A(LDA,*),X(*)
46 *
47 *      Wyznacza x, gdzie A*x = b lub A'*x = b
48 *
49 ********************  Poziom 3  ****************
50 *
51        SUBROUTINE SGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,
52                                          B,LDB,BETA,C,LDC)
53 *      .. parametry tablicowe ..
54        REAL ALPHA,BETA
55        INTEGER K,LDA,LDB,LDC,M,N
56        CHARACTER TRANSA,TRANSB
57 *      ..
58 *      .. parametry tablicowe ..
59        REAL A(LDA,*),B(LDB,*),C(LDC,*)
60 *
61 *      Wyznacza
62 *
63 *      C := alpha*op( A )*op( B ) + beta*C,
64 *
65 *      gdzie  op( X ) to op( X ) = X  lub  op( X ) = X'
66 *
67 ***********************************************
```

```
68        SUBROUTINE STRSM(SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,
69                                                  A,LDA,B,LDB)
70  *     .. parametry skalarne ..
71        REAL ALPHA
72        INTEGER LDA,LDB,M,N
73        CHARACTER DIAG,SIDE,TRANSA,UPLO
74  *     ..
75  *     .. parametry tablicowe ..
76        REAL A(LDA,*),B(LDB,*)
77  *     ..
78  *     Wyznacza X, gdzie
79  *
80  *     op( A )*X = alpha*B   lub   X*op( A ) = alpha*B
81  *
82  *     oraz
83  *
84  *     op( A ) = A    or    op( A ) = A'
85  *
86  *     przy czym B:=X
87  ********************************************
```

Istnieje możliwość łatwego użycia podprogramów BLAS-u z poziomu języka C. Trzeba w takim przypadku przyjąć następujące założenia:
— podprogramy `subroutine` należy traktować jako funkcje `void`,
— w języku Fortran parametry są przekazywane jako wskaźniki,
— macierze są przechowywane kolumnami, zatem parametr `LDA` określa liczbę wierszy w tablicy przechowującej macierz.
Listing 3.2 przedstawia odpowiedniki nagłówków z listingu 3.1 dla języka C.

Listing 3.2. Przykładowe nagłówki w języku C

```
1  void SAXPY(const int *n, const float *alpha, const float *x,
2             const int *incx, float *y, const int *incy);
3
4  float SDOT(const int *n, const float *x, const int *incx,
5             const float *y, const int *incy);
6
7  void SGEMV(const char *trans, const int *m, const int *n,
8             const float *alpha, const float *a,
9             const int *lda, const float *x, const int *incx,
10            const float *beta, float *y, const int *incy);
11
12 void STRSV(const char *uplo, const char *trans,
13            const char *diag, const int *n, const float *a,
14            const int *lda, float *x, const int *incx);
15
16 void SGEMM(const char *transa, const char *transb,
17            const int *m, const int *n, const int *k,
18            const float *alpha, const float *a,
```

```
19              const int *lda, const float *b, const int *ldb,
20              const float *beta, float *c, const int *ldc);
21
22 void STRSM(const char *side, const char *uplo,
23              const char *transa, const char *diag,
24              const int *m, const int *n, const float *alpha,
25              const float *a, const int *lda, float *b,
26              const int *ldb);
```

Na listingu 3.3 przedstawiamy przykład użycia podprogramów z biblioteki BLAS (poziom 1 i 2) dla rozwiązywania układów równań liniowych eliminacją Gaussa.

Listing 3.3. Rozwiązywanie układów równań liniowych eliminacją Gaussa przy użyciu biblioteki BLAS

```
1  void elim_gaussa(int n,double *a, int lda,
2                   double tol, int *info){
3  /* n    - liczba niewiadomych
4     a    - macierz o n wierszach i n+1 kolumnach
5            ostatnia kolumna to prawa strona układu
6     lda  - liczba wierszy w tablicy
7     tol  - tolerancja (bliska zeru)
8     info  == 0 - obliczono rozwiązanie
9           != 0 - macierz osobliwa
10 */
11   int i,j,k, incx=1;
12   *info=0; k=0;
13
14   while((k<n-1)&&(*info==0)){
15
16     int dv=n-k;    // wyznaczenie wiersza głównego
17     int piv=k+IDAMAX(&dv,&a[k*lda+k],&incx)-1;
18
19     if(abs(a[k*lda+piv])<tol){ // macierz osobliwa
20       *info=k+1;
21     }else{
22       if (k!=piv) { // wymiana wierszy "k" i "piv"
23         int n1=n+1;
24         DSWAP(&n1,&a[k],&lda,&a[piv],&lda);
25       }
26       for(i=k+1;i<n;i++){
27         // aktualizacja wierszy k+1,...,n-1
28         double alfa=-a[k*lda+i]/a[k*lda+k];
29         DAXPY(&dv,&alfa,&a[(k+1)*lda+k],
30               &lda,&a[(k+1)*lda+i],&lda);
31       }
32     }
33     k++;
34   }
```

```
35
36    if (abs(a[(n-1)*lda+n-1])<tol) // macierz osobliwa
37        *info=n;
38
39    if(*info==0){ // odwrotna eliminacja
40      char uplo='U';
41      char trans='N';
42      char diag='N';
43      DTRSV(&uplo,&trans,&diag,&n,a,&lda,&a[n*lda],&incx);
44    }
45  }
```

3.5. LAPACK

Biblioteka LAPACK (ang. *Linear Algebra Package*) jest zbiorem podprogramów do rozwiązywania układów równań liniowych oraz algebraicznego zagadnienia własnego. Pełny opis znajduje się w książce [3]. Listing 3.4 zamieszczamy przykładowy kod do rozwiązywania układów równań liniowych przy pomocy eliminacji Gaussa.

Listing 3.4. Rozwiązywanie układów równań liniowych przy użyciu biblioteki LAPACK

```
1  #include "mkl.h"
2  #include "omp.h"
3  #define MAXN 4001
4
5  int main() {
6
7    int n=2000,    // n - liczba niewiadomych
8    lda=MAXN,      // lda - wiodący rozmiar tablicy
9    info=0;        // info - o wyniku (==0 - jest rozwiązanie
10                  //                  !=0 - brak rozwiązania)
11   double t;      // t - do mierzenia czasu
12   double *a,     // a - tablica - macierz i prawa strona
13          *x;     // x - rozwiązanie
14
15   // alokujemy pamięć na tablice a i wektor x
16   a=malloc(MAXN*MAXN*sizeof *a);
17   x=malloc(MAXN*sizeof *a);
18   int *ipiv;
19   ipiv=malloc(MAXN*sizeof *ipiv);
20   int nrhs=1;
21
22   // generujemy dane wejściowe
23   ustawdane(n,a,lda);
24
```

```
25    // rozwiązywanie układu eliminacja Gaussa
26    DGESV(&n,&nrhs,a,&lda,ipiv,&a[n*lda],&lda,&info);
27
28    // przetwarzanie wyników
29    // ........
30 }
```

Biblioteki BLAS i LAPACK są dostępne na większość architektur komputerowych. Istnieje również możliwość skompilowania ich źródeł. Na szczególną uwagę zasługuje biblioteka Atlas (ang. *Automatically Tuned Linear Algebra Software* [65]), która dobrze wykorzystuje właściwości współczesnych procesorów ogólnego przeznaczenia (CPU).[4] Na platformy komputerowe oparte na procesorach Intela ciekawą (komercyjną) propozycję stanowi biblioteka MKL (ang. *Math Kernel Library*[5]). Zawiera ona między innymi całą bibliotekę BLAS oraz LAPACK. Na rysunku 3.5 podajemy przykładowy plik `Makefile`, który może posłużyć do kompilacji i łączenia programów wykorzystujących bibliotekę MKL.

```
MKLPATH = ${MKLROOT}/lib/em64t
FILES   = mkl05ok.c
PROG    = mkl05ok
LDLIBS  = -L${MKLPATH} -Wl,--start-group \
   ${MKLPATH}/libmkl_intel_lp64.a \
   ${MKLPATH}/libmkl_intel_thread.a \
   ${MKLPATH}/libmkl_core.a -Wl,--end-group
OPTFLG  = -O3 -xT -openmp  -static
all:
        icc $(OPTFLG) -o $(PROG) $(FILES) $(LDLIBS)
clean:
        rm -f core *.o
```

Rysunek 3.1. Przykładowy plik `Makefile` z użyciem biblioteki MKL

[4] Kod źródłowy jest dostępny na stronie `http://www.netlib.org/atlas`

[5] Więcej informacji na stronie `http://software.intel.com/en-us/articles/intel-mkl/`

3.6. Zadania

Poniżej zamieściliśmy kilka zadań do samodzielnego wykonania. Ich celem jest utrwalenie umiejętności posługiwania się biblioteką BLAS.

Zadanie 3.1.
Napisz program obliczający iloczyn skalarny dwóch wektorów liczb rzeczywistych. Do wyznaczenia iloczynu wykorzystaj funkcję biblioteki BLAS poziomu 1 (`DDOT`).

Zadanie 3.2.
Mając dane dwie tablice **a** i **b** elementów typu rzeczywistego, napisz program liczący iloczyn skalarny dwóch wektorów **x** i **y** utworzonych z elementów tablic **a** i **b** w następujący sposób. Wektor **x** składa się z 30 pierwszych elementów tablicy **a** o indeksach nieparzystych, a wektor **y** z 30 elementów tablicy **b** o indeksach podzielnych przez 3, począwszy od indeksu 30.

Zadanie 3.3.
Napisz program wykonujący operację AXPY (uogólniona operacja dodawania wektorów) dla dwóch wektorów liczb rzeczywistych **x** i **y**. Wykorzystaj funkcję biblioteki BLAS poziomu 1 (`DAXPY`).

Zadanie 3.4.
Napisz program, w którym wszystkie elementy jednego wektora oraz co trzeci element począwszy od szóstego drugiego wektora przemnożysz o liczbę 3. Wykorzystaj funkcję biblioteki BLAS poziomu 1 (`DSCAL`).

Zadanie 3.5.
Napisz program, w którym utworzysz wektor **y** zawierający wszystkie elementy wektora **x** o indeksach nieparzystych. Wykorzystaj funkcję biblioteki BLAS poziomu 1 (`DCOPY`).

Zadanie 3.6.
Napisz program, wykonujący operację uogólnionego mnożenia macierzy przez wektor. Wykorzystaj funkcję BLAS poziomu 2 (`DGEMV`).

Zadanie 3.7.
Opisz w postaci funkcji, algorytmy wykonujące mnożenie macierzy. Macierze należy pomnożyć za pomocą następujących metod:
1. wykorzystując funkcję biblioteki BLAS poziomu 1 (`DDOT`),
2. wykorzystując funkcję biblioteki BLAS poziomu 2 (`DGEMV`),
3. wykorzystując funkcję biblioteki BLAS poziomu 3 (`DGEMM`).

Rozdział 4

Programowanie w OpenMP

OpenMP to standard programowania komputerów z pamięcią wspólną dla języków C/C++ oraz Fortran [6,56]. Umożliwia on zawarcie w programie komputerowym trzech kluczowych aspektów programu równoległego, czyli

1. specyfikacji równoległego wykonywania obliczeń;
2. mechanizmów komunikacji pomiędzy poszczególnymi wątkami;
3. synchronizacji pracy wątków.

W standardzie OpenMP realizuje się powyższe aspekty przy użyciu kilku dyrektyw (w językach C oraz C++ są to *pragmy*, zaś w języku Fortran komentarze specjalne) dla kompilatora oraz niewielkiego zbioru funkcji. Umożliwia to kompilację programu również przy użyciu kompilatorów, które nie wspierają OpenMP, a zatem możliwe jest takie skonstruowanie programu, aby kod działał również na tradycyjnych systemach jednoprocesorowych.

4.1. Model wykonania programu

Model wykonania programu w OpenMP jest następujący. Program rozpoczyna się realizacją instrukcji pojedynczego wątku głównego (ang. *main thread*). Następnie, gdy wystąpi konstrukcja specyfikująca region równoległy, wówczas tworzona jest grupa działających równolegle wątków i jednym z nich jest wątek główny. W przypadku natrafienia przez pewien wątek na kolejny region równoległy, jego wykonanie przebiega sekwencyjnie (chyba, że odpowiednio wyspecyfikowanie zagnieżdżanie wątków). Na końcu regionu równoległego występuje niejawna bariera. Po jej osiągnięciu wątek główny kontynuuje pracę sekwencyjną.

4.2. Ogólna postać dyrektyw

Poniżej przedstawiamy ogólny schemat dyrektyw OpenMP dla języków C oraz C++. W książce [6] przedstawiono szczegółowo składnię OpenMP dla języka Fortran.

```
1 #pragma omp dyrektywa klauzula ... klauzula
2     instrukcja
```

Dyrektywę OpenMP wraz z następującą bezpośrednio po niej instrukcją będziemy nazywać *konstrukcją OpenMP*. Opcjonalne klauzule specyfikują dodatkowe szczegóły dotyczące danej konstrukcji.

Pragmy **omp** są ignorowane przez kompilatory, które nie wspierają standardu OpenMP. Ukrycie wywołań funkcji OpenMP wymaga użycia dyrektyw warunkowej kompilacji. Umożliwia ona zaakceptowanie kodu przez każdy kompilator. Ilustruje to poniższy przykład.

```
#ifdef _OPENMP
      iam=omp_get_thread_num(); // specyficzne instrukcje
      ...                       // OpenMP
#endif
```

4.3. Specyfikacja równoległości obliczeń

Przedstawimy teraz konstrukcje OpenMP służące do specyfikacji potencjalnej równoległości obliczeń.

4.3.1. Konstrukcja *parallel*

Podstawowa konstrukcja OpenMP jest oparta na dyrektywie **parallel** oraz następującym bezpośrednio po niej bloku strukturalnym, czyli instrukcją (na ogół złożoną), która ma jedno wejście i jedno wyjście. Ogólna postać tej konstrukcji jest następująca:

```
#pragma omp parallel klauzule
  { // blok strukturalny
    ...
  }
```

W dyrektywie **parallel** mogą się pojawić następujące klauzule:
— **if**(*wyrażenie skalarne*)
— **num_threads**(*wyrażenie skalarne*)
— **private**(*lista zmiennych*)
— **firstprivate**(*lista zmiennych*)
— **shared**(*lista zmiennych*)
— **default**(**shared** albo **none**)
— **copyin**(*lista zmiennych*)
— **reduction**(*operator* **:** *lista zmiennych*)

Wykonanie konstrukcji **parallel** przebiega następująco. Gdy wątek *master* osiągnie miejsce określone przez dyrektywę **parallel**, wówczas tworzona jest grupa wątków pod warunkiem, że
1. nie występuje klauzula **if**,
2. wyrażenie w klauzuli **if** ma wartość różną od zera.

Gdy nie wystąpi sytuacja opisana wyżej (punkty 1, 2), wówczas blok strukturalny jest wykonywany przez wątek główny. W grupie wątków, *master* staje się wątkiem o numerze 0, a numeracja nie zmienia się w trakcie wykonania. Liczba wątków w grupie zależy od zmiennej środowiskowej `OMP_NUM_THREADS`, wartości wyrażenia w klauzuli **num_threads** oraz wywołania poniższej funkcji.

```
1 // ustawienie liczby wątków
2     void omp_set_num_threads(int num)
```

Każdy wątek wykonuje instrukcję opisaną blokiem strukturalnym. Po konstrukcji **parallel** występuje niejawna bariera. Po zakończeniu wykonywania instrukcji bloku przez wszystkie wątki, dalsze instrukcje wykonuje sekwencyjnie wątek główny. Ilustruje to poniższy przykład.

Listing 4.1. Użycie dyrektywy parallel

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main(){
4   int iam, np;
5   printf("begin\n");
6   #pragma omp parallel private(np,iam)
7   {
8       iam=omp_get_thread_num();
9       np=omp_get_num_threads();
10      printf("Hello from %d of %d\n",iam,np);
11  }
12  printf("end\n");
13 }
```

Funkcje wywoływane w liniach 8 i 9 zwracają odpowiednio numer wątku oraz liczbę wszystkich wątków.

4.3.2. Zasięg zmiennych

Pozostałe klauzule definiują zasięg zmiennych. Dotyczą one tylko bloku strukturalnego występującego bezpośrednio po dyrektywie. Domyślnie, jeśli zmienna jest widoczna wewnątrz bloku i nie została wyspecyfikowana w jednej z klauzul typu „private", wówczas jest wspólna dla wszystkich wątków w grupie. Zmienne automatyczne (deklarowane w bloku) są prywatne. W szczególności

— **private(***lista***)**: zmienne na liście są prywatnymi zmiennymi każdego wątku (obiekty są automatycznie alokowane dla każdego wątku),
— **firstprivate(***lista***)**: jak wyżej, ale dodatkowo w każdym wątku zmienne są inicjowane wartościami z wątku głównego,
— **shared(***lista***)**: zmienne z listy są wspólne dla wszystkich wątków w grupie,
— **default(shared)**: wszystkie zmienne domyślnie są wspólne (o ile nie znajdują się na żadnej liście typu „private"),
— **default(none)**: dla wszystkich zmiennych trzeba określić, czy są wspólne, czy też prywatne,

— **reduction(***operator* **:** *lista***)**: dla zmiennych z listy jest wykonywana operacja redukcyjna określona przez *operator*.

Listing 4.2 ilustruje zastosowanie klauzuli **reduction** do numerycznego wyznaczenia wartości całki ze wzoru

$$\int_a^b f(x)dx = \sum_{k=0}^{p-1} \int_{x_i}^{x_{i+1}} f(x)dx,$$

gdzie $x_i = a + ih$, $i = 0, \ldots, p$, $h = (b-a)/p$ oraz

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx h\frac{f(x_i) + f(x_{i+1})}{2}.$$

Listing 4.2. Obliczanie całki

```
#include <stdio.h>
#include <omp.h>

float f(float x){
  return sin(x);
}

int main(){
  int iam, np;
  float a=0;
  float b=1;
  float s;
  #pragma omp parallel private(np,iam) shared(a,b) \
                                          reduction(+:s)
  {
    iam=omp_get_thread_num();
    np=omp_get_num_threads();
    float xa,xb,h;
    h=(b-a)/np;
    xa=a+iam*h;
    xb=xa+h;
    s=0.5*h*(f(xa)+f(xb));
  }
  printf("Calka=%f\n",s);
}
```

Dopuszczalnymi operatorami redukcyjnymi są

```
// dopuszczalne operatory redukcyjne
   + - * & | ^ && ||
```

Typ zmiennych musi być odpowiedni dla operatora. Zmienne występujące na listach w klauzulach **reduction** nie mogą być na listach **shared** ani **private**.

4.4. Konstrukcje dzielenia pracy

Podamy teraz konstrukcje dzielenia pracy (ang. *work-sharing*), które znacznie ułatwiają programowanie. Wszystkie należy umieszczać w bloku strukturalnym po **parallel**.

4.4.1. Konstrukcja *for*

Konstrukcja **for** umożliwia dzielenie wykonań refrenu pętli **for** między wątki w grupie, przy czym każdy obrót będzie wykonany tylko raz. Przyjmuje ona następującą postać.

```
1  #pragma omp parallel ...
2  {
3     ...
4     #pragma omp for ....
5     for (...;...;...){
6        ...
7     }
8     ...
9  }
```

Wymaga się, aby pętla **for** była w postaci „kanonicznej", co oznacza, że przed jej wykonaniem musi być znana liczba obrotów. Dopuszczalne klauzule to:

— **private**(*lista zmiennych*)
— **firstprivate**(*lista zmiennych*)
— **reduction**(*operator* **:** *lista zmiennych*)
— **lastprivate**(*lista zmiennych*)
— **nowait**
— **ordered**
— **schedule**(*rodzaj*,*rozmiar*)

Rola pierwszych trzech jest analogiczna do roli w dyrektywie **parallel**. Klauzula **lastprivate** działa jak **private**, ale po zakończeniu wykonywania pętli zmienne mają taką wartość, jak przy sekwencyjnym wykonaniu pętli.

Po pętli **for** jest niejawna bariera, którą można znieść umieszczając klauzulę **nowait**. Ilustruje to następujący przykład.

```
1  #pragma omp parallel shared(n,...)
2  {
3     int i;
4     #pragma omp for nowait ...
5     for (i=0;i<n;i++){
6        ...
7     }
8     #pragma omp for ...
```

```
9       for (i=0;i<n;i++){
10        ...
11      }
12    }
```

Wątek, który zakończy wykonywanie przydzielonych mu obrotów pierwszej pętli, przejdzie do wykonania jego obrotów drugiej pętli bez oczekiwania na pozostałe wątki w grupie.

Klauzula **ordered** umożliwia realizację wybranej części refrenu pętli **for** w takim porządku, jakby pętla była wykonywana sekwencyjnie. Wymaga to użycia konstrukcji **ordered**. Ilustruje to następujący przykład.

```
1   int iam;
2   #pragma omp parallel private(iam)
3   {
4     iam=omp_get_thread_num();
5     int i;
6     #pragma omp for ordered
7     for (i=0;i<32;i++){
8       ... // instrukcje obliczeniowo "intensywne"
9       #pragma omp ordered
10      printf("Wątek %d wykonuje %d\n",iam,i);
11    }
12  }
```

Klauzula **schedule** specyfikuje sposób podziału wykonań refrenu pętli między wątki. Decyduje o tym parametr *rodzaj*. Możliwe są następujące przypadki.

— **static** – gdy przyjmuje postać **schedule(static,***rozmiar***)**, wówczas pula obrotów jest dzielona na kawałki o wielkości *rozmiar*, które są cyklicznie przydzielane do wątków; gdy nie poda się rozmiaru, to pula obrotów jest dzielona na mniej więcej równe części;
— **dynamic** – jak wyżej, ale przydział jest dynamiczny; gdy wątek jest wolny, wówczas dostaje kolejny kawałek; pusty rozmiar oznacza wartość 1;
— **guided** – jak **dynamic**, ale rozmiary kawałków maleją wykładniczo, aż liczba obrotów w kawałku będzie mniejsza niż *rozmiar*;
— **runtime** – przydział będzie wybrany w czasie wykonania programu na podstawie wartości zmiennej środowiskowej `OMP_SCHEDULE`, co pozwala na użycie wybranego sposobu szeregowania w trakcie wykonania programu, bez konieczności ponownej jego kompilacji.

Domyślny rodzaj zależy od implementacji. Dodajmy, że w pętli nie może wystąpić instrukcja **break**, zmienna sterująca musi być typu całkowitego. Dyrektywy **schedule**, **ordered** oraz **nowait** mogą wystąpić tylko raz. Poniżej zamieszczamy przykład użycia **schedule**.

```
1  int iam, i;
2  #pragma omp parallel private(iam, i)
3  {
4    iam=omp_get_thread_num();
5    #pragma omp for schedule(static,2)
6    for (i=0;i<32;i++){
7      printf("Wątek %d wykonuje %d\n",iam, i);
8    }
9  }
```

4.4.2. Konstrukcja *sections*

Konstrukcja **sections** służy do dzielenia pracy, której nie da się opisać w postaci pętli **for**. Ogólna postać jest następująca.

```
1  #pragma omp parallel ....
2   {
3  ...
4    #pragma omp sections ....
5     {
6      #pragma omp section
7       { // blok strukturalny 1
8         ...
9       }
10     #pragma omp section
11      { // blok strukturalny 2
12        ...
13      }
14     #pragma omp section
15      { // blok strukturalny 3
16        ...
17      }
18    ...
19  }
20   ...
21 }
```

Dopuszczalne klauzule to
— **private**(*lista zmiennych*)
— **firstprivate**(*lista zmiennych*)
— **reduction**(*operator* **:** *lista zmiennych*)
— **lastprivate**(*lista zmiennych*)
— **nowait**
Ich znaczenie jest takie, jak dla klauzuli **for**. Podobnie, na koniec jest dodawana domyślna bariera, którą można znieść stosując **nowait**. Przydział bloków poprzedzonych konstrukcją **section** odbywa się zawsze dynamicznie.

4.4.3. Konstrukcja *single*

Konstrukcja **single** umieszczona w bloku strukturalnym po **parallel** powoduje, że tylko jeden wątek wykonuje blok strukturalny. Jej postać jest następująca.

```
1 #pragma omp parallel ....
2   {
3     ...
4    #pragma omp single ....
5     {
6       ...
7     }
8   ...
9 }
```

Po konstrukcji **single** jest umieszczana niejawna bariera. Lista dopuszczalnych klauzul jest następująca.
— **private**(*lista zmiennych*)
— **firstprivate**(*lista zmiennych*)
— **nowait**
Ich znaczenie jest identyczne jak opisane wcześniej.

4.5. Połączone dyrektywy dzielenia pracy

Przedstawimy teraz dyrektywy ułatwiające zrównoleglanie istniejących fragmentów kodu. Stosowane są wtedy, gdy konstrukcja **for** lub **sections** jest jedyną częścią bloku strukturalnego po dyrektywie **parallel**.

4.5.1. Konstrukcja *parallel for*

Postać konstrukcji jest następująca.

```
1 #pragma omp parallel for ...
2   for (...;...;...) {
3     ...
4   }
```

Możliwe do zastosowania klauzule są takie jak dla **parallel** oraz **for**, z wyjątkiem **nowait**. Poniżej pokazujemy przypadki jej zastosowania.

Niech $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ oraz $\alpha \in \mathbb{R}$. Poniższy kod realizuje operację aktualizacji wektora (tzw. AXPY) postaci $\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}$. Zakładamy, że wektory $\mathbf{x}, \mathbf{y}$ są reprezentowane w tablicach $\mathbf{x}$[**MAX**], $\mathbf{y}$[**MAX**].

```
1 float x[MAX], y[MAX], alpha;
2 int i,n;
```

```
3
4   #pragma omp parallel for shared(n,x,y,alpha)
5     for (i=0;i<n;i++){
6       y[i]+=alpha*x[i];
7     }
```

Niech teraz $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ oraz $dot \in \mathbb{R}$. Poniższy kod realizuje operację wyznaczania iloczynu skalarnego (tzw. *DOT product*) postaci $dot \leftarrow \mathbf{y}^T\mathbf{x}$. Podobnie jak w powyższym przykładzie, wektory $\mathbf{x}, \mathbf{y}$ są reprezentowane w tablicach jednowymiarowych **x[MAX]**, **y[MAX]**.

```
1  float x[MAX], y[MAX], dot;
2  int i,n;
3
4  #pragma omp parallel for shared(n,x,y) \
5                           reduction(+:dot)
6   for (i=0;i<n;i++){
7     dot+=y[i]*x[i];
8   }
```

Niech teraz $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$ oraz $A \in \mathbb{R}^{m \times n}$. Poniższy kod realizuje operację wyznaczania iloczynu macierzy przez wektor postaci

$$\mathbf{y} \leftarrow \mathbf{y} + A\mathbf{x}.$$

Wektory $\mathbf{x}, \mathbf{y}$ są reprezentowane w tablicach **x[MAX]**, **y[MAX]**, zaś macierz A w tablicy dwuwymiarowej **a[MAX][MAX]**.

```
1  float x[MAX], y[MAX], a[MAX][MAX];
2  int i,n, m;
3
4  #pragma omp parallel for shared(m,n,x,y,a) private(j)
5   for (i=0;i<m;i++)
6     for (j=0;i<n;j++)
7       y[i]+=a[i][j]*x[j];
```

4.5.2. Konstrukcja *parallel sections*

Ogólna postać konstrukcji jest następująca.

```
1  #pragma omp parallel sections ....
2   {
3     #pragma omp section
4       { // blok strukturalny 1
5         ...
6       }
7     #pragma omp section
8       { // blok strukturalny 2
```

```
9           ...
10        }
11      #pragma omp section
12        { // blok strukturalny 3
13          ...
14        }
15       ...
16   }
```

Możliwe do zastosowania klauzule są takie jak dla **parallel** oraz **sections**, z wyjątkiem **nowait**.

4.6. Konstrukcje zapewniające synchronizację grupy wątków

Przedstawimy teraz ważne konstrukcje synchronizacyjne, które mają szczególnie ważne znaczenia dla zapewnienia prawidłowego dostępu do wspólnych zmiennych w pamięci.

4.6.1. Konstrukcja *barrier*

Konstrukcja definiuje jawną barierę następującej postaci.

```
1    ...
2   #pragma omp barrier
3    ...
```

Umieszczenie tej dyrektywy powoduje wstrzymanie wątków, które dotrą do bariery aż do czasu, gdy wszystkie wątki osiągną to miejsce w programie.

4.6.2. Konstrukcja *master*

Konstrukcja występuje we wnętrzu bloku strukturalnego po `parallel` i oznacza, że blok strukturalny jest wykonywany tylko przez wątek główny. Nie ma domyślnej bariery na wejściu i wyjściu. Postać konstrukcji jest następująca.

```
1  #pragma omp parallel ....
2   {
3     ...
4    #pragma omp master
5      {
6        ...
7      }
8  ...
9   }
```

4.6.3. Konstrukcja *critical*

Konstrukcja występuje we wnętrzu bloku strukturalnego po *parallel* i oznacza, że blok strukturalny jest wykonywany przez wszystkie wątki w trybie wzajemnego wykluczania, czyli stanowi sekcję krytyczną. Postać konstrukcji jest następująca.

```
#pragma omp parallel ....
  {
    ...
    #pragma omp critical
      {
        ...
      }
    ...
  }
```

Możliwa jest również postać z nazwanym regionem krytycznym.

```
#pragma omp parallel ....
  {
    ...
    #pragma omp critical(nazwa)
      {
        ...
      }
    ...
  }
```

Wątek czeka na wejściu do sekcji krytycznej aż do chwili, gdy żaden inny wątek nie wykonuje sekcji krytycznej (o podanej nazwie). Następujący przykład ilustruje działanie **critical**. Rozważmy następujący kod sumujący wartości składowych tablicy.

```
#pragma omp parallel for reduction(+:sum)
  for(i=0;i<n;i++){
    sum+=a[i];
  }
```

Równoważny kod bez użycia *reduction* wymaga zastosowania konstrukcji *critical*.

```
#pragma omp parallel private(priv_sum) shared(sum)
  {
    priv_sum=0;
    #pragma omp for nowait
      for(i=0;i<n;i++){
        priv_sum+=a[i];
      }
```

```
8     #pragma omp critical
9       {
10       sum+=priv_sum;
11      }
12   }
```

Jako przykład rozważmy problem wyznaczenia wartości maksymalnej wśród składowych tablicy. Prosty algorytm sekwencyjny przyjmie postać.

```
1  max=a[0];
2  for(i=1;i<n;i++)
3     if(a[i]>max)
4        max=a[i];
```

Zrównoleglenie wymaga zastosowania sekcji krytycznej. Otrzymamy w ten sposób następujący algorytm, który właściwie będzie działał sekwencyjnie.

```
1  max=a[0];
2  #pragma omp parallel for
3  for(i=1;i<n;i++)
4     #pragma omp critical
5     if(a[i]>max)
6        max=a[i];
```

Poniższa wersja działa efektywniej, gdyż porównania z linii numer 4 są wykonywane równolegle.

```
1  max=a[0];
2  #pragma omp parallel for
3  for(i=1;i<n;i++)
4     if(a[i]>max){
5        #pragma omp critical
6           if(a[i]>max)
7              max=a[i];
8     }
```

W przypadku jednoczesnego wyznaczania minimum i maksimum można użyć nazw sekcji krytycznych. Otrzymamy w ten sposób następujący algorytm.

```
1  max=a[0];
2  min=a[0];
3  #pragma omp parallel for
4  for(i=1;i<n;i++){
5     if(a[i]>max){
6        #pragma omp critical(maximum)
7           if(a[i]>max)
8              max=a[i];
9     }
```

```
10      if(a[i]<mim){
11        #pragma omp critical(minimum)
12          if(a[i]<min)
13            min=a[i];
14      }
15    }
```

Trzeba jednak zaznaczyć, że problem znalezienia maksimum lepiej rozwiązać algorytmem następującej postaci.

```
1   max=a[0];
2   priv_max=a[0];
3   #pragma omp parallel private(priv_max)
4   {
5     #pragma omp for nowait
6       for(i=0;i<n;i++)
7           if(a[i]>priv_max)
8             priv_max=a[i];
9     #pragma omp critical
10      if(priv_max>max)
11        max=priv_max;
12  }
```

4.6.4. Konstrukcja *atomic*

W przypadku, gdy w sekcji krytycznej aktualizujemy wartość zmiennej, lepiej jest posłużyć się konstrukcją *atomic* następującej postaci.

```
1   #pragma omp parallel ....
2     {
3       ...
4       #pragma omp atomic
5         zmienna=expr;
6       ...
7     }
```

4.6.5. Dyrektywa *flush*

Dyrektywa powoduje uzgodnienie wartości zmiennych wspólnych podanych na liście, albo gdy nie ma listy – wszystkich wspólnych zmiennych.

```
1 #pragma omp parallel ....
2   {
3   ...
4     #pragma omp flush(lista zmiennych)
5   ...
6   }
```

Uzgodnienie wartości zmiennych następuje automatycznie w następujących sytuacjach:
— po barierze (dyrektywa **barrier**),
— na wejściu i wyjściu z sekcji krytycznej (konstrukcja **critical**),
— na wejściu i wyjściu z konstrukcji **ordered**),
— na wyjściu z **parallel**, **for**, **section** oraz **single**.

4.7. Biblioteka funkcji OpenMP

Przedstawimy teraz wybrane (najważniejsze) funkcje zdefiniowane w ramach standardu OpenMP.

— **void omp_set_num_threads(int num)** określa, że liczba wątków w grupie (dla następnych regionów równoległych) ma wynosić `num`.
— **int omp_get_num_threads(void)** zwraca liczbę wątków w aktualnie realizowanym regionie równoległym.
— **int omp_get_thread_num(void)** zwraca numer danego wątku w aktualnie realizowanym regionie równoległym.
— **int omp_get_num_procs(void)** zwraca liczbę procesorów, które są dostępne dla programu.
— **int omp_in_parallel(void)** zwraca informację, czy aktualnie jest wykonywany region równoległy.
— **void omp_set_nested(int)** ustawia pozwolenie lub zabrania na zagnieżdżanie wykonania regionów równoległych (wartość zerową traktuje się jako *false*, różną od zera jako *true*).
— **int omp_get_nested(void)** zwraca informację, czy dozwolone jest zagnieżdżanie wykonania regionów równoległych (wartość zerową traktuje się jako *false*, różną od zera jako *true*).

Ciekawym mechanizmem oferowanym przez *runtime* OpenMP jest dynamiczne dopasowywanie liczby wątków do dostępnych zasobów (procesorów) w komputerze. W przypadku gdy jednocześnie wykonuje się wiele programów równoległych, przydzielenie każdemu jednakowej liczby procesorów może prowadzić do degradacji szybkości wykonania programu. W takim przypadku OpenMP dynamicznie dopasuje liczbę wątków wykonujących region równoległy do dostępnych zasobów. Trzeba podkreślić, że w ramach wykonywanego regionu równoległego liczba wątków jest zawsze niezmienna. Mechanizmem tym możemy sterować posługując się wywołaniami następujących funkcji.
— `void omp_set_dynamic(int)` ustawia pozwolenie lub zabrania na dynamiczne dopasowywanie liczby wątków (wartość zerową traktuje się jako *false*, różną od zera jako *true*).

— `int omp_get_nested(void)` zwraca informację, czy dozwolone jest dynamiczne dopasowywanie liczby wątków (wartość zerową traktuje się jako *false*, różną od zera jako *true*).

Omówione powyżej funkcjonalności mogą być również ustawione z poziomu zmiennych środowiskowych.

— `OMP_SCHEDULE` definiuje sposób szeregowania obrotów pętli w konstrukcji **for** (na przykład **"dynamic, 16"**).
— `OMP_NUM_THREADS` określa liczbę wątków w grupie.
— `OMP_NESTED` pozwala lub zabrania na zagnieżdżanie regionów równoległych (należy ustawiać **TRUE** lub **FALSE**).
— `OMP_DYNAMIC` pozwala lub zabrania na dynamiczne dopasowywanie liczby wątków (należy ustawiać **TRUE** lub **FALSE**).

Przy ustawianiu zmiennych środowiskowych należy się posługiwać odpowiednimi poleceniami wykorzystywanej powłoki. Przykładowo dla powłok *sh* oraz *bash* przyjmie to postać

```
export OMP_NUM_THREADS=16
```

zaś dla powłok *csh* oraz *tcsh* następującą postać

```
setenv OMP_NUM_THREADS 16
```

Do mierzenia czasu obliczeń w programach OpenMP możemy wykorzystać funkcję `omp_get_wtime`.

— **double omp_get_wtime( )** zwraca liczbę sekund jaka upłynęła od pewnego, z góry ustalonego, punktu z przeszłości.

Konieczne jest jej dwukrotne wywołanie, co ilustruje poniższy fragment kodu.

```
1  double start = omp_get_wtime( );
2
3  ... // obliczenia, których czas chcemy zmierzyć
4
5  double end = omp_get_wtime( );
6
7  printf("start = %.12f\n", start);
8  printf("end = %.12f\n", end);
9  printf("diff = %.12f\n", end - start);
```

Przedstawione w sekcji 4.6 mechanizmy służące synchronizacji wątków mogą się okazać niewystarczające w przypadku bardziej złożonych algorytmów. Wówczas można skorzystać z mechanizmów synchronizacji oferowanych przez zestaw funkcji zdefiniowanych w ramach standardu OpenMP, które wykorzystują pojęcie *blokady* (ang. lock) – zmiennych typu `omp_lock_t`.

— **void omp_init_lock(omp_lock_t *lock)** inicjuje blokadę.
— **void omp_destroy_lock(omp_lock_t *lock)** niszczy blokadę zwalniając pamięć zajmowaną przez zmienną.
— **void omp_set_lock(omp_lock_t *lock)** wykonuje próbę przejęcia blokady. Jeśli blokada jest wolna, wówczas wątek wywołujący funkcję przejmuje blokadę na własność i kontynuuje działanie. W przypadku, gdy blokada jest w posiadaniu innego wątku, wątek wywołujący oczekuje na zwolnienie blokady.
— **void omp_unset_lock(omp_lock_t *lock)** zwalnia blokadę, w wyniku czego inne wątki mogą współzawodniczyć w próbie przejęcia na własność danej blokady.
— **int omp_test_lock(omp_lock_t *lock)** testuje i ewentualnie przejmuje blokadę. Jeśli blokada jest dostępna (wolna), wówczas przejmuje blokadę zwracając zero. Gdy blokada jest w posiadaniu innego wątku, wówczas zwracana jest wartość różna od zera. W obu przypadkach wątek kontynuuje pracę.

Wykorzystanie blokad ilustruje następujący przykład.

```
1  omp_lock_t lck;
2  omp_init_lock(&lck);
3  sum=0;
4
5  #pragma omp parallel private(priv_sum) shared(sum,lck)
6   {
7     priv_sum= .....; // obliczenia lokalne
8
9     omp_set_lock(&lck);
10    sum+=priv_sum;
11    omp_unset_lock(&lck);
12  }
13
14  omp_destroy_lock(&lck);
```

4.8. Przykłady

Podamy teraz kilka przykładów algorytmów numerycznych, których wykonanie można przyspieszyć łatwo stosując zrównoleglenie przy pomocy dyrektyw OpenMP. Więcej informacji na temat podanych algorytmów numerycznych można znaleźć w książce [24].

4.8.1. Mnożenie macierzy

Rozważmy teraz problem wyznaczenia iloczynu macierzy AB, a ściślej wykonania operacji $C \leftarrow C + AB$, gdzie $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$ oraz $C \in$

$\mathbb{R}^{m\times n}$, przy wykorzystaniu wzoru

$$c_{ij} \leftarrow c_{ij} + \sum_{l=1}^{k} a_{il} b_{lk}, \qquad i = 1, \ldots, m,\ j = 1, \ldots, n.$$

Zdefiniowane wyżej obliczenia można wykonać posługując się następującym kodem sekwencyjnym.

```
1  int i,j,l;
2  for(i=0;i<m;i++)
3     for(j=0;j<n;j++)
4        for(l=0;l<k;l++)
5           c[i][j]+=a[i][l]*b[l][j];
```

Zrównoleglenie może być zrealizowane „wierszami", to znaczy zrównoleglona zostanie zewnętrzna pętla. Otrzymamy w ten sposób poniższy kod (listing 4.3).

Listing 4.3. Równoległe mnożenie macierzy

```
1 int i,j,l;
2 #pragma omp parallel for shared(a,b,c) private(j,l) \
3                                                    schedule(static)
4 for(i=0;i<m;i++){
5    for(j=0;j<n;j++)
6       for(l=0;l<k;l++)
7          c[i][j]+=a[i][l]*b[l][j];
8 }
```

4.8.2. Metody iteracyjne rozwiązywania układów równań

Niech będzie dany układ równań liniowych

$$A\mathbf{x} = \mathbf{b}, \tag{4.1}$$

gdzie $A \in \mathbb{R}^{n\times n}$, $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ oraz macierz A jest nieosobliwa, przy czym $a_{ii} \neq 0$, $i = 1, \ldots, n$. Niech dalej będą zdefiniowane macierze

$$L = \begin{pmatrix} 0 & & & 0 \\ a_{21} & 0 & & \\ \vdots & \ddots & \ddots & \\ a_{n1} & \ldots & a_{n,n-1} & 0 \end{pmatrix}, \qquad U = \begin{pmatrix} 0 & a_{12} & \ldots & a_{1n} \\ & 0 & & \vdots \\ & & \ddots & a_{n-1,n} \\ 0 & & & 0 \end{pmatrix}$$

oraz

$$D = \begin{pmatrix} a_{11} & & & 0 \\ & a_{22} & & \\ & & \ddots & \\ 0 & & & a_{nn} \end{pmatrix} = \mathrm{diag}(a_{11}, \ldots, a_{nn})$$

takie, że $A = L + D + U$. Wówczas przybliżenie rozwiązania układu (4.1) może być wyznaczone metodą iteracyjną *Jacobiego*

$$\mathbf{x}^{k+1} = -D^{-1}((L+U)\mathbf{x}^k - \mathbf{b}). \tag{4.2}$$

Zauważmy, że wzór (4.2) wykorzystuje operację mnożenia macierzy przez wektor, a zatem nadaje się do łatwego zrównoleglenia. Inną metodę iteracyjną stanowi metoda *Gaussa-Seidla* określona następującym wzorem

$$\mathbf{x}^{k+1} = -(L+D)^{-1}((U\mathbf{x}^k - \mathbf{b}), \tag{4.3}$$

która wymaga w każdym kroku rozwiązania układu równań liniowych o macierzy dolnotrójkątnej. Następujące twierdzenie charakteryzuje zbieżność obu wprowadzonych wyżej metod.

Twierdzenie 4.1. *Niech macierz $A \in \mathbb{R}^{n\times n}$ będzie macierzą o dominującej głównej przekątnej:*

$$|a_{ii}| > \sum_{j\neq i} |a_{ij}|.$$

Wówczas metody Jacobiego i Gaussa-Seidla są zbieżne do jednoznacznego rozwiązania układu $A\mathbf{x} = \mathbf{b}$ dla dowolnego przybliżenia początkowego $\mathbf{x}^0$.

W metodach iteracyjnych (4.2) oraz (4.3) stosuje się powszechnie następujące dwa kryteria stopu (zakończenia postępowania iteracyjnego):

— maksymalna względna zmiana składowej przybliżonego rozwiązania nie przekracza pewnego z góry zadanego małego parametru ε:

$$\max_{1\leq i\leq n}\{|x_i^{k+1} - x_i^k|\} < \varepsilon \max_{1\leq i\leq n}\{|x_i^k|\}, \tag{4.4}$$

— składowe wektora residualnego $r^k = b - Ax^k$ będą stosunkowo niewielkie, a ściślej

$$\max_{1\leq i\leq n}\{|r_i^k|\} < \varepsilon. \tag{4.5}$$

Poniżej (listing 4.4) przedstawiamy równoległą implementację metody Jacobiego z kryterium stopu (4.5). Jednocześnie pozostawiamy Czytelnikowi implementację metody Gaussa-Seidla.

Listing 4.4. Równoległa implementacja metody Jacobiego

```
1  double a[MAXN][MAXN], b[MAXN], x_old[MAXN],
2                                  x_new[MAXN], r[MAXN];
3  res=1.0e+20;
4  eps=1.0e-10;
5  #pragma omp parallel default(shared) private(i,j,rmax)
6   {
7     while(res>=eps){
8
9       #pragma omp for schedule(static) nowait
10      for(i=0;i<n;i++){
11        x_new[i]=b[i];
12        for(j=0;i<i;j++)
13            x_new[j]+=a[i][j]*x_old[j];
14        for(j=i+1;i<n;j++)
15            x_new[j]+=a[i][j]*x_old[j];
16        x_new[i]/=-a[i][i];
17      }
18      #pragma omp single
19       {
20       res=0;
21       }
22      rmax=0;
23     // tu jest bariera (domyślna po single)
24      #pragma omp for schedule(static) nowait
25      for(i=0;i<n;i++){
26        x_old[i]=x_new[i];
27        r[i]=b[i];
28        for(j=0;j<n;j++)
29            r[i]-=a[i][j]*x_old[j];
30        if(abs(r[i])>rmax)
31            rmax=abs(r[i]);
32      }
33      #pragma omp critical
34       {
35        if(res<rmax)
36          res=rmax;
37       }
38    }
39  }
```

4.8.3. Równanie przewodnictwa cieplnego

Rozważmy następujące równanie różniczkowe (tzw. *równanie przewodnictwa cieplnego*)

$$u_t = u_{xx}, \qquad a \le x \le b, \quad t \ge 0, \tag{4.6}$$

gdzie indeksy oznaczają pochodne cząstkowe. Przyjmijmy następujące warunki brzegowe

$$u(0,x) = g(x), \qquad a \le x \le b \tag{4.7}$$

oraz

$$u(t,a) = \alpha, \quad u(t,b) = \beta, \qquad t \ge 0, \tag{4.8}$$

gdzie g jest daną funkcją, zaś α, β danymi stałymi. Równanie (4.6) wraz z warunkami (4.7)–(4.8) jest matematycznym modelem temperatury u cienkiego pręta, na którego końcach przyłożono temperatury α i β. Rozwiązanie $u(t,x)$ określa temperaturę w punkcie x i czasie t, przy czym początkowa temperatura w punkcie x jest określona funkcją $g(x)$.

Równanie (4.6) może być również uogólnione na więcej wymiarów. W przypadku dwuwymiarowym równanie

$$u_t = u_{xx} + u_{yy} \tag{4.9}$$

określa temperaturę cienkiego płata o wymiarach 1×1, czyli współrzędne x, y spełniają nierówności

$$0 \le x, y \le 1.$$

Dodatkowo przyjmujemy stałą temperaturę na brzegach płata

$$u(t,x,y) = g(x,y), \qquad (x,y) \text{ na brzegach} \tag{4.10}$$

oraz zakładamy, że w chwili $t = 0$ temperatura w punkcie (x,y) jest określona przez funkcję $f(x,y)$, czyli

$$u(0,x,y) = f(x,y). \tag{4.11}$$

Podamy teraz proste metody obliczeniowe wyznaczania rozwiązania równań (4.6) i (4.9) wraz z implementacją przy użyciu standardu OpenMP.

4.8.3.1. Rozwiązanie równania 1-D

Aby numerycznie wyznaczyć rozwiązanie równania (4.6) przyjmijmy, że rozwiązanie będzie dotyczyć punktów siatki oddalonych od siebie o Δx oraz Δt – odpowiednio dla zmiennych x i t. Pochodne cząstkowe zastępujemy ilorazami różnicowymi. Oznaczmy przez u_j^m przybliżenie rozwiązania w punkcie $x_j = j\Delta x$ w chwili $t_m = m\Delta t$, przyjmując $\Delta x = 1/(n+1)$. Wówczas otrzymamy następujący schemat różnicowy dla równania (4.6)

$$\frac{u_j^{m+1} - u_j^m}{\Delta t} = \frac{1}{(\Delta x)^2}(u_{j+1}^m - 2u_j^m + u_{j-1}^m), \tag{4.12}$$

lub inaczej

$$u_j^{m+1} = u_j^m + \mu(u_{j+1}^m - 2u_j^m + u_{j-1}^m), \qquad j = 1, \ldots, n, \tag{4.13}$$

gdzie

$$\mu = \frac{\Delta t}{(\Delta x)^2}. \tag{4.14}$$

Warunki brzegowe (4.7) przyjmą postać

$$u_0^m = \alpha, \quad u_{n+1}^m = \beta, \qquad m = 0, 1, \ldots,$$

zaś warunek początkowy (4.7) sprowadzi się do

$$u_j^0 = g(x_j), \qquad j = 1, \ldots, n.$$

Schemat różnicowy (4.13) jest schematem *otwartym* albo inaczej *jawnym* (ang. explicit). Do wyznaczenia wartości u_j^{m+1} potrzebne są jedynie wartości wyznaczone w poprzednim kroku czasowym, zatem obliczenia przeprowadzane według wzoru (4.13) mogą być łatwo zrównoleglone. Trzeba tutaj koniecznie zaznaczyć, że przy stosowaniu powyższego schematu bardzo istotnym staje się właściwy wybór wielkości kroku czasowego Δt. Schemat (4.13) będzie *stabilny*, gdy wielkości Δt oraz Δx będą spełniały nierówność

$$\Delta t \leq \frac{1}{2}(\Delta x)^2. \tag{4.15}$$

Poniżej przedstawiamy kod programu w OpenMP, który realizuje obliczenia w oparciu o wprowadzony wyżej schemat (4.13).

Listing 4.5. Rozwiązanie równania przewodnictwa cieplnego (1-D)

```
1  #include <stdio.h>
2  #include <omp.h>
3  #define MAXN 100002
4
5  double u_old[MAXN], u_new[MAXN];
6  int main(){
7    double mu,dt,dx,alpha,beta,time;
8    int j,m,n,max_m;
9    n=100000;
10   max_m=10000;
11   alpha=0.0;  beta=10.0;
12
13   u_old[0]=alpha;  u_new[0]=alpha;
14   u_old[n+1]=beta; u_new[n+1]=beta;
15
16   dx=1.0/(double)(n+1);
17   dt=0.4*dx*dx;
18   mu=dt/(dx*dx);
19
20   #pragma omp parallel default(shared) private(j,m)
21     {
```

```
22        #pragma omp for schedule(static)
23          for(j=1;j<=n;j++){
24            u_old[j]=20.0;
25          }
26
27        for(m=0;m<max_m;m++){
28          #pragma omp for schedule(static)
29          for(j=1;j<=n;j++){
30            u_new[j]
31                =u_old[j]+mu*(u_old[j+1]-2*u_old[j]+u_old[j-1]);
32          }
33          #pragma omp for schedule(static)
34          for(j=1;j<=n;j++){
35            u_old[j]=u_new[j];
36          }
37        }
38      }
39    // wydanie wyników ........
40    }
```

Przykładowe czasy wykonania programu na komputerze z dwoma procesorami Xeon Quadcore 3.2GHz dla użytych ośmiu, czterech oraz jednego rdzenia wynoszą odpowiednio 0.29, 0.44, 1.63 sekundy.

4.8.3.2. Rozwiązanie równania 2-D

W celu wyznaczenia rozwiązania równania (4.9) przyjmijmy, że wewnętrzne punkty siatki są dane przez

$$(x_i, y_j) = (ih, jh), \qquad i, j = 1, \ldots, n,$$

gdzie $(n+1)h = 1$. Stosując przybliżenia pochodnych cząstkowych

$$u_{xx}(x_i, y_j) \doteq \frac{1}{h^2}[u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)]$$

oraz

$$u_{yy}(x_i, y_j) \doteq \frac{1}{h^2}[u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})],$$

otrzymujemy schemat obliczeniowy

$$u_{ij}^{m+1} = u_{ij}^m + \frac{\Delta t}{h^2}(u_{i,j+1}^m + u_{i,j-1}^m + u_{i+1,j}^m + u_{i-1,j}^m - 4u_{ij}^m), \tag{4.16}$$

dla $m = 0, 1, \ldots$, oraz $i, j = 1, \ldots, n$. Wartość u_{ij}^m oznacza przybliżenie wartości temperatury w punkcie siatki o współrzędnych (i, j) w m-tym kroku czasowym. Podobnie jak w przypadku równania 1-D określone są warunki brzegowe

$$u_{0,j} = g(0, y_j), \quad u_{n+1,j} = g(1, y_j), \qquad j = 0, 1, \ldots, n+1,$$

$$u_{i,0} = g(x_i, 0), \quad u_{i,n+1} = g(x_i, 1), \qquad i = 0, 1, \ldots, n+1,$$

oraz warunki początkowe

$$u_{ij}^0 = f_{ij}, \qquad i, j = 1, \ldots, n.$$

Z uwagi na wymóg stabilności, wielkości Δt oraz h powinny spełniać nierówność

$$\Delta t \leq \frac{h^2}{4}.$$

Listing 4.6 przedstawia fragment kodu realizującego schemat iteracyjny (4.16).

Listing 4.6. Rozwiązanie równania przewodnictwa cieplnego (2-D)

```
1
2    dx=1.0/(double)(n+1);
3    dt=0.4*dx*dx;
4    mu=dt/(dx*dx);
5
6    #pragma omp parallel default(shared) private(i,j,m)
7    {
8      for(m=0;m<max_m;m++){
9       #pragma omp for schedule(static) private(j)
10        for(i=1;i<=n;i++)
11          for(j=1;j<=n;j++)
12            u_new[i][j]=u_old[i][j]+mu*(u_old[i][j+1]+
13                          u_old[i][j-1]+u_old[i+1][j]+
14                          u_old[i-1][j]-4*u_old[i][j]);
15
16      #pragma omp for schedule(static) private(j)
17        for(i=1;i<=n;i++)
18          for(j=1;j<=n;j++)
19            u_old[i][j]=u_new[i][j];
20    }
21  }
```

4.9. Zadania

Poniżej zamieściliśmy szereg zadań do samodzielnego wykonania. Do ich rozwiązania należy wykorzystać standard OpenMP.

Zadanie 4.1.

Napisz program wczytujący ze standardowego wejścia liczbę całkowitą n ($n \leq 0$), która ma stanowić rozmiar dwóch tablic **a** i **b**.

Następnie w bloku równoległym zamieść dwie pętle for. Niech w pierwszej pętli wątki w sposób równoległy wypełnią obydwie tablice wartościami,

do pierwszej wstawiając wartość swojego identyfikatora, do drugiej całkowitą wartość losową z zakresu $< 0; 10)$. W drugiej pętli wykonaj równoległe dodawanie tych tablic. Wynik zamieść w tablicy **a**. Do podziału pracy pomiędzy wątki użyj dyrektywy „omp for" oraz szeregowania statycznego z kwantem 5. Po wyjściu z bloku wyświetl obydwie tablice.

Zadanie 4.2.

Opisz w postaci funkcji algorytm równoległy, zwracający maksimum z wartości bezwzględnych elementów tablicy **a**, gdzie tablica **a** oraz jej rozmiar są parametrami tej funkcji.

Zadanie 4.3.

Napisz program wczytujący ze standardowego wejścia liczbę całkowitą n ($n \leq 0$), a następnie n liczb. Program ma wyświetlić na standardowym wyjściu sumę tych liczb. Sumowanie powinno zostać wykonane przez 4 wątki. Jeżeli n nie jest podzielne przez cztery, to dodatkowe elementy sumowane powinny być przez wątek główny.

Zadanie 4.4.

Opisz w postaci funkcji `double sredniaArytm(double a[], int n)` algorytm wyznaczający średnią arytmetyczną ze wszystkich elementów tablicy **a** o rozmiarze n. Wykorzystaj operację redukcji z operatorem „+".

Zadanie 4.5.

Opisz w postaci funkcji algorytm równoległy wyznaczający wartość poniższego ciągu, gdzie n jest parametrem tej funkcji. Wykorzystaj operację redukcji z operatorem „-".

$$-1 - \frac{1}{2} - \frac{1}{3} ... - \frac{1}{n}$$

Zadanie 4.6.

Niech funkcje `f1`, `f2`, `f3`, `f4` i `f5` będą funkcjami logicznymi (np. zwracającymi wartość „prawda" w przypadku gdy wykonały się one pomyślnie oraz „fałsz" gdy ich wykonanie nie powiodło się.) Opisz w postaci funkcji algorytm równoległy, który wykona funkcje od `f1` do `f5` i zwróci wartość „prawda" gdy wszystkie funkcje wykonają się z powodzeniem lub „fałsz" w przeciwnym wypadku. Algorytm nie powinien wymuszać aby każda funkcja wykonana została przez osobny wątek.

Zadanie 4.7.

Opisz w postaci funkcji algorytm równoległy wyznaczający wartość liczby π ze wzoru Wallisa.

$$\pi = 2 \prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)}$$

Zadanie 4.8.

Opisz w postaci funkcji algorytm równoległy wyznaczający wartość liczby π metodą Monte Carlo.

Jeśli mamy koło oraz kwadrat opisany na tym kole to wartość liczby π można wyznaczyć ze wzoru:

$$\pi = 4\frac{P_{\bigcirc}}{P_{\square}}$$

We wzorze tym występuje pole koła, do czego potrzebna jest wartość liczby π. Sens metody Monte Carlo polega jednak na tym, że w ogóle nie trzeba wyznaczać pola koła. To co należy zrobić to wylosować odpowiednio dużą liczbę punktów należących do kwadratu i sprawdzić jaka ich część należy również do koła. Stosunek tej części do liczby wszystkich punktów będzie odpowiadał stosunkowi pola koła do pola kwadratu w powyższym wzorze.

Zadanie 4.9.

Dla dowolnego zadania, w którym wystąpiła redukcja, zmodyfikuj rozwiązanie w taki sposób, aby wykonać operację redukcji bez używania klauzuli `reduction`.

Zadanie 4.10.

Opisz w postaci funkcji algorytm wyznaczający wartość poniższej całki metodą prostokątów.

$$\int_0^1 (\frac{4}{1+x^2})dx$$

Zadanie 4.11.

Opisz w postaci funkcji `double iloczynSkal(double x[], double y[], int n)` algorytm wyznaczający iloczyn skalarny dwóch wektorów **x** i **y** w n-wymiarowej przestrzeni euklidesowej.

Zadanie 4.12.

Opisz w postaci funkcji `double dlugoscWektora(double x [], int n)` algorytm wyznaczający długość wektora **x** w n-wymiarowej przestrzeni euklidesowej.

Zadanie 4.13.

Opisz w postaci funkcji `void mnozMacierze(double A[], double B[], double C[], int lwierszyA, int lkolumnA, int lkolumnB)`, algorytm równoległy wykonujący równolegle mnożenie macierzy. Wynik mnożenia macierzy A i B należy zamieścić w macierzy C. Napisz program, w którym przetestujesz działanie funkcji. Przed mnożeniem program powinien zainicjować macierze A i B wartościami. Wykonaj pomiary czasu działania funkcji `mnozMacierze` dla odpowiednio dużej macierzy i dla różnej liczby wątków.

Zadanie 4.14.

Opisz w postaci funkcji `bool czyRowne(int a [], int b [], int n)` algorytm sprawdzający, czy wszystkie odpowiadające sobie elementy tablic **a** i **b** o rozmiarze n są równe.

Zadanie 4.15.

Opisz w postaci funkcji `int minIndeks(int a [], int M, int N, int wzor, int &liczba)` algorytm równoległy wyznaczający najwcześniejszy indeks wystąpienia wartości `wzor` w tablicy **a** wśród składowych **a**[**M**]..**a**[**N**]. Poprzez parametr wyjściowy `liczba` należy zwrócić liczbę wszystkich wystąpień wartości `wzor`.

Zadanie 4.16.

Podaj przykład zrównoleglonej pętli for, która będzie wykonywała się szybciej z szeregowaniem statycznym z kwantem 1, aniżeli statycznym bez określania kwantu, czyli z podziałem na w przybliżeniu równe, ciągłe grupy kroków pętli dla każdego wątku.

Zadanie 4.17.

Dodaj do przykładu 4.2 dyrektywy warunkowej kompilacji, tak aby program działał bez OpenMP.

Zadanie 4.18.

Opisz w postaci funkcji `int maxS(int *A[], int m, int n)` algorytm wyznaczający wartość

$$\max_{0 \le i \le m-1} \sum_{j=0}^{n-1} (|A_{ij}|),$$

gdzie m i n to odpowiednio liczba wierszy i kolumn macierzy A.

Zadanie 4.19.
Używając metody trapezów (patrz poniższy wzór)

$$\int_a^b f(x)dx \approx T(h) = h[\frac{1}{2}f_0 + f_1 + ... + f_{n-1} + \frac{1}{2}f_n]$$

napisz program równoległy liczący całkę:

$$\int_{0.01}^{1} (x + sin\Big(\frac{1}{x}\Big))dx$$

Rozdział 5

Message Passing Interface – podstawy

W tym rozdziale wprowadzimy podstawowe informacje na temat Message Passing Interface (MPI), jednego z najstarszych, ale ciągle rozwijanego, a przede wszystkim bardzo popularnego standardu programowania równoległego. Omówimy ogólną koncepcję MPI oraz podstawowe schematy komunikacji między procesami. Dla pełniejszego przestudiowania możliwości MPI odsyłamy Czytelnika do [47, 51, 56].

5.1. Wprowadzenie do MPI

Program MPI zakłada działanie kilku równoległych procesów, w szczególności procesów rozproszonych, czyli działających na różnych komputerach, połączonych za pomocą sieci.

W praktyce MPI jest najczęściej używany na klastrach komputerów i implementowany jako biblioteka funkcji oraz makr, które możemy wykorzystać pisząc programy w językach C/C++ oraz Fortran. Przykładowe implementacje dla tych języków to MPICH oraz OpenMPI. Oczywiście znajdziemy też wiele implementacji dla innych języków.

Naturalnym elementem każdego programu, w skład którego wchodzi kilka równoległych procesów (lub wątków), jest wymiana danych pomiędzy tymi procesami (wątkami). W przypadku wątków OpenMP odbywa się to poprzez współdzieloną pamięć. Jeden wątek zapisuje dane do pamięci, a następnie inne wątki mogą tę daną przeczytać. W przypadku procesów MPI rozwiązanie takie nie jest możliwe, ponieważ nie posiadają one wspólnej pamięci.

Komunikacja pomiędzy procesami MPI odbywa się na zasadzie przesyłania komunikatów, stąd nazwa standardu. Poprzez komunikat należy rozumieć zestaw danych stanowiących właściwą treść wiadomości oraz informacje dodatkowe, np. identyfikator komunikatora, w ramach którego odbywa się komunikacja, czy numery procesów komunikujących się ze sobą. Komunikacja może zachodzić pomiędzy dwoma procesami, z których jeden wysyła wiadomość a drugi ją odbiera, wówczas nazywana jest komunikacją typu punkt-punkt, ale może też obejmować więcej niż dwa procesy i wówczas jest określana mianem komunikacji grupowej.

Na listingu 5.1 przedstawiono prosty program, od którego zaczniemy omawiać standard MPI.

Listing 5.1. Prosty program MPI w języku C - „Witaj świecie!”.

```
1 #include <stdio.h>
2 #include "mpi.h"
3
4 int main(int argc, char **argv)
5 {
```

```
6       int myid;
7       int numprocs;
8
9       // Funkcji MPI nie wywołujemy przed MPI_Init
10
11      MPI_Init(&argc, &argv);
12
13      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
14      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
15
16      printf("Witaj świecie! ");
17      printf("Proces %d z %d.\n", myid, numprocs);
18
19      MPI_Finalize();
20
21      // ... ani po MPI_Finalize
22
23      return 0;
24  }
```

Powyższy program skompilować możemy poleceniem:

```
mpicc program.c -o program
```

Aby uruchomić skompilowany program MPI wykorzystujemy polecenie:

```
mpirun -np p ./program
```

gdzie w miejsce `p` wpisujemy liczbę równoległych procesów MPI dla uruchamianego programu.

Wynik programu dla czterech procesów może być następujący:

```
Witaj świecie! Proces 0 z 5.
Witaj świecie! Proces 1 z 5.
Witaj świecie! Proces 3 z 5.
Witaj świecie! Proces 4 z 5.
Witaj świecie! Proces 2 z 5.
```

Oczywiście jest to tylko przykładowy wynik. Należy pamiętać, że procesy MPI wykonują się równolegle, zatem kolejność wyświetlania przez nie komunikatów na ekran będzie przypadkowa.

W powyższym programie znajdziemy kilka elementów stanowiących pewną ogólną strukturę każdego programu MPI. Po pierwsze w każdym programie znaleźć się musi poniższa dyrektywa.

```
#include "mpi.h"
```

W pliku `mpi.h` zawarte są wszystkie niezbędne definicje, makra i nagłówki funkcji MPI.

Następnie, aby móc korzystać z biblioteki MPI, zanim użyta zostanie jakakolwiek inna funkcja z tej biblioteki, musimy wywołać funkcję `MPI_Init`. Funkcja ta jako swoje argumenty przyjmuje wskaźniki do argumentów funkcji main. Na zakończenie programu konieczne jest wywołanie `MPI_Finalize`. Obydwie te funkcje stanowią swego rodzaju klamrę każdego programu MPI.

Kolejnymi elementami, które znajdziemy równie często w każdym programie MPI, są funkcje `MPI_comm_rank` oraz `MPI_comm_size`.

Składnia tych funkcji jest następująca[1]:

```
int MPI_comm_rank(MPI_comm comm, int *id)
```

MPI_Comm comm – [IN] Komunikator.
int *id – [OUT] Identyfikator procesu w ramach komunikatora `comm`.

```
int MPI_comm_size(MPI_comm comm, int *size)
```

MPI_Comm comm – [IN] Komunikator.
int *size – [OUT] Liczba wszystkich procesów w komunikatorze `comm`.

Pierwszym argumentem obydwu tych funkcji jest *komunikator*. Komunikator jest to przestrzeń porozumiewania się dla procesów MPI, które dołączyły do tej przestrzeni i dzięki temu mogą się komunikować. Inaczej mówiąc, jest to po prostu zbiór wszystkich procesów, które mogą wysyłać do siebie wzajemne komunikaty. W ramach jednego programu MPI może istnieć więcej niż jeden komunikator, ale dla prostych programów ograniczymy się do komunikatora `MPI_COMM_WORLD`, do którego należą wszystkie procesy uruchomione dla danego programu MPI. W drugim argumencie funkcji `MPI_comm_rank` zapisany zostanie identyfikator, jaki dany proces otrzymał w ramach komunikatora. Identyfikator procesu MPI jest liczbą od 0 do $p-1$, gdzie p to rozmiar danego komunikatora. Wartość ta zostanie zapisana w drugim, wyjściowym parametrze funkcji `MPI_comm_size`. Wywołanie obydwu tych funkcji daje każdemu z procesów informację na temat własnego otoczenia. W wielu programach wiedza ta będzie niezbędna do właściwego podziału pracy pomiędzy równoległe procesy.

Większość funkcji MPI zwraca stałą całkowitą oznaczającą kod błędu. `MPI_Success` oznacza prawidłowe wykonanie funkcji, a pozostałe kody zależą od implementacji MPI. W praktyce jednak bardzo często kody błędu są ignorowane, a funkcje wywoływane tak jak procedury[2]. Dlatego też, dla uproszczenia, w podawanych przez nas przykładach zostały one pominięte.

W odróżnieniu od wątków OpenMP, procesy programu MPI startują jednocześnie z chwilą startu programu, co można zaobserwować uruchamiając program z listingu 5.2.

[1] W całym rozdziale przy opisie będziemy używać oznaczeń [IN] oraz [OUT] dla wskazania parametrów wejściowych i wyjściowych.

[2] Procedurą czasem określa się funkcję typu `void`.

Listing 5.2. Ilustracja czasu działania procesów MPI.

```
1 #include <stdio.h>
2 #include "mpi.h"
3
4 int main(int argc, char **argv)
5 {
6     printf("Proces wystartował.\n");
7
8     int myid;
9     int numprocs;
10
11    MPI_Init(&argc, &argv);
12    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
13    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
14
15    printf("Proces %d z %d pracuje.\n", myid, numprocs);
16
17    MPI_Finalize();
18
19    printf("Proces kończy działanie.\n");
20
21    return 0;
22 }
```

Wynikiem tego programu dla czterech procesów będzie następujący wydruk:

```
Proces wystartował.
Proces wystartował.
Proces wystartował.
Proces wystartował.
Proces 0 z 4 pracuje.
Proces 1 z 4 pracuje.
Proces 2 z 4 pracuje.
Proces 3 z 4 pracuje.
Proces kończy działanie.
Proces kończy działanie.
Proces kończy działanie.
Proces kończy działanie.
```

Widzimy, że od początku do końca programu działa 4 procesy. Jednak przed `MPI_Init` i po `MPI_Finalize` procesy te nie należą do komunikatora MPI. W praktyce też, przed wywołaniem funkcji `MPI_Init`, nie będziemy zamieszczać nic oprócz definicji zmiennych. Podobnie po `MPI_Finalize` znajdziemy tylko operacje zwalniające dynamicznie przydzieloną pamięć.

Dla każdego procesu, oprócz jego identyfikatora, mamy możliwość sprawdzenia, na którym węźle równoległego komputera (klastra komputerów) został on uruchomiony. Posłuży do tego funkcja `MPI_Get_processor_name`.

Składnia tej funkcji jest następująca:

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

char *name – [OUT] Nazwa rzeczywistego węzła, na którym działa dany proces MPI. Wymagane jest, aby była to tablica o rozmiarze conajmniej `MPI_MAX_PROCESSOR_NAME`.
int *resultlen – [OUT] Długość tablicy `name`.

Specyfikacja tej funkcji określa, że wartość zapisana w tablicy `name` powinna jednoznacznie identyfikować komputer, na którym wystartował dany proces. Efekt może być różny w zależności od implementacji MPI. Przykładowo, może to być wynik działania takich funkcji jak: gethostname, uname czy sysinfo.

Użycie funkcji `MPI_Get_processor_name` zaprezentowano na listingu 5.3.

Listing 5.3. Działanie funkcji `MPI_Get_processor_name`

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  int main(int argc, char **argv)
5  {
6      int myid;
7      int numprocs;
8
9      int namelen;
10     char processor_name[MPI_MAX_PROCESSOR_NAME];
11
12     MPI_Init(&argc, &argv);
13     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
14     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
15
16     MPI_Get_processor_name(processor_name, &namelen);
17
18     printf("Proces %d z %d działa na serwerze %s.\n", myid,
19             numprocs, processor_name);
20
21     MPI_Finalize();
22
23     return 0;
24 }
```

Poniżej przedstawiono przykładowy wydruk będący wynikiem powyższego programu. Program wykonany został na klastrze dwóch komputerów pracujących pod systemem operacyjnym Centos. Uruchomiono 4 procesy, pod dwa na każdy węzeł klastra. Nazwy tutor1 oraz tutor2 to wynik systemowego polecenia hostname dla każdego z węzłów.

```
Proces 0 z 4 działa na serwerze tutor1.
Proces 1 z 4 działa na serwerze tutor1.
Proces 2 z 4 działa na serwerze tutor2.
Proces 3 z 4 działa na serwerze tutor2.
```

5.2. Komunikacja typu punkt-punkt

Do przesłania wiadomości pomiędzy dwoma procesami można użyć następujących funkcji: `MPI_Send` oraz `MPI_Recv`.

Pierwsza z nich służy do wysłania komunikatu, druga do jego odebrania. Ich składnia jest następująca:

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm )
```

void *buf – [IN] Adres początkowy bufora z danymi. Może to być adres pojedynczej zmiennej lub początek tablicy.
int count – [IN] Długość bufora `buf`. Dla pojedynczej zmiennej będzie to wartość 1.
MPI_Datatype datatype – [IN] Typ pojedynczego elementu bufora.
int dest – [IN] Identyfikator procesu, do którego wysyłany jest komunikat.
int tag – [IN] Znacznik wiadomości. Używany do rozróżniania wiadomości w przypadku gdy proces wysyła więcej komunikatów do tego samego odbiorcy.
MPI_Comm comm – [IN] Komunikator.

```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status )
```

void *buf – [OUT] Adres początkowy bufora do zapisu odebranych danych. Może to być adres pojedynczej zmiennej lub początek tablicy.
int count – [IN] Maksymalna liczba elementów w buforze `buf`.
MPI_Datatype datatype - [IN] Typ pojedynczego elementu bufora.
int source – [IN] Identyfikator procesu, od którego odbierany jest komunikat.
int tag – [IN] Znacznik wiadomości. Używany do rozróżniania wiadomości w przypadku gdy proces odbiera więcej komunikatów od tego samego nadawcy.
MPI_Comm comm – [IN] Komunikator.
MPI_Status *status – [OUT] Zmienna, w której zapisywany jest status przesłanej wiadomości.

Zawartość bufora po stronie nadawcy zostaje przekazana do bufora po stronie odbiorcy. Wielkość przesłanej wiadomości określają kolejne parametry funkcji `MPI_Send`, jest to ciąg `count` elementów typu `datatype`. Po stronie odbiorcy specyfikujemy maksymalną liczbę elementów, jaką może pomieścić bufor, oczywiście wiadomość odebrana zostanie również w przypadku gdy tych elementów będzie mniej. Parametr `datatype` określa typ elementów składowych wiadomości. Może to być jeden spośród predefiniownych typów MPI, których listę dla języka C przedstawiono w tabeli 5.1.

Tabela 5.1. Predefiniowane typy MPI dla języka C.

MPI_Datatype	Odpowiednik w języku C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Składowe wiadomości mogą być również typu złożonego. Mogą to być obiekty struktur bądź klas, możemy też tworzyć typy pochodne od przedstawionych w tabeli 5.1 typów predefiniowanych, o czym więcej w kolejnym rozdziale.

Przykład użycia funkcji `MPI_Send` oraz `MPI_Recv` przedstawiono na listingu 5.4.

Listing 5.4. Komunikacja typu punkt-punkt, funkcje `MPI_Send` oraz `MPI_Recv`. Przesłanie pojedynczej danej.

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  int main(int argc, char **argv)
5  {
6      int numprocs, myid;
7      int tag, from, to;
8
9      double data;
10     MPI_Status status;
11
12     MPI_Init(&argc, &argv);
```

```
13      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
14      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
15
16      if(myid == 0) {
17          to = 3;
18          tag = 2010;
19
20          data = 2.5;
21
22          MPI_Send(&data, 1, MPI_DOUBLE, to, tag,
23                   MPI_COMM_WORLD);
24      }
25      else if(myid == 3) {
26
27          from = 0;
28          tag = 2010;
29
30          MPI_Recv(&data, 1, MPI_DOUBLE, from, tag,
31                   MPI_COMM_WORLD, &status);
32
33          printf("Proces %d odebrał: %f\n", myid, data);
34
35      }else{
36          // Nie rób nic
37      }
38
39      MPI_Finalize();
40
41      return 0;
42  }
```

Program ten wymusza dla poprawnego działania uruchomienie czterech procesów MPI. Jakąkolwiek pracę wykonują tylko dwa z nich. Proces numer 0 wysyła wartość pojedynczej zmiennej, proces numer 3 tę wartość odbiera.

Wynikiem tego programu dla czterech procesów będzie poniższy wydruk:

```
Proces 3 odebrał: 2.500000
```

Kolejny przykład (listing 5.5) stanowi niewielką modyfikację poprzedniego. Tutaj również zachodzi komunikacja pomiędzy dwoma procesami. Tym razem proces numer 1 wysyła do procesu numer 3 tablicę danych.

Listing 5.5. Komunikacja typu punkt-punkt. Przesłanie tablicy danych. Zmienna status.

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  int main(int argc, char **argv)
5  {
```

```
6       int numprocs, myid;
7       int count, tag, from, to, i;
8       int r_count, r_source, r_tag;
9       double data[100];
10      MPI_Status status;
11
12      MPI_Init(&argc, &argv);
13      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
14      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
15
16      if(myid == 0) {
17
18          for (i=0; i<100; i++) data[i] = i;
19
20          count = 7;
21          to = 3;
22          tag = 2010;
23
24          MPI_Send(data, count, MPI_DOUBLE, to, tag,
25                   MPI_COMM_WORLD);
26      }
27      else if(myid == 3) {
28
29          count = 100;
30          from = MPI_ANY_SOURCE;
31          tag = MPI_ANY_TAG;
32
33          MPI_Recv(data, count, MPI_DOUBLE, from, tag,
34                   MPI_COMM_WORLD, &status);
35
36          MPI_Get_count(&status, MPI_DOUBLE, &r_count);
37          r_source= status.MPI_SOURCE;
38          r_tag= status.MPI_TAG;
39
40          printf("Informacja o zmiennej status\n");
41          printf("źródło : %d\n", r_source);
42          printf("znacznik : %d\n", r_tag);
43          printf("liczba odebranych elementów : %d\n", r_count);
44
45          printf("Proces %d odebrał : \n", myid);
46
47          for(i=0; i<r_count; i++) {
48              printf("%f ", data[i]);
49          }
50          printf("\n");
51      }
52
53      MPI_Finalize();
54
55      return 0;
56  }
```

W przykładzie tym podczas wywołania funkcji `MPI_Recv` nie określamy dokładnie od jakiego procesu ma przyjść komunikat, wstawiając w miejsce zmiennej `source` stałą `MPI_ANY_SOURCE`. Podobnie nie jest precyzowana zmienna `tag`, a w jej miejscu pojawia się stała `MPI_ANY_TAG`. Proces numer 3 jest skłonny tym samym odebrać wiadomość od dowolnego nadawcy z dowolnym znacznikiem. W takim przypadku bardzo pomocna staje się zmienna `status`, w której po odebraniu komunikatu znajdziemy takie informacje jak nadawca wiadomości, jej znacznik oraz wielkość. `MPI_Status` jest strukturą, pole `MPI_SOURCE` zawiera identyfikator nadawcy a pole `MPI_TAG` znacznik wiadomości. Do określenia liczny elementów składowych wiadomości należy wywołać funkcję `MPI_Get_count`.

Wynikiem tego programu dla czterech procesów będzie poniższy wydruk:

```
Informacja o zmiennej status:
źródło = 0
znacznik = 2010
liczba odebranych elementów = 7
Proces 3 odebrał:
0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000
```

W [51] znajdziemy program „Pozdrowienia” – klasyczny przykład ilustrujący komunikację typu punkt-punkt. W całości został on przytoczony na listingu 5.6.

Listing 5.6. Program „Pozdrowienia”.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "mpi.h"
4
5  int main(int argc, char **argv)
6  {
7      int myid, numprocs;
8
9      int source, dest, tag=2010;
10
11     char message[100];
12
13     MPI_Status status;
14
15     MPI_Init(&argc, &argv);
16     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
17     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
18
19     if (myid != 0)
20     {
21         sprintf(message, "Pozdrowienia od procesu %d!",
22                 myid);
23
```

```
24          dest = 0;
25          MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
26                   tag, MPI_COMM_WORLD);
27      }
28      else
29      {
30          for (source=1; source<numprocs; source++){
31
32              MPI_Recv(message, 100, MPI_CHAR, source, tag,
33                       MPI_COMM_WORLD, &status);
34
35              printf("%s\n", message);
36              }
37      }
38
39      MPI_Finalize();
40
41      return 0;
42  }
```

W programie tym proces numer 0 odbiera komunikaty od wszystkich pozostałych procesów. Wszystkie procesy, począwszy od procesu numer 1, przygotowują wiadomość w postaci łańcucha znaków i wysyłają go do procesu numer 0, który odbiera te wiadomości i wyświetla je na ekran. Kolejność odbierania wiadomości jest taka: najpierw od procesu numer 1, potem od procesu numer 2 i tak kolejno.

Wynikiem tego programu dla sześciu procesów będzie poniższy wydruk:

```
Pozdrowienia od procesu 1!
Pozdrowienia od procesu 2!
Pozdrowienia od procesu 3!
Pozdrowienia od procesu 4!
Pozdrowienia od procesu 5!
```

Algorytm odbierania wiadomości w przykładzie 5.6 jest dobry pod warunkiem, że z jakiegoś powodu zależy nam na odebraniu wiadomości w takiej kolejności. Jeśli natomiast zależy nam na tym, aby program był optymalny, a nie zależy nam na z góry ustalonej kolejności odbierania wiadomości, wówczas w programie należałoby zmodyfikować wywołanie funkcji `MPI_Recv`, jak poniżej.

```
32      MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag,
33               MPI_COMM_WORLD, &status);
```

Po takiej modyfikacji proces numer 0 nie będzie musiał czekać na któryś z kolejnych procesów w przypadku gdy ten jest jeszcze niegotowy, mimo

że w tym samym czasie w kolejce do nawiązania komunikacji czekają inne procesy.

Przykładowy wynik programu po tej modyfikacji dla sześciu procesów będzie poniższy wydruk:

```
Pozdrowienia od procesu 1!
Pozdrowienia od procesu 3!
Pozdrowienia od procesu 5!
Pozdrowienia od procesu 4!
Pozdrowienia od procesu 2!
```

Oczywiście wydruk ten może się różnić, i zapewne będzie, za każdym uruchomieniem programu.

Jeżeli procesy potrzebują wymienić się komunikatami nawzajem, wówczas można użyć funkcji `MPI_Sendrecv`, która łączy w sobie funkcjonalności zarówno `MPI_Send` jaki i `MPI_Recv`.

Jej składnia wygląda następująco:

```
int MPI_Sendrecv( void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  int dest, int sendtag,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype,
                  int source, int recvtag,
                  MPI_Comm comm, MPI_Status *status )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi do wysłania. Może to być adres pojedynczej zmiennej lub początek tablicy.
int sendcount – [IN] Długość bufora `sendbuf`. Dla pojedynczej zmiennej będzie to wartość 1.
MPI_Datatype sendtype – [IN] Typ elementów bufora `sendbuf`.
int dest – [IN] Identyfikator procesu, do którego wysyłany jest komunikat.
int sendtag – [IN] Znacznik wiadomości wysyłanej.
void *recvbuf – [OUT] Adres początkowy bufora do zapisu odebranych danych. Może to być adres pojedynczej zmiennej lub początek tablicy.
int recvcount – [IN] Liczba elementów w buforze `recvbuf`.
MPI_Datatype recvtype – [IN] Typ elementów bufora `recvbuf`.
int source – [IN] Identyfikator procesu, od którego odbierany jest komunikat.
int recvtag – [IN] Znacznik wiadomości odbieranej.
MPI_Comm comm – [IN] Komunikator.
MPI_Status *status – [OUT] Zmienna, w której zapisywany jest status przesłanej wiadomości.

Funkcja ta nie ogranicza się jedynie do komunikacji pomiędzy dwoma procesami, które wymieniają się wiadomościami, pozwala aby proces wysyłał wiadomość do jednego procesu a odbierał od jeszcze innego. W sumie jednak każdy z procesów, wywołujący `MPI_Sendrecv`, wysyła jedną wiadomość i odbiera jedną wiadomość.

Inną odmianą tej funkcji jest `MPI_Sendrecv_replace`, której składnia jest następująca:

```
int MPI_Sendrecv_replace( void *buf, int count,
                          MPI_Datatype datatype,
                          int dest, int sendtag,
                          int source, int recvtag,
                          MPI_Comm comm, MPI_Status *status )
```

void *buf – [IN] Adres początkowy bufora z danymi do wysłania, który jednocześnie jest adresem początkowym bufora do zapisu odebranych danych. Może to być adres pojedynczej zmiennej lub początek tablicy.
int count – [IN] Długość bufora `buf`. Dla pojedynczej zmiennej będzie to wartość 1.
MPI_Datatype datatype – [IN] Typ pojedynczego elementu bufora `buf`.
int dest – [IN] Identyfikator procesu, do którego wysyłany jest komunikat.
int sendtag – [IN] Znacznik wiadomości wysyłanej.
int source – [IN] Identyfikator procesu, od którego odbierany jest komunikat.
int recvtag – [IN] Znacznik wiadomości odbieranej.
MPI_Comm comm – [IN] Komunikator.
MPI_Status *status – [OUT] Zmienna, w której zapisywany jest status przesłanej wiadomości.

Różni się ona od `MPI_Sendrecv` tym, że nie wymaga dodatkowego bufora na odebraną wiadomość. W miejsce danych, które zostały wysłane, zapisywane są dane, które zostały odebrane, oczywiście kasując poprzednią zawartość.

5.3. Synchronizacja procesów MPI – funkcja `MPI_Barrier`

W programach MPI często będą takie miejsca, w których konieczna będzie synchronizacja procesów. Będziemy wymagać aby wszystkie procesy skończyły pewien etap pracy zanim przejdą do następnej części.

Rozważmy pewien przykład (listing 5.7):

Listing 5.7. Funkcja MPI_Barrier.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "mpi.h"
4
5 void printArray(int id, int *array, int size){
6     int i;
7
8     printf("␣%d␣:␣", id);
9     for(i=0; i<size; ++i)
10        printf("␣%d␣", array[i]);
11    printf("\n");
12
13    return;
14 }
15
16 int main(int argc, char **argv)
17 {
18    int myid, numprocs;
19    int i;
20
21    int buf1[10]={0}, buf2[10]={0};
22
23    MPI_Init(&argc, &argv);
24    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
25    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
26
27
28    for(i=0; i<10; ++i)
29        buf1[i] = myid;
30
31    for(i=0; i<10; ++i)
32        buf2[i] = 10 + myid;
33
34    printArray(myid, buf1, 10);
35
36    printArray(myid, buf2, 10);
37
38    MPI_Finalize();
39
40    return 0;
41 }
```

Założeniem w tym programie było, aby każdy proces, po tym jak ustawi wartości dwóch tablic, wyświetlił je na ekran. Przykładowym wynikiem działania tego programu jest poniższy wydruk.

```
0 :  0  0  0  0  0  0  0  0  0  0
2 :  2  2  2  2  2  2  2  2  2  2
2 :  12  12  12  12  12  12  12  12  12  12
1 :  1  1  1  1  1  1  1  1  1  1
3 :  3  3  3  3  3  3  3  3  3  3
0 :  10  10  10  10  10  10  10  10  10  10
1 :  11  11  11  11  11  11  11  11  11  11
3 :  13  13  13  13  13  13  13  13  13  13
```

Załóżmy jednak, że zależy nam na bardziej przejrzystym wydruku na ekranie. Niech zatem najpierw każdy z procesów wyświetli na ekranie pierwszą tablicę, potem jeden z procesów wyświetli linię rozdzielającą, a potem każdy proces wyświetli drugą tablicę. Gwarancję takiego efektu uzyskać możemy dzięki mechanizmowi bariery, który czytelnik miał już okazję poznać w rozdziale o OpenMP. Barierę w MPI realizujemy poprzez wywołanie funkcji `MPI_Barrier`.

Składnia tej funkcji jest następująca.

```
int MPI_Barrier ( MPI_Comm comm )
```

MPI_Comm comm – [IN] Komunikator.

Funkcja ta blokuje wszystkie procesy do momentu, aż zostanie wywołana przez każdy z procesów. Jest to takie miejsce w programie, do którego musi dojść najwolniejszy z procesów, zanim wszystkie ruszą dalej.

Rozważmy zatem następującą modyfikację kodu z listingu 5.7.

```
34      printArray(myid, buf1, 10);
35
36      MPI_Barrier(MPI_COMM_WORLD);
37
38      if (myid==0) printf("----------\n");
39
40      MPI_Barrier(MPI_COMM_WORLD);
41
42      printArray(myid, buf2, 10);
```

Poniżej przedstawiono przykładowy wydruk programu po takiej modyfikacji. Bariery pozwoliły nam rozdzielić trzy kolejne bloki kodu.

```
0 :  0  0  0  0  0  0  0  0  0  0
3 :  3  3  3  3  3  3  3  3  3  3
1 :  1  1  1  1  1  1  1  1  1  1
2 :  2  2  2  2  2  2  2  2  2  2
----------
3 :  13  13  13  13  13  13  13  13  13  13
0 :  10  10  10  10  10  10  10  10  10  10
2 :  12  12  12  12  12  12  12  12  12  12
1 :  11  11  11  11  11  11  11  11  11  11
```

Niestety w naszym przykładowym programie możliwy jest również i taki wydruk:

```
0 :  0  0  2 :  2  2  2  2  2  2  2  2  2  2
0  1 :  1  1  1  1  1  1  1  1  1  1
3 :  3  3  3  3  3  3  3  3  3  3
0  0  0  0  0  0  0
---------------------------------
0 :  10  10  10  10  10  10  10  10  10  10
1 :  2 :  12  12  12  12  12  12  12  12  12  12
11  11  11  11  11  11  3 :  13  13  13  13  13  13  13  13  13  13
11  11  11  11
```

Dzieje się tak dlatego, że przeplot procesów może nastąpić w trakcie wykonywania operacji wyświetlania w funkcji `printArray`.

Aby tego uniknąć musielibyśmy wymusić aby wyświetlanie wewnątrz funkcji `printArray` odbywało się sekwencyjnie, czyli aby jej wywołanie było swego rodzaju sekcją krytyczną. W MPI nie mamy do tego specjalnych mechanizmów i jesteśmy zmuszeni zastosować własne rozwiązania programistyczne. Dokonamy zatem modyfikacji funkcji `printArray` tak, aby wykluczyć przeplot.

Na listingu 5.8 zamieszczono modyfikację programu 5.7. Najważniejszym elementem tej modyfikacji jest usunięcie funkcji `printArray` i użycie w jej miejsce funkcji `safePrintArray`. Wewnątrz tej funkcji widzimy pętlę. W każdym obrocie tylko jeden z procesów wyświetla dane na ekran, a pozostałe czekają na barierze. Jednocześnie funkcja ta wymusza, aby procesy wyświetlały swoją tablicę po kolei, począwszy od procesu 0, co ułatwi nam porównywanie wydruków na ekranie.

Listing 5.8. Serializacja wyświetlania danych na ekranie.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "mpi.h"
4
5  void safePrintArray(int id, int nprocs, int *array,
6                                                  int size){
7      int i, n;
8      for(n=0; n<nprocs; ++n){
9
10     if(n == id){
11         printf("␣%d␣:␣", id);
12         for(i=0; i<size; ++i)
13         printf("␣%d␣", array[i]);
14         printf("\n");
15
16         if (id==nprocs-1) printf("\n");
17     }
```

```
18
19      MPI_Barrier(MPI_COMM_WORLD);
20      }
21 }
22
23 int main(int argc, char **argv)
24 {
25      int myid, numprocs;
26      int i;
27
28      int buf1[10]={0}, buf2[10]={0};
29
30      MPI_Init(&argc, &argv);
31      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
32      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
33
34      for(i=0; i<10; ++i)
35          buf1[i] = myid;
36
37      for(i=0; i<10; ++i)
38          buf2[i] = 10 + myid;
39
40      safePrintArray(myid, numprocs, buf1, 10);
41
42      MPI_Barrier(MPI_COMM_WORLD);
43
44      safePrintArray(myid, numprocs, buf2, 10);
45
46      MPI_Finalize();
47
48      return 0;
49 }
```

W dalszej części tego rozdziału funkcja `safePrintArray` będzie często używana.

5.4. Komunikacja grupowa – funkcje `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`

Jeśli jakaś dana ma być przesłana z jednego procesu do wszystkich pozostałych, wówczas używanie komunikacji typu punkt-punkt nie jest wydajne, zarówno pod względem zwięzłości kodu, jak i czasu działania takiej operacji, która wiązałaby się z wielokrotnym wywołanie funkcji wysyłającej i odbierającej. W takiej sytuacji należy używać funkcji specjalnie opracowanych do komunikacji grupowej. Do rozesłania pojedynczej danej lub tablicy danych z jednego procesu do wszystkich pozostałych posłuży nam funkcja `MPI_Bcast`.

Składnia tej funkcji jest następująca:

```
int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm )
```

void *buffer – [IN/OUT] Na procesie `root`, adres początkowy bufora z danymi. Na pozostałych procesach, adres początkowy bufora gdzie dane mają być zapisane. Może to być adres pojedynczej zmiennej lub początek tablicy.
int count – [IN] Długość bufora `buffer`. Dla pojedynczej zmiennej będzie to wartość 1.
MPI_Datatype datatype – [IN] Typ pojedynczego elementu bufora.
int root – [IN] Identyfikator procesu, od którego rozsyłany jest komunikat.
MPI_Comm comm – [IN] Komunikator.

`MPI_Bcast` rozsyła wiadomość zamieszczoną w tablicy `buffer` procesu `root` do wszystkich pozostałych procesów.

Na listingu 5.9 przedstawiono przykład użycia funkcji `MPI_Bcast`.

Listing 5.9. Funkcja `MPI_Bcast`.

```
1   int myid, numprocs;
2   int root;
3
4   int buf[10]={0};
5
6   ...
7
8   if (myid == 0)
9       for(i=0; i<10; ++i)
10          buf[i] = i;
11
12  safePrintArray(myid, numprocs, buf, 10);
13
14  root = 0;
15
16  MPI_Bcast(buf, 10, MPI_INT, root, MPI_COMM_WORLD);
17
18  safePrintArray(myid, numprocs, buf, 10);
```

Proces numer 0 wypełnia tablicę wartościami a następnie tablica ta jest rozesłana do wszystkich pozostałych procesów.

Wynik działania programu zamieszczono na wydruku poniżej.

```
0 :  0  1  2  3  4  5  6  7  8  9
1 :  0  0  0  0  0  0  0  0  0  0
2 :  0  0  0  0  0  0  0  0  0  0
3 :  0  0  0  0  0  0  0  0  0  0
```

```
0 :  0  1  2  3  4  5  6  7  8  9
1 :  0  1  2  3  4  5  6  7  8  9
2 :  0  1  2  3  4  5  6  7  8  9
3 :  0  1  2  3  4  5  6  7  8  9
```

Jednym z ważniejszych zastosowań obliczeń równoległych jest wykonanie operacji, której argumentem jest tablica wartości (lub tablice), a jej wynikiem pojedyncza liczba (lub pojedyncza tablica). Praca związana z taką operacją może zostać podzielona pomiędzy procesy. Każdy proces wyznacza wynik częściowy. Następnie wyniki częściowe są scalane (redukowane) do wyniku końcowego.

Przykładem może być obliczenie iloczynu skalarnego wektorów, znalezienie wartości minimalnej lub maksymalnej spośród elementów tablicy itp.

Operacje redukcji mają duże znaczenie w obliczeniach równoległych, stąd też większość standardów udostępnia specjalne rozwiązania do ich realizacji. W MPI do tego celu stosowana jest funkcja `MPI_Reduce`.

Jej składnia jest następująca:

```
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,
                 MPI_Datatype datatype, MPI_Op op, int root,
                 MPI_Comm comm )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi do redukcji. Może to być adres pojedynczej zmiennej lub początek tablicy.

void *recvbuf – [OUT] Adres początkowy bufora do zapisu redukowanych danych. Może to być adres pojedynczej zmiennej lub początek tablicy. Istotny tylko dla procesu `root`.

int count – [IN] Długość bufora `sendbuf`. Dla pojedynczej zmiennej będzie to wartość 1.

MPI_Datatype datatype – [IN] Typ pojedynczego elementu buforów.

MPI_Op op – [IN] Operator redukcji.

int root – [IN] Identyfikator procesu, do którego dane zostaną zredukowane.

MPI_Comm comm – [IN] Komunikator.

Funkcje do komunikacji grupowej charakteryzują się tym, że wywoływane są przez wszystkie procesy, ale niektóre parametry istotne są tylko dla procesu `root`. Tak jest w przypadku tablicy `recvbuf`. W wyniku wywołania funkcji `MPI_Reduce`, dane zapisywane są tylko w tablicy `recvbuf` procesu 0. Nie jest też konieczna alokacja tej tablicy dla pozostałych procesów.

Prosty przykład użycia funkcji `MPI_Reduce` przedstawiono na listingu 5.10.

Listing 5.10. Funkcja MPI_Reduce.

```
1   int myid, numprocs;
2   int root;
3   int buf[10];
4   int reducebuf[10]={0};
5
6   ...
7
8   srand(myid*time(NULL));
9
10  for(i=0; i<10; ++i)
11      buf[i] = random()%10;
12
13  safePrintArray(myid, numprocs, buf, 10);
14
15  if (myid==0) printf("\n");
16  MPI_Barrier(MPI_COMM_WORLD);
17
18  root = 0;
19  MPI_Reduce(buf, reducebuf, 10, MPI_INT, MPI_MAX, root,
20             MPI_COMM_WORLD);
21
22  safePrintArray(myid, numprocs, reducebuf, 10);
```

Każdy z procesów wypełnia własną tablicę `buf` losowymi danymi. Po wywołaniu funkcji `MPI_Reduce` w tablicy `reducebuf` procesu 0 znajdą się maksymalne wartości spośród elementów wszystkich tablic o identycznych indeksach. Przykładowe wyniki zamieszczono poniżej.

```
0 :  3  6  7  5  3  5  6  2  9  1
1 :  6  4  3  4  2  3  9  0  9  8
2 :  7  5  2  7  9  7  4  6  7  2
3 :  4  4  0  0  7  6  2  4  2  0

0 :  7  6  7  7  9  7  9  6  9  8
1 :  0  0  0  0  0  0  0  0  0  0
2 :  0  0  0  0  0  0  0  0  0  0
3 :  0  0  0  0  0  0  0  0  0  0
```

Pewną odmianą funkcji `MPI_Reduce` jest funkcja `MPI_Allreduce`, która różni się tym, że wyniki są scalane i zamieszczane w pamięci każdego procesu.

Składnia funkcji `MPI_Allreduce` jest następująca:

```
int MPI_Allreduce ( void *sendbuf, void *recvbuf, int count,
                    MPI_Datatype datatype, MPI_Op op,
                    MPI_Comm comm )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi do wysłania. Może to być adres pojedynczej zmiennej lub początek tablicy.
void *recvbuf – [OUT] Adres początkowy bufora do zapisu odebranych danych. Może to być adres pojedynczej zmiennej lub początek tablicy.
int count – [IN] Długość bufora `sendbuf`. Dla pojedynczej zmiennej będzie to wartość 1.
MPI_Datatype datatype – [IN] Typ pojedynczego elementu buforów.
MPI_Op op – [IN] Operator redukcji.
MPI_Comm comm – [IN] Komunikator.

Funkcja ta nie posiada parametru `root`, który w `MPI_Reduce` określał miejsce przesłania wyniku redukcji. Wywołanie funkcji `MPI_Allreduce` dla programu z listingu 5.10 przyjmie następującą postać.

```
1       MPI_Allreduce(buf, reducebuf, 10, MPI_INT, MPI_MAX,
2                     MPI_COMM_WORLD);
```

Zmienią się też wyniki programu. Każdy z procesów posiada taki sam zestaw danych w tablicy `reducebuf`.

```
0 :  3  6  7  5  3  5  6  2  9  1
1 :  1  8  5  2  8  5  8  1  8  2
2 :  9  9  0  0  4  4  6  9  7  2
3 :  4  9  7  6  4  7  5  8  7  8

0 :  9  9  7  6  8  7  8  9  9  8
1 :  9  9  7  6  8  7  8  9  9  8
2 :  9  9  7  6  8  7  8  9  9  8
3 :  9  9  7  6  8  7  8  9  9  8
```

Funkcja `MPI_Allreduce` daje takie same rezultaty jak wywołanie `MPI_Reduce` a następnie rozesłanie wyniku redukcji przy pomocy `MPI_Bcast`.

Na listingu 5.11 zamieszczono przykład zastosowania operacji rozsyłania i redukcji dla wyznaczenie poniższej całki metodą trapezów.

$$\int_0^1 (\frac{4}{1+x^2})dx$$

Listing 5.11. Program MPI – Całkowanie metodą trapezów

```
1 #include <stdio.h>
2 #include <math.h>
3 #include "mpi.h"
4
5 double f(double x){
6     return 4.0/(1.0+x*x);
7 }
```

```
 8
 9 int main(int argc, char **argv)
10 {
11
12     int myid, numprocs, root=0;
13     int i,n;
14
15     double pi,h,sum,x;
16
17     MPI_Init(&argc, &argv);
18     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
19     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
20
21
22     if (myid == root)
23     {
24         printf("Wprowadź liczbę przedziałów całkowania: ");
25         scanf("%d",&n);
26     }
27
28     MPI_Bcast(&n, 1, MPI_INT, root, MPI_COMM_WORLD);
29
30         h=1.0/(double)n;
31
32     sum=0.0;
33     for (i=myid; i<n; i+=numprocs)
34     {
35             x=h*((double)i+0.5);
36         sum+=f(x);;
37     }
38
39     MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, root,
40                MPI_COMM_WORLD);
41
42     if (myid == root){
43         pi = h*pi;
44         printf("pi=%.14f\n",pi);
45     }
46
47     MPI_Finalize();
48
49     return 0;
50 }
```

Liczona całka daje w wyniku wartość π co możemy zobaczyć na przykładowym wydruku.

```
Wprowadź liczbę przedziałów całkowania: 8000000
pi=3.14159265358975
```

5.5. Pomiar czasu wykonywania programów MPI

Celem zrównoleglania programów jest uzyskanie przyspieszenia dla nich. Dzięki przyspieszeniu obliczeń pewne problemy mogą być rozwiązywane w krótszym czasie, inne mogą być wykonywane dla większych zestawów danych. Nieodłącznym elementem wykonywania różnego typu symulacji jest pomiar czasu ich działania. W programach MPI pomocna może się zatem okazać funkcja `MPI_Wtime`.

Jest to jedna z niewielu funkcji MPI, która nie zwraca kodu błędu. Składnia funkcji `MPI_Wtime` jest następująca:

```
double MPI_Wtime()
```

Zwraca ona czas w sekundach jaki upłynął od pewnej ustalonej daty z przeszłości.

Na listingu 5.12 zamieszczono przykład użycia funkcji `MPI_Wtime` do zmierzenia pewnego sztucznie wygenerowanego opóźnienia programu poprzez funkcję `sleep`.

Listing 5.12. Pomiar czasu metodą `MPI_Wtime`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "mpi.h"
4
5  int main( int argc, char *argv[] )
6  {
7      double t_start, t_stop;
8
9      MPI_Init(&argc, &argv);
10
11     t_start = MPI_Wtime();
12     sleep(10);
13     t_stop = MPI_Wtime();
14
15     printf("10 sekund uśpienia zmierzone przez MPI_Wtime:");
16     printf(" %1.2f\n",t_stop-t_start);
17
18     MPI_Finalize( );
19
20     return 0;
21 }
```

Wynik działania programu:

```
10 sekund uśpienia zmierzone przez MPI_Wtime: 10.00
10 sekund uśpienia zmierzone przez MPI_Wtime: 10.00
10 sekund uśpienia zmierzone przez MPI_Wtime: 10.00
```

Na powyższym przykładzie zmierzony czas był jednakowy dla wszystkich procesów, co jednak nie jest często występującą sytuacją w rzeczywistych problemach. Jeśli każdy proces pracował różną ilość czasu, to czas działania programu lub jego fragmentu będzie równy czasowi najwolniejszego z procesów. Jeden ze sposobów na zmierzenie globalnego czasu przy różnych czasach dla poszczególnych procesów przedstawiono na przykładzie 5.13.

Listing 5.13. Pomiar maksymalnego czasu działania programu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include "mpi.h"
5
6  int main( int argc, char *argv[] )
7  {
8      double t_start, t_stop, t, t_max;
9      int myid;
10
11     MPI_Init(&argc, &argv);
12     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
13
14     srand(myid*time(NULL));
15
16     t_start = MPI_Wtime();
17     sleep(random()%10);
18     t_stop = MPI_Wtime();
19
20     t = t_stop-t_start;
21
22     printf("Czas uśpienia zmierzony przez proces %d: ",myid);
23     printf("%1.2f\n", t);
24
25     MPI_Reduce(&t, &t_max, 1, MPI_DOUBLE, MPI_MAX, 0,
26                MPI_COMM_WORLD);
27
28     if (myid == 0)
29         printf("Maksymalny czas działania: %1.2f\n", t_max);
30
31     MPI_Finalize( );
32
33     return 0;
34 }
```

Wynik działania programu dla ośmiu procesów:

```
Czas uśpienia zmierzony przez proces 4: 0.00
Czas uśpienia zmierzony przez proces 1: 1.00
Czas uśpienia zmierzony przez proces 0: 3.00
Czas uśpienia zmierzony przez proces 7: 5.00
```

```
Czas uśpienia zmierzony przez proces 3: 7.00
Czas uśpienia zmierzony przez proces 5: 7.00
Czas uśpienia zmierzony przez proces 2: 8.00
Czas uśpienia zmierzony przez proces 6: 9.00
Maksymalny czas działania: 9.00
```

Innym rozwiązaniem jest wstawienie bariery, potem pierwsze wywołanie funkcji `MPI_Wtime` na jednym z procesów, na końcu obliczeń, których czas chcemy zmierzyć, następnie druga bariera i drugi pomiar czasu na tym samym procesie (listing 5.14).

Listing 5.14. Pomiar maksymalnego czasu działania programu

```
1   double t_start, t_stop, t;
2   int myid;
3
4   ...
5
6   srand(myid*time(NULL));
7
8   MPI_Barrier(MPI_COMM_WORLD);
9
10  if (myid==0)
11      t_start = MPI_Wtime();
12
13  sleep(random()%10);
14
15  MPI_Barrier(MPI_COMM_WORLD);
16  if (myid==0){
17      t_stop = MPI_Wtime();
18      t = t_stop-t_start;
19      printf("Czas uśpienia zmierzony przez proces %d: ",
20              myid);
21      printf("%1.2f\n", t);
22  }
```

Przykładowy wynik dla czterech procesów:

```
Czas uśpienia zmierzony przez proces 0: 9.00
```

Powyższe metody posłużą nam do mierzenia fragmentów programu. Jeżeli chcielibyśmy zmierzyć czas działania całego programu, możemy użyć systemowego polecenia `time`, np.:

```
time mpirun -np 4 ./program
```

Przykładowy wynik dla przykładu 5.14:

```
Czas uśpienia zmierzony przez proces 0: 9.00

real    0m12.184s
user    0m0.353s
sys     0m0.129s
```

5.6. Komunikacja grupowa – MPI_Scatter, MPI_Gather, MPI_Allgather, MPI_Alltoall

W tej części przedstawimy kilka kolejnych funkcji realizujących komunikację grupową. Pierwsza z nich to `MPI_Scatter`. Służy ona do rozdzielenia danych na części i rozesłania poszczególnych części do każdego procesu.

Składnia funkcji `MPI_Scatter` jest następująca:

```
int MPI_Scatter ( void *sendbuf, int sendcnt,
                  MPI_Datatype sendtype,
                  void *recvbuf, int recvcnt,
                  MPI_Datatype recvtype,
                  int root, MPI_Comm comm )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi do rozdzielenia i rozesłania pomiędzy wszystkie procesy. Ma znaczenie tylko dla procesu `root`.

int sendcnt – [IN] Liczba elementów wysłanych do każdego procesu. Ma znaczenie tylko dla procesu `root`.

MPI_Datatype sendtype – [IN] Typ elementów bufora `sendbuf`. Ma znaczenie tylko dla procesu `root`.

void *recvbuf – [OUT] Adres początkowy bufora do zapisu rozdzielonych danych.

int recvcnt – [IN] Liczba elementów w `recvbuf`.

MPI_Datatype recvtype – [IN] Typ elementów bufora `recvbuf`.

int root – [IN] Identyfikator procesu, z którego dane zostaną rozdzielone.

MPI_Comm comm – [IN] Komunikator.

Bufor z danymi znajduje się w tablicy `sendbuf` na jednym z procesów, tzw. procesie `root`, a po wywołaniu `MPI_Scatter` każdy z procesów, łącznie z procesem `root`, ma zapisany fragment bufora `sendbuf` w tablicy `recvbuf`. Liczba wysyłanych elementów, jak również ich typ specyfikowane są zarówno po stronie wysyłającej, jak i odbierającej. Wynika to z tego, że dane po obu stronach mogą być reprezentowane w innym typie. Ważne aby zgadzał się sumaryczny rozmiar danych przesłanych i odebranych. Więcej o typach czytelnik przeczytać może w rozdziale 6. W wielu programach parametry

`sendcnt` oraz `recvcnt` a także `sendtype` oraz `recvtype` będą przyjmować takie same wartości. Takie uproszczenie przyjęto we wszystkich przykładach z niniejszego rozdziału.

Poniżej (listing 5.15) zamieszczono program ilustrujący działanie funkcji `MPI_Scatter`, a także przykładowy wynik jego działania.

Listing 5.15. Funkcja `MPI_Scatter`.

```
1
2       int myid, numprocs;
3       int root;
4       int *sendbuf, recvbuf[10]={0};
5
6       ...
7
8       if (myid == 0){
9           sendbuf = malloc(numprocs*10*sizeof(int));
10
11          for(i=0; i<numprocs*10; ++i)
12          sendbuf[i] = i+1;
13      }
14
15      root = 0;
16
17      if(myid == 0){
18          printf("␣%d␣:␣", myid);
19          for(i=0; i<numprocs*10; ++i)
20          printf("␣%d␣", sendbuf[i]);
21          printf("\n");
22      }
23
24      safePrintArray(myid, numprocs, recvbuf, 10);
25
26      MPI_Scatter(sendbuf, 10, MPI_INT, recvbuf, 10, MPI_INT,
27                  root, MPI_COMM_WORLD);
28
29      safePrintArray(myid, numprocs, recvbuf, 10);
30
31      if (myid == 0){
32          free(sendbuf);
33      }
```

```
0 :  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18
19  20 21 22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
37  38  39  40
0 :  0  0  0  0  0  0  0  0  0  0
1 :  0  0  0  0  0  0  0  0  0  0
2 :  0  0  0  0  0  0  0  0  0  0
3 :  0  0  0  0  0  0  0  0  0  0
```

```
0 :  1  2  3  4  5  6  7  8  9  10
1 :  11  12  13  14  15  16  17  18  19  20
2 :  21  22  23  24  25  26  27  28  29  30
3 :  31  32  33  34  35  36  37  38  39  40
```

Operacja odwrotna do MPI_Scatter może zostać realizowana przy pomocy funkcji MPI_Gather.

Jej składnia jest następująca:

```
int MPI_Gather ( void *sendbuf, int sendcnt,
                 MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt,
                 MPI_Datatype recvtype,
                 int root, MPI_Comm comm )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi przeznaczonymi do zebrania.
int sendcnt – [IN] Liczba elementów w sendbuf.
MPI_Datatype sendtype – [IN] Typ elementów bufora sendbuf.
void *recvbuf – [OUT] Adres początkowy bufora do zapisu zebranych danych. Ma znaczenie tylko dla procesu root.
int recvcnt – [IN] Liczba elementów od pojedynczego procesu. Ma znaczenie tylko dla procesu root.
MPI_Datatype recvtype – [IN] Typ elementów bufora recvbuf. Ma znaczenie tylko dla procesu root.
int root – [IN] Identyfikator procesu zbierającego dane.
MPI_Comm comm – [IN] Komunikator.

Na każdym z procesów znajduje się bufor z danymi w postaci tablicy sendbuf. Po wywołaniu MPI_Gather dane ze wszystkich buforów sendbuf są zbierane i zamieszczane w tablicy recvbuf zdefiniowanej w procesie root. Podobnie jak w przypadku funkcji MPI_Scatter, liczba wysyłanych elementów oraz ich typ specyfikowane są zarówno po stronie wysyłającej, jak i odbierającej.

Na dwóch kolejnych listingach (5.16 oraz 5.17) zamieszczono programy ilustrujące działanie funkcji MPI_Gather. Różnica pomiędzy nimi jest tylko w alokacji bufora recvbuf. W pierwszym przykładzie tablica recvbuf tworzona jest dla wszystkich procesów, dla wszystkich też jest wyświetlana. W drugim przykładzie tablica ta tworzona jest tylko dla procesu root, ponieważ tylko dla niego ma ona znaczenie.

Listing 5.16. Funkcja `MPI_Gather`

```
int myid, numprocs;
int root;
int sendbuf[4], *recvbuf;

...

for(i=0; i<4; ++i)
    sendbuf[i] = myid+1;

root = 0;

recvbuf = malloc(numprocs*4*sizeof(int));
for(i=0; i<numprocs*4; ++i)
    recvbuf[i] = 0;

safePrintArray(myid, numprocs, sendbuf, 4);

MPI_Gather(sendbuf, 4, MPI_INT, recvbuf, 4, MPI_INT,
           root, MPI_COMM_WORLD);

safePrintArray(myid, numprocs, recvbuf, numprocs*4);

free(recvbuf);
```

Przykładowy wynik działania programu dla czterech procesów:

```
0 :  1  1  1  1
1 :  2  2  2  2
2 :  3  3  3  3
3 :  4  4  4  4

0 :  1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4
1 :  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
2 :  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
3 :  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Listing 5.17. Funkcja `MPI_Gather`

```
int myid, numprocs;
int sendbuf[4], *recvbuf;
int root;

...

for(i=0; i<4; ++i)
    sendbuf[i] = myid+1;

if(myid == 0){
    recvbuf = malloc(numprocs*4*sizeof(int));
```

```
12          for(i=0; i<numprocs*4; ++i)
13              recvbuf[i] = 0;
14      }
15
16      root = 0;
17
18      safePrintArray(myid, numprocs, sendbuf, 4);
19
20      MPI_Gather(sendbuf, 4, MPI_INT, recvbuf, 4, MPI_INT,
21                 root, MPI_COMM_WORLD);
22
23      if(myid == 0){
24          printf("␣%d␣:␣", myid);
25          for(i=0; i<numprocs*4; ++i)
26          printf("␣%d␣", recvbuf[i]);
27          printf("\n");
28      }
29
30      if(myid == 0){
31          free(recvbuf);
32      }
```

Przykładowy wynik działania programu dla czterech procesów:

```
0 :  1  1  1  1
1 :  2  2  2  2
2 :  3  3  3  3
3 :  4  4  4  4

0 :  1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4
```

Istnieje też funkcja `MPI_Allgather`, która różni się od `MPI_Gather` praktycznie tym samym co `MPI_Allreduce` od `MPI_Reduce`.

Składnia funkcji `MPI_Allgather` jest następująca:

```
int MPI_Allgather ( void *sendbuf, int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf, int recvcount,
                    MPI_Datatype recvtype,
                    MPI_Comm comm )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi przeznaczonymi do zebrania.
int sendcnt – [IN] Liczba elementów w `sendbuf`.
MPI_Datatype sendtype – [IN] Typ elementów bufora `sendbuf`.
void *recvbuf – [OUT] Adres początkowy bufora do zapisu zebranych danych.
int recvcnt – [IN] Liczba elementów od pojedynczego procesu.

MPI_Datatype recvtype – [IN] Typ elementów bufora `recvbuf`.
MPI_Comm comm – [IN] Komunikator.

W odróżnieniu od funkcji `MPI_Gather`, zebrane dane ze wszystkich buforów `sendbuf` zamieszczane są nie na jednym procesie ale na wszystkich.

Na listingu 5.18 zamieszczono przykład ilustrujący sposób użycia funkcji `MPI_Allgather`.

Listing 5.18. Funkcja `MPI_Allgather`

```
1   int myid, numprocs;
2   int sendbuf[4], *recvbuf;
3
4   int i;
5
6   ...
7
8   for(i=0; i<4; ++i)
9       sendbuf[i] = myid+1;
10
11  recvbuf = malloc(numprocs*4*sizeof(int));
12  for(i=0; i<numprocs*4; ++i)
13      recvbuf[i] = 0;
14
15  safePrintArray(myid, numprocs, sendbuf, 4);
16
17  MPI_Allgather(sendbuf, 4, MPI_INT, recvbuf, 4, MPI_INT,
18                MPI_COMM_WORLD);
19
20  safePrintArray(myid, numprocs, recvbuf, numprocs*4);
21
22  free(recvbuf);
```

Przykładowy wynik działania programu:

```
0 :  1  1  1  1
1 :  2  2  2  2
2 :  3  3  3  3
3 :  4  4  4  4

0 :  1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4
1 :  1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4
2 :  1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4
3 :  1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4
```

Jeszcze jedną ciekawą funkcją, służącą do komunikacji grupowej, jest `MPI_Alltoall`. Jest ona swego rodzaju połączeniem funkcji `MPI_Scatter` oraz `MPI_Gather`. Każdy z procesów ma dwa bufory, jeden z danymi, drugi pusty. Każdy dzieli swoje dane na części i wysyła po jednej części do każdego,

sobie również zachowując jedną część. W wyniku działania tej funkcji, każdy proces ma po jednej części danych od wszystkich pozostałych plus jedną własną.

Składnia funkcji `MPI_Alltoall` jest następująca:

```
int MPI_Alltoall( void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf, int recvcnt,
                  MPI_Datatype recvtype,
                  MPI_Comm comm )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi do rozdzielenia i rozesłania pomiędzy wszystkie procesy. Każdy proces ma własny bufor do rozdzielenia.
int sendcnt – [IN] Liczba elementów wysłanych do każdego procesu.
MPI_Datatype sendtype – [IN] Typ elementów bufora `sendbuf`.
void *recvbuf – [OUT] Adres początkowy bufora do zapisu rozdzielonych i rozesłanych danych.
int recvcnt – [IN] Liczba elementów w `recvbuf`.
MPI_Datatype recvtype – [IN] Typ elementów bufora `recvbuf`.
MPI_Comm comm – [IN] Komunikator.

Na przykładzie 5.19 podano przykład ilustrujący w jaki sposób użyć funkcji `MPI_Alltoall` a poniżej wynik działania tego przykładu.

Listing 5.19. Funkcja `MPI_Alltoall`

```
1   int myid, numprocs;
2   int *sendbuf, *recvbuf;
3
4   ...
5
6   sendbuf = malloc(numprocs*4*sizeof(int));
7
8   for(i=0; i<numprocs*4; ++i)
9       sendbuf[i] = myid*4+(i/4);
10
11  safePrintArray(myid, numprocs, sendbuf, numprocs*4);
12
13  recvbuf = malloc(numprocs*4*sizeof(int));
14
15  MPI_Alltoall(sendbuf, 4, MPI_INT, recvbuf, 4, MPI_INT,
16               MPI_COMM_WORLD);
17
18  safePrintArray(myid, numprocs, recvbuf, numprocs*4);
19
20  free(sendbuf);
21  free(recvbuf);
```

```
0 :  0  0  0  0  1  1  1  1  2  2  2  2  3  3  3  3
1 :  4  4  4  4  5  5  5  5  6  6  6  6  7  7  7  7
2 :  8  8  8  8  9  9  9  9  10  10  10  10  11  11  11  11
3 :  12  12  12  12  13  13  13  13  14  14  14  14  15  15  15  15

0 :  0  0  0  0  4  4  4  4  8  8  8  8  12  12  12  12
1 :  1  1  1  1  5  5  5  5  9  9  9  9  13  13  13  13
2 :  2  2  2  2  6  6  6  6  10  10  10  10  14  14  14  14
3 :  3  3  3  3  7  7  7  7  11  11  11  11  15  15  15  15
```

5.7. Komunikacja grupowa – `MPI_Scatterv`, `MPI_Gatherv`

W funkcji `MPI_Scatter` następował podział pewnego ciągłego obszaru pamięci na części, z których każda była jednakowego rozmiaru. Każdy proces otrzymywał zatem tyle samo danych. Czasem zachodzi jednak potrzeba, aby dane były dzielone w inny sposób. Posłuży do tego funkcja `MPI_Scatterv`.

Jej składnia jest następująca:

```
int MPI_Scatterv ( void *sendbuf, int *sendcnts,
                   int *displs, MPI_Datatype sendtype,
                   void *recvbuf, int recvcnt,
                   MPI_Datatype recvtype,
                   int root, MPI_Comm comm )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi do rozdzielenia i rozesłania pomiędzy wszystkie procesy. Ma znaczenie tylko dla procesu `root`.

int *sendcnts – [IN] Tablica liczb całkowitych o rozmiarze takim jak liczba procesów. Określa ile elementów ma być przesłane do każdego z procesów.

int *displs – [IN] Tablica liczb całkowitych o rozmiarze takim jak liczba procesów. Określa przesunięcia w stosunku do `sendbuf`, dzięki czemu wiadomo gdzie zaczynają się dane dla poszczególnych procesów.

MPI_Datatype sendtype – [IN] Typ elementów bufora `sendbuf`.

void *recvbuf – [OUT] Adres początkowy bufora do zapisu rozdzielonych danych.

int recvcnt – [IN] Liczba elementów w `recvbuf`.

MPI_Datatype recvtype – [IN] Typ elementów bufora `recvbuf`.

int root – [IN] Identyfikator procesu, z którego dane zostaną rozdzielone.

MPI_Comm comm – [IN] Komunikator.

Parametr `sendcnt` z funkcji `MPI_Scatter` został tutaj zastąpiony dwoma tablicami: `sendcnts` oraz `displs`. Są to tablice liczb całkowitych o rozmiarze równym liczbie procesów. Pierwsza z nich określa ile elementów ma otrzymać

każdy proces. Pod indeksem 0 znajduje się informacja o liczbie elementów dla procesu 0, pod indeksem 1 dla procesu 1 itd. Druga tablica określa skąd mają być brane elementy dla poszczególnych procesów. Znajdują się tam wartości przesunięć w stosunku do `sendbuf`, dzięki czemu może być określony początek danych dla każdego z procesów. Analogicznie do tablicy `counts`, element o indeksie 0 jest dla procesu 0 itd.

Na listingu 5.20 zamieszczono przykład użycia `MPI_Scatterv`.

Listing 5.20. Funkcja `MPI_Scatterv`

```
1   int myid, numprocs;
2   int *sendbuf, recvbuf[10]={0};
3   int stride, *displs, *scounts;
4   int root, i;
5
6   ...
7
8   root = 0;
9   stride = 12;
10
11  if (myid == root){
12      sendbuf = malloc(numprocs*stride*sizeof(int));
13
14      for(i=0; i<numprocs*stride; ++i)
15      sendbuf[i] = i+1;
16
17      displs = malloc(numprocs*sizeof(int));
18      scounts = malloc(numprocs*sizeof(int));
19
20          for(i=0; i<numprocs; ++i){
21          displs[i] = i*stride;
22          scounts[i] = i+1;
23      }
24
25      printf("␣%d␣:␣", myid);
26      for(i=0; i<numprocs*stride; ++i)
27      printf("␣%d␣", sendbuf[i]);
28      printf("\n\n");
29  }
30
31  MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, recvbuf,
32               10, MPI_INT, root, MPI_COMM_WORLD);
33
34  safePrintArray(myid, numprocs, recvbuf, 10);
35
36  if (myid == root){
37      free(sendbuf);
38      free(displs);
39      free(scounts);
40  }
```

Przykładowy wynik działania programu:

```
0 :  1  2  3  4  5  6  7  8  9  10  11
12  13  14  15  16  17  18  19  20  21
22  23  24  25  26  27  28  29  30  31
32  33  34  35  36  37  38  39  40  41
42  43  44  45  46  47  48

0 :  1  0  0  0  0  0  0  0  0  0
1 :  13  14  0  0  0  0  0  0  0  0
2 :  25  26  27  0  0  0  0  0  0  0
3 :  37  38  39  40  0  0  0  0  0  0
```

Funkcją odwrotną do funkcji `MPI_Scatterv` jest `MPI_Gatherv`
Składnia funkcji `MPI_Gatherv` jest następująca:

```
int MPI_Gatherv ( void *sendbuf, int sendcnt,
                  MPI_Datatype sendtype,
                  void *recvbuf, int *recvcnts,
                  int *displs,
                  MPI_Datatype recvtype,
                  int root, MPI_Comm comm )
```

void *sendbuf – [IN] Adres początkowy bufora z danymi do zebrania.
int sendcnt – [IN] Liczba elementów w `sendbuf`.
MPI_Datatype sendtype – [IN] Typ elementów bufora `sendbuf`.
void *recvbuf – [OUT] Adres początkowy bufora do zapisu zebranych danych.
int *recvcnts – [IN] Tablica liczb całkowitych o rozmiarze takim jak liczba procesów. Określa ile elementów ma być odebrane od każdego z procesów.
int *displs – [IN] Tablica liczb całkowitych o rozmiarze takim jak liczba procesów. Określa przesunięcia w stosunku do `recvbuf`, dzięki czemu wiadomo gdzie mają być zapisywane dane od poszczególnych procesów.
MPI_Datatype recvtype – [IN] Typ elementów bufora `recvbuf`.
int root – [IN] Identyfikator procesu, do którego dane mają być zebrane.
MPI_Comm comm – [IN] Komunikator.

Przykład jej użycia oraz przykładowe wyniki zamieszczono poniżej.

Listing 5.21. Funkcja `MPI_Gatherv`

```
1   int myid, numprocs;
2   int sendbuf[10]={0}, *recvbuf;
3   int stride, *displs, *rcounts;
4   int root, i;
```

```
5       ...
6
7       root = 0;
8       stride = 12;
9
10      for(i=0; i<10; ++i)
11          sendbuf[i] = i+myid*10+1;
12
13      if (myid == root){
14          recvbuf = malloc(numprocs*stride*sizeof(int));
15          for(i=0; i<numprocs*stride; ++i)
16              recvbuf[i] = 0;
17
18          displs = malloc(numprocs*sizeof(int));
19          rcounts = malloc(numprocs*sizeof(int));
20
21          for(i=0; i<numprocs; ++i){
22              displs[i] = i*stride;
23              rcounts[i] = i+1;
24          }
25      }
26
27      safePrintArray(myid, numprocs, sendbuf, 10);
28
29      MPI_Gatherv(sendbuf, myid+1, MPI_INT, recvbuf, rcounts,
30                  displs, MPI_INT, root, MPI_COMM_WORLD);
31
32      if(myid == root){
33          printf("␣%d␣:␣", myid);
34          for(i=0; i<numprocs*stride; ++i)
35              printf("␣%d␣", recvbuf[i]);
36          printf("\n");
37      }
38
39      if (myid == root){
40          free(recvbuf);
41          free(displs);
42          free(rcounts);
43      }
```

```
0 :  1  2  3  4  5  6  7  8  9  10
1 :  11  12  13  14  15  16  17  18  19  20
2 :  21  22  23  24  25  26  27  28  29  30
3 :  31  32  33  34  35  36  37  38  39  40

0 :  1  0  0  0  0  0  0  0  0  0  0  0  11
12  0  0  0  0  0  0  0  0  0  0  21  22  23
0  0  0  0  0  0  0  0  0  31  32  33  34  0
0  0  0  0  0  0  0
```

5.8. Zadania

Poniżej zamieściliśmy szereg zadań do samodzielnego wykonania. Do ich rozwiązania należy wykorzystać bibliotekę MPI. Przy niektórych zadaniach zaznaczono, że do ich wykonania potrzebne będą jednocześnie MPI oraz OpenMP.

Zadanie 5.1.
Napisz program równoległy, w którym dwa procesy będą naprzemiennie przesyłać komunikaty do siebie, tzn. pierwszy proces wysyła komunikat do drugiego procesu, następnie drugi po odebraniu odsyła ten sam komunikat do pierwszego. Przesłanie i odebranie powinno być powtórzone ustaloną liczbę razy. W programie zaimplementuj następujące funkcjonalności:
a) Program działa tylko jeśli zostały uruchomione dwa procesy, w przeciwnym przypadku przerywa działanie z odpowiednim komunikatem.
b) Każdy komunikat ma być ciągłym fragmentem tablicy, tablica powinna być odpowiednio wcześniej zainicjowana, program powinien być przetestowany dla rożnej wielkości komunikatów.
e) Wykorzystując funkcję `MPI_Wtime` należy zmierzyć i wyświetlić średni czas przesłania każdego komunikat oraz wydajność komunikacji w B/s.

Zadanie 5.2.
Napisz program równoległy, który otrzyma na wejściu zbiór liczb o takim rozmiarze jak liczba działających procesów MPI, i który rozdzieli liczby w taki sposób, że każdy proces o mniejszym identyfikatorze będzie posiadał mniejszą liczbę. Liczby ma wczytywać proces 0, który sprawdza, czy wczytana liczba jest mniejsza od obecnie posiadanej. Liczbę większą przesyła do procesu o identyfikatorze o jeden większym. Każdy proces postępuje podobnie aż do rozesłania wszystkich liczb.

Zadanie 5.3.
Napisz program równoległy, w którym proces 0 tworzy dynamicznie tablicę elementów o rozmiarze 113. Wypełnia tę tablicę kolejnymi liczbami całkowitymi i rozsyła mniej więcej równą część do pozostałych procesów. Użyj tutaj blokującej komunikacji typu punkt-punk. Proces 0 sobie zostawia największą część. Każdy proces po podzieleniu wyświetla ile elementów otrzymał oraz wyświetla wszystkie elementy.

Zadanie 5.4.
Napisz program równoległy, w którym każdy proces inicjuje wartością swojego identyfikatora jakąś zmienną. Następnie procesy wymieniają się wartością tej zmiennej w następujący sposób: zerowy wysyła do pierwszego,

pierwszy do drugiego itd., ostatni proces wysyła do zerowego. Użyj tutaj funkcji `MPI_Sendrecv`.

Zadanie 5.5.

Napisz program równoległy, w którym proces 0 wczytuje z klawiatury 10 liczb, zamieszczając je w tablicy. Następnie rozsyła całą tę tablicę do pozostałych procesów.

Zadanie 5.6.

Napisz program równoległy, w którym każdy z 4 procesów wypełnia tablicę o wielkości 256 elementów losowymi wartościami całkowitymi z zakresu 0..15. Następnie tablice te powinny być zredukowane do procesu 3, tak aby utworzona w ten sposób tablica pod każdym indeksem zawierała najmniejsze elementy.

Zadanie 5.7.

Napisz program równoległy, liczący iloczyn skalarny dwóch wektorów liczb rzeczywistych. Proces 0 pobiera z wejścia (lub generuje) elementy obydwu wektorów. Następnie dzieli wektory pomiędzy procesy rozsyłając każdemu w przybliżeniu równą liczbę elementów obydwu wektorów (liczba elementów nie musi być podzielna przez liczbę procesów). Następnie cząstkowe sumy iloczynów odpowiadających sobie elementów są redukowane, a wynik kopiowany do wszystkich procesów. Proces 0 wyświetla iloczyn skalarny wektorów.

Zadanie 5.8.

Napisz program równoległy wyznaczający wartość liczby π metodą Monte Carlo.

Jeśli mamy koło oraz kwadrat opisany na tym kole to wartość liczby π można wyznaczyć ze wzoru.

$$\pi = 4\frac{P_{\bigcirc}}{P_{\square}}$$

We wzorze tym występuje pole koła, do czego potrzebna jest wartość liczby π. Sens metody Monte Carlo polega jednak na tym, że w ogóle nie trzeba wyznaczać pola koła. To co należy zrobić to wylosować odpowiednio dużą liczbę punktów należących do kwadratu i sprawdzić jaka ich część należy również do koła. Stosunek tej części do liczby wszystkich punktów będzie odpowiadał stosunkowi pola koła do pola kwadratu w powyższym wzorze.

Zadanie 5.9.

Opisz w postaci funkcji `int czyRowne(int a [], int b [], int n)` algorytm równoległy sprawdzający, czy wszystkie odpowiadające sobie elementy tablic **a** i **b** o rozmiarze n są równe.

Zadanie 5.10.

Opisz w postaci funkcji `int minIndeks(int a [], int M, int N, int wzor, int &liczba)` algorytm równoległy wyznaczający najwcześniejszy indeks wystąpienia wartości `wzor` w tablicy **a** wśród składowych **a**[**M**]..**a**[**N**]. Poprzez parametr wyjściowy `liczba` należy zwrócić liczbę wszystkich wystąpień wartości `wzor`.

Zadanie 5.11.

Napisz program równoległy, który będzie się wykonywał tylko wówczas jeśli liczba procesów będzie równa 2, 4 lub 8. W programie proces o numerze 1 definiuje tablicę `sendbuf` o rozmiarze 120 (typ `double`). Niech proces 1 wypełni tablicę `sendbuf` wartościami od 1.0 do 120.0. Następnie tablica `sendbuf` ma być rozdzielona po równo pomiędzy procesy. Każdy proces swoją część ma przechowywać w tablicy `recvbuf`. Tablica ta powinna być alokowana dynamicznie i mieć dokładnie taki rozmiar jak liczba elementów przypadająca na każdy z procesów. Po odebraniu swojej części każdy proces modyfikuje element tablicy `recvbuf` wyliczając w ich miejsce pierwiastek z danego elementu. Następnie tablice `recvbuf` mają być zebrane przez proces 1 i zapisane w tablicy `sendbuf`.

Zadanie 5.12.

Napisz program równoległy, w którym wykonasz następujące operacje. Program sprawdza, czy działa 4 procesy, jeśli nie to kończy działanie. Proces 0 definiuje tablicę `sendbuf` elementów typu int o rozmiarze 95. Następnie następuje rozproszenie tablicy `sendbuf` na wszystkie procesy w następujący sposób: proces 0 dostaje elementy o indeksach od 12 do 18, proces 1 od 20 do 37, proces 2 od 40 do 65, proces 3 od 70 do 94. Każdy proces swoją porcję danych odbiera do tablicy `recvbuf` (niekoniecznie alokowanej dynamicznie). Każdy z procesów wykonuje operację inkrementacji na elementach tablicy `recvbuf`. Następnie proces 0 zbiera od wszystkich procesów po 3 pierwsze elementy z tablicy `recvbuf` i zapisuje je na początku tablicy `sendbuf`.

Zadanie 5.13.

Napisz program równoległy, w którym wykonasz następujące operacje. Każdy z procesów definiuje tablicę `sendbuf` elementów typu całkowitego o rozmiarze 10. Każdy z procesów wypełnią tablicę losowymi wartościami całkowitymi z zakresu od 1 do 20. Następnie Każdy proces znajduje element minimalny swojej tablicy. Wartość ta jest redukowana do zmiennej `maxmin1` z operatorem `MPI_MAX` do procesu 0. Następnie każdy proces rozdziela swoją tablicę pomiędzy wszystkie procesy z wykorzystaniem funkcji `MPI_Alltoall`, w wyniku której na każdym procesie powstanie tablica `recvbuf`. Następnie każdy proces znajduje minimum z tablicy `recvbuf`, a wartość ta, podobnie jak poprzednio, redukowana jest do zmiennej `maxmin2` z operatorem `MPI_MAX` do procesu 0. Proces 0 wyświetla na ekranie komunikat czy `maxmin1` jest równy `maxmin2`.

Zadanie 5.14.

Napisz program równoległy, w którym wykonasz następujące operacje. Każdy z procesów definiuje tablicę `sendbuf` elementów typu całkowitego o rozmiarze 10. Każdy z procesów wypełnią tablicę losowymi wartościami całkowitymi z zakresu od -5 do 5. Następnie tworzona jest tablica `recvbuf` będąca złączeniem wszystkich tablic począwszy od procesu 0. Złączona tablica zamieszczana jest w każdym z procesów. Dodatkowo każdy z procesów ma posiadać tablicę `sumbuf` będącą wynikiem operacji redukcji na tablicach `sendbuf` z operatorem `MPI_SUM`.

Zadanie 5.15.

Zmodyfikuj przykład z listingu 5.17 tak aby to samo zrobione było bez użycia funkcji `MPI_Gather`.

Zadanie 5.16.

Zmodyfikuj przykład z listingu 5.18 tak aby to samo zrobione było bez użycia funkcji `MPI_Allgather`.

Zadanie 5.17.

Zmodyfikuj przykład z listingu 5.19 tak aby to samo zrobione było bez użycia funkcji `MPI_Alltoall`.

Zadanie 5.18.

Napisz program równoległy (MPI + OpenMP), w którym działać będzie tyle procesów, ile podane zostanie przy uruchomieniu programu (opcja -np polecenia mpirun). Jeden proces będzie procesem głównym (master). Zadaniem procesu master będzie odebranie wszystkich komunikatów od pozostałych procesów i wyświetlenie ich na ekranie. Zadaniem pozostałych

procesów (procesów typu slave) będzie wysłanie komunikatów do procesu głównego. W ramach każdego procesu typu slave powinno uruchomić się po 4 wątki. Każdy wątek, oprócz wątku głównego, tworzy komunikat zawierający numer procesu, w ramach którego działa oraz swój numer wątku. Wątek główny (wszystkie wątki główne z każdego procesu typu slave) wysyła tak utworzony komunikat do procesu master.

Zadanie 5.19.

Napisz program równoległy (MPI + OpenMP), w którym działać będzie tyle procesów MPI, ile podane zostanie przy uruchomieniu programu (opcja -np polecenia mpirun). Dodatkowo w ramach każdego procesu powinno uruchomić się tyle wątków, ile procesorów jest na komputerze, na którym dany proces działa. Każdy wątek powinien wyświetlić na ekranie komunikat zawierający numer procesu, w ramach którego działa oraz swój numer wątku. Program powinien również działać jeśli będzie kompilowany bez MPI, a także bez OpenMP, jak również bez obydwu z nich.

Rozdział 6

Message Passing Interface – techniki zaawansowane

Przedstawimy teraz zaawansowane mechanizmy MPI, które pozwolą nam na tworzenie programów realizujących bardziej złożone schematy komunikacji między procesami oraz umożliwią współbieżne wykonywanie obliczeń i komunikacji. Więcej informacji można znaleźć w książkach [47, 51, 56].

6.1. Typy pochodne

W przypadku prostej komunikacji między procesami w programie MPI, wywołania funkcji `MPI_Send` oraz `MPI_Recv` wykorzystują typy standardowe (na przykład `MPI_INT` lub `MPI_DOUBLE`), co umożliwia przesłanie w ramach pojedynczego komunikatu określonej liczby danych tego samego typu. W przypadku, gdy chcemy przesyłać dane niejednorodne lub zawrzeć w jednym komunikacie dane, które nie tworzą w pamięci zwartego obszaru, wygodnie jest posłużyć się typami pochodnymi oraz zaawansowanym mechanizmem pakowania i rozpakowywania wiadomości.

Typy pochodne mają postać ciągu par postaci

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

gdzie $type_i$ jest pewnym typem (standardowym lub pochodnym), zaś $disp_0$ jest przesunięciem względem początku bufora. Do tworzenia typów pochodnych wykorzystuje się tzw. konstruktory typów. Przesyłana wiadomość jest opisywana przez sygnaturę typu postaci

$$Typesig = \{type_0, ..., type_{n-1}\}.$$

Poniżej podajemy najważniejsze konstruktory typów pochodnych, które mają postać funkcji MPI.

6.1.1. Typ *Contiguous*

Najprostszym typem pochodnym jest *Contiguous*, który definiuje replikację pewnej liczby danych typu bazowego, które zajmują zwarty obszar pamięci. Jego konstruktor ma następującą postać[1]:

```
int MPI_Type_contiguous( int count, MPI_Datatype_oldtype,
                         MPI_Datatype *newtype )
```

int count – [IN] Liczba elementów (nieujemna wartość int).
MPI_Datatype_oldtype – [IN] Typ bazowy.
MPI_Datatype *newtype – [OUT] Nowy typ.

[1] W całym rozdziale przy opisie będziemy używać oznaczeń [IN] oraz [OUT] dla wskazania parametrów wejściowych i wyjściowych.

Listing 6.1 pokazuje prosty przykład użycia konstruktora typu pochodnego *contiguous* dla zdefiniowania nowego typu postaci $3 \times$ `MPI_DOUBLE`. W programie definiujemy nowy typ używając podanego wyżej konstruktora. Następnie wywołujemy funkcję `MPI_Type_commit` dla zatwierdzenia zdefiniowanego typu. W dalszym ciągu proces 0 wysyła jedną daną tego nowego typu do procesu numer 1. W komunikacji wykorzystujemy standardowe funkcje `MPI_Send` oraz `MPI_Recv` wskazując w trzecim parametrze typ wysyłanych danych. Pierwszy parametr obu wywołań ma wartość 1, co oznacza, że będzie przesyłana jedna dana zdefiniowanego typu.

Listing 6.1. Wykorzystanie konstruktora *Contiguous*

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <math.h>
4
5  int main( int argc, char *argv[])
6  {
7      int myid, numprocs, i;
8      MPI_Status stat;
9      MPI_Datatype typ;
10     double a[4];
11
12     MPI_Init(&argc,&argv);
13     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
14     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
15
16     // definiowanie i zatwierdzenie typu "3 x MPI_DOUBLE"
17     MPI_Type_contiguous(3,MPI_DOUBLE,&typ);
18     MPI_Type_commit(&typ);
19
20     if(myid==0){
21         a[0]=10.0; a[1]=20.0; a[2]=30.0; a[3]=40.0;
22
23         MPI_Send(&a[1],1,typ,1,100,MPI_COMM_WORLD);
24
25     }else{ // odbiera proces numer 1
26
27         a[0]=-1.0; a[1]=-1.0; a[2]=-1.0; a[3]=-1.0;
28
29         MPI_Recv(&a,1,typ,MPI_ANY_SOURCE,100,
30                                         MPI_COMM_WORLD,&stat);
31         // teraz a[0]==20, a[1]==30, a[2]==40, a[3]==-1
32     }
33     MPI_Finalize();
34     return 0;
35 }
```

6.1.2. Typ *Vector*

Typ pochodny *Vector* daje znacznie większe możliwości. Definiuje on sekwencję zwartych obszarów pamięci, których początki muszą być od siebie oddalone o taką samą odległość, liczoną w elementach danych typu bazowego. Przykładowe użycie tego typu pokazano na listingu 6.2. Zauważmy, że wysyłana jest jedna dana typu pochodnego *Vector*, zaś odbierane cztery dane typu `MPI_DOUBLE`.

```
int MPI_Type_vector( int count, int blocklength,
                     int stride, MPI_Datatype oldtype,
                     MPI_Datatype *newtype )
```

int count – [IN] Liczba bloków (nieujemna wartość int).
int blocklength – [IN] Liczba elementów w każdym bloku (nieujemna).
int stride – [IN] Liczba elementów pomiędzy początkami bloków.
MPI_Datatype oldtype – [IN] Typ bazowy.
MPI_Datatype *newtype – [OUT] Nowy typ.

Listing 6.2. Wykorzystanie konstruktora *Vector*

```
1  int myid, numprocs, i;
2  MPI_Status stat;
3  MPI_Datatype typ;
4  double a[6];
5  double b[4];
6  ...
7  // przesłanie "bloku macierzy"
8
9  MPI_Type_vector(2,2,3,MPI_DOUBLE,&typ);
10 MPI_Type_commit(&typ);
11
12 if(myid==0){
13
14    a[0]=11.0;    a[3]=12.0;
15    a[1]=21.0;    a[4]=22.0;
16    a[2]=31.0;    a[5]=32.0;
17
18    // wysłanie - q dana typu Vector (zmienna typ)
19    MPI_Send(a,1,typ,1,100,MPI_COMM_WORLD);
20
21 }else{
22    // odebranie: 4 x MPI_DOUBLE
23    MPI_Recv(b,4,MPI_DOUBLE,MPI_ANY_SOURCE,100,
24                                    MPI_COMM_WORLD,&stat);
25    // b[0]==11.0   b[2]==12.0
26    // b[1]==21.0   b[3]==22.0
27 }
```

6.1.3. Typ *Indexed*

Typ pochodny *Indexed* stanowi uogólnienie typu *Vector*. Definiuje on sekwencję bloków różnych długości. Dla każdego bloku oddzielnie definiuje się przemieszczenie względem początku bufora liczone w liczbie elementów danych typu podstawowego. Konstruktor ma następującą postać.

```
int MPI_Type_indexed( int count, int *array_of_bls,
                      int *array_of_dis, MPI_Datatype oldtype,
                      MPI_Datatype *newtype)
```

int count – [IN] Liczba bloków (nieujemna wartość int).
int *array_of_bls – [IN] Tablica długości bloków.
int *array_of_dis – [IN] Tablica przesunięć początków bloków.
MPI_Datatype oldtype – [IN] Typ bazowy.
MPI_Datatype *newtype – [OUT] Nowy typ.

Na listingu 6.3 pokazano przykład zastosowania typu pochodnego *Indexed* dla przesłania transpozycji danej macierzy. Zauważmy, że proces 0 alokuje w swojej pamięci tablicę przechowującą macierz 3×2, a następnie przesyła sześć wartości typu `MPI_DOUBLE`, które są pakowane do bufora według kolejności opisanej typem pochodnym *Indexed*. Proces 1 odbiera jedną daną typu pochodnego *Indexed*, który zawiera przechowywaną kolumnami macierz 2×3 będącą transpozycją macierzy przechowywanej przez wysyłający proces 0.

Listing 6.3. Wykorzystanie konstruktora *Indexed* do przesłania transpozycji macierzy

```
1     MPI_Datatype typ1;
2     double a[6];
3     int bsize[6], disp[6];
4
5     ...
6     bsize[0]=1; bsize[1]=1; bsize[2]=1;
7     bsize[3]=1; bsize[4]=1; bsize[5]=1;
8     disp[0]=0;  disp[1]=2;  disp[2]=4;
9     disp[3]=1;  disp[4]=3;  disp[5]=5;
10
11    MPI_Type_indexed(6,bsize,disp,MPI_DOUBLE,&typ1);
12    MPI_Type_commit(&typ1);
13
14    if(myid==0){
15
16        a[0]=11.0;  a[3]=12.0;
17        a[1]=21.0;  a[4]=22.0;
18        a[2]=31.0;  a[5]=32.0;
19
```

```
20          MPI_Send(a,6,MPI_DOUBLE,1,100,MPI_COMM_WORLD);
21
22      }else{
23
24          MPI_Recv(a,1,typ1,MPI_ANY_SOURCE,100,
25                                          MPI_COMM_WORLD,&stat);
26
27          // a[0]==11.0   a[2]==21.0   a[4]==31.0
28          // a[1]==12.0   a[3]==22.0   a[5]==32.0
29      }
```

Listing 6.4 ilustruje zastosowanie typu pochodnego *Indexed* dla przesłania dolnego trójkąta macierzy 3×3, którą proces 0 przechowuje kolumnami. Używamy do tego definicji typu zawierającego trzy bloki danych typu `MPI_DOUBLE` o długościach odpowiednio 3, 2 i 1, na które składają się części kolumn, każda rozpoczynająca się elementem na głównej przekątnej.

Listing 6.4. Wykorzystanie konstruktora *Indexed* do przesłania dolnego trójkąta macierzy

```
1       int myid, numprocs, i, j, n;
2       MPI_Status stat;
3       MPI_Datatype typ1;
4       double a[MAXN*MAXN];
5       int bsize[MAXN], disp[MAXN];
6
7       ...
8       n=MAXN;
9       for(i=0;i<n;i++){
10        bsize[i]=n-i;
11        disp[i]=i*MAXN+i;
12      }
13
14      MPI_Type_indexed(n,bsize,disp,MPI_DOUBLE,&typ1);
15      MPI_Type_commit(&typ1);
16
17      if(myid==0){
18
19          for(j=0;j<n;j++)
20              for(i=0;i<n;i++)
21                  a[j*MAXN+i]=10.0*(i+1)+j+1;
22
23          MPI_Send(a,1,typ1,1,100,MPI_COMM_WORLD);
24
25      }else{
26
27          for(j=0;j<n;j++)
28              for(i=0;i<n;i++)
29                  a[j*MAXN+i]=0.0;
30
```

```
31          MPI_Recv(a,1,typ1,MPI_ANY_SOURCE,100,
32                                          MPI_COMM_WORLD,&stat);
33          for(i=0;i<n;i++){
34              for(j=0;j<n;j++)
35                printf("%10.2lf ",a[i+j*MAXN]);
36              printf("\n");
37          }
38      }
```

6.1.4. Typ *Struct*

Typ pochodny *Struct* stanowi uogólnienie omówionych wyżej typów pochodnych. Składa się z sekwencji bloków różnej długości, z których każdy blok może zawierać dane innego typu.

```
int MPI_Type_struct( int count, int *array_of_bls,
                     int *array_of_dis,
                     MPI_Datatype *array_of_types,
                     MPI_Datatype *newtype )
```

int count – [IN] Liczba bloków (nieujemna wartość int).
int *array_of_bls – [IN] Tablica długości bloków.
int *array_of_dis – [IN] Tablica przesunięć początków bloków.
MPI_Datatype *array_of_types – [IN] Tablica definicji typów.
MPI_Datatype *newtype – [OUT] Nowy typ.

Dokładne omówienie wszystkich typów pochodnym można znaleźć w książce [47].

6.2. Pakowanie danych

W przypadku przesyłania w ramach pojedynczych komunikatów danych niejednorodnych (bez regularnej struktury, którą można byłoby opisać przy pomocy typów pochodnych) wygodnie jest użyć mechanizmów pakowania danych do bufora. Programista ma do dyspozycji dwie podstawowe funkcje `MPI_Pack` oraz `MPI_Unpack`.

```
int MPI Pack( void* inbuf, int incount,
              MPI Datatype datatype,
              void *outbuf, int outsize,
              int *position,
              MPI Comm comm)
```

void* inbuf – [IN] Początek danych do zapakowania.

int incount – [IN] Liczba danych do zapakowania.
MPI Datatype datatype – [IN] Typ danych umieszczanych w buforze.
void *outbuf – [OUT] Bufor, w którym umieszczane są dane.
int outsize – [IN] Rozmiar bufora (w bajtach).
int *position – [IN/OUT] Bieżąca pozycja w buforze (aktualizowana po mieszczeniu danych).
MPI Comm comm – [IN] Komunikator.

```
int MPI_Unpack( void* inbuf, int insize, int *position,
                void *outbuf, int outcount,
                MPI_Datatype datatype,
                MPI_Comm_comm )
```

void* inbuf – [IN] Bufor z danymi do rozpakowania.
int insize – [IN] Rozmiar bufora (w bajtach).
int *position – [IN/OUT] Bieżąca pozycja w buforze (aktualizowana po-braniu danych).
void *outbuf – [OUT] Miejsce do rozpakowania danych.
int outcount – [IN] Liczba danych do rozpakowania.
MPI Datatype datatype – [IN] Typ rozpakowywanych danych.
MPI Comm comm – [IN] Komunikator.

Listing 6.5 ilustruje podstawowy przypadek użycia bufora. Proces 0 umieszcza w buforze (tablica `buf`) jedną daną typu `MPI_INT` oraz trzy dane typu `MPI_DOUBLE` wykonując dwukrotnie funkcję `MPI_Pack`. Po każdym wywołaniu aktualizowana jest wartość zmiennej `pos`, która wskazuje na pierwszą wolną składową bufora. Następnie proces wysyła jedną daną typu `MPI_PACKED`. Proces 1 odbiera wiadomość i dwukrotnie wywołuje `MPI_Unpack`.

Listing 6.5. Przesylanie danych niejednorodnych

```
1   int myid, numprocs, n, pos;
2   MPI_Status stat;
3   double a[MAXN];
4   char buf[BSIZE];
5   ...
6   if(myid==0){
7
8       n=3;
9       a[0]=10.0; a[1]=20.0; a[2]=30.0;
10
11      pos=0;
12      MPI_Pack(&n,1,MPI_INT,buf,BSIZE,&pos,
13                                          MPI_COMM_WORLD);
14      MPI_Pack(a,n,MPI_DOUBLE,buf,BSIZE,&pos,
15                                          MPI_COMM_WORLD);
16
```

```
17          MPI_Send(buf,pos,MPI_PACKED,1,100,MPI_COMM_WORLD);
18
19      }else{
20
21          MPI_Recv(buf,BSIZE,MPI_PACKED,MPI_ANY_SOURCE,100,
22                                                  MPI_COMM_WORLD,&stat);
23          pos=0;
24          MPI_Unpack(buf,BSIZE,&pos,&n,1,MPI_INT,
25                                                  MPI_COMM_WORLD);
26          MPI_Unpack(buf,BSIZE,&pos,a,n,MPI_DOUBLE,
27                                                  MPI_COMM_WORLD);
28          ...
29      }
```

Listing 6.6 pokazuje wykorzystanie mechanizmu umieszczania danych w buforze dla realizacji przesłania dolnego trójkąta macierzy. Kolejny przykład (listing 6.7) pokazuje, że umieszczona w buforze sekwencja danych tego samego typu może być odebrana przy użyciu funkcji `MPI_Recv`.

Listing 6.6. Przesłanie dolnego trójkąta macierzy

```
1   int myid, numprocs, i, j, n, pos;
2   MPI_Status stat;
3   double a[MAXN*MAXN];
4   char buf[BSIZE];   // BSIZE >= MAXN*(MAXN+1)/2
5   ...
6   n=3;
7   if(myid==0){
8       for(j=0;j<n;j++)
9           for(i=0;i<n;i++)
10              a[j*MAXN+i]=10.0*(i+1)+j+1;
11      pos=0;
12      for(i=0;i<n;i++){ // pakowanie do bufora kolejnych
13                        // kolumn
14          MPI_Pack(&a[i*MAXN+i],n-i,MPI_DOUBLE,buf,BSIZE,
15                                          &pos,MPI_COMM_WORLD);
16      }
17      MPI_Send(buf,pos,MPI_PACKED,1,100,MPI_COMM_WORLD);
18  }else{
19
20      pos=0;
21      MPI_Recv(buf,BSIZE,MPI_PACKED,MPI_ANY_SOURCE,100,
22                                          MPI_COMM_WORLD,&stat);
23      for(i=0;i<n;i++){ // rozpakowanie z bufora
24                        // kolejnych kolumn
25          MPI_Unpack(buf,BSIZE,&pos,&a[i*MAXN+i],n-i,
26                                  MPI_DOUBLE,MPI_COMM_WORLD);
27      }
28  }
```

Listing 6.7. Pakowanie do bufora i odbiór bez użycia `MPI_PACKED`

```
1   int myid, numprocs, n, pos;
2   MPI_Status stat;
3   double a[MAXN];
4   char buf[BSIZE];
5   ...
6   n=3;
7
8   if(myid==0){
9
10      a[0]=10.0; a[1]=20.0; a[2]=30.0;
11      pos=0;
12      MPI_Pack(a,n,MPI_DOUBLE,buf,BSIZE,&pos,
13                                          MPI_COMM_WORLD);
14      MPI_Send(buf,pos,MPI_PACKED,1,100,MPI_COMM_WORLD);
15
16  }else{
17
18      MPI_Recv(a,3,MPI_DOUBLE,MPI_ANY_SOURCE,100,
19                                      MPI_COMM_WORLD,&stat);
20  }
```

6.3. Wirtualne topologie

W celu łatwego zrealizowania schematu komunikacji pomiędzy procesami korzystnie jest tworzyć wirtualne topologie procesów. Przedstawimy teraz mechanizmy tworzenia topologii kartezjańskich.[2] Podstawową funkcją jest `MPI_Cart_create`. Wywołanie tworzy nowy komunikator, w którym procesy są logicznie zorganizowane w kartezjańską siatkę o liczbie wymiarów `ndims`. Listing 6.8 pokazuje przykładowe zastosowanie tej funkcji do zorganizowania wszystkich procesów (komunikator `MPI_COMM_WORLD`) w dwuwymiarową siatkę procesów. Przy pomocy wywołań funkcji `MPI_Cart_rank` oraz `MPI_Cart_coords` można uzyskać informację o numerze procesu o zadanych współrzędnych oraz współrzędnych procesu o zadanym numerze.

```
int MPI_Cart_create( MPI_Comm comm_old, int ndims,
                     int *dims, int *periods,
                     int reorder, MPI_Comm *comm_cart )
```

MPI_Comm comm_old – [IN] Bazowy komunikator.
int ndims – [IN] Liczba wymiarów tworzonej siatki procesów.
int *dims – [IN] Tablica specyfikująca liczbę procesów w każdym wymiarze.

[2] Możliwe jest również utworzenie topologii zdefiniowanej poprzez pewien graf [47].

int *periods – [IN] Tablica wartości logicznych specyfikująca okresowość poszczególnych wymiarów (następnikiem ostatniego procesu jest pierwszy).

int reorder – [IN] Wartość logiczna informująca o dopuszczalności zmiany numerów procesów w nowym komunikatorze (wartość 0 oznacza, że numer każdego procesu w ramach nowego komunikatora jest taki sam jak numer w komunikatorze bazowym).

MPI_Comm *comm_cart – [OUT] Nowy komunikator.

Listing 6.8. Tworzenie komunikatora kartezjańskiego (liczba procesów równa kwadratowi pewnej liczby)

```
1  int main( int argc, char *argv[])
2  {
3      int myid, numprocs;
4      MPI_Status stat;
5      MPI_Comm cartcom;
6
7      int dims[2];
8      int pers[2];
9      int coord[2];
10
11     MPI_Init(&argc,&argv);
12
13     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
14     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
15
16     dims[0]=sqrt(numprocs);
17     dims[1]=sqrt(numprocs);
18     pers[0]=1;
19     pers[1]=1;
20
21     MPI_Cart_create(MPI_COMM_WORLD,2,dims,pers,0,&cartcom);
22     MPI_Cart_coords(cartcom,myid,2,coord);
23
24     printf("Rank= %d (%d,%d)\n",myid,coord[0],coord[1]);
25     MPI_Finalize();
26     return 0;
27 }
```

```
int MPI_Cart_rank( MPI_Comm comm, int *coords, int *rank )
```

MPI_Comm comm – [IN] Komunikator kartezjański.

int *coords – [IN] Tablica opisująca współrzędne pewnego procesu.

int *rank – [OUT] Numer procesu o podanych współrzędnych.

```
int MPI_Cart_coords( MPI_Comm comm, int rank,
                     int maxdims, int *coords )
```

MPI_Comm comm – [IN] Komunikator kartezjański.
int rank – [IN] Numer pewnego procesu w komunikatorze `comm`.
int maxdims – [IN] Liczba składowych tablicy `coords`.
int *coords – [OUT] Tablica opisująca współrzędne wskazanego procesu.

Bardzo przydatną funkcją jest `MPI_Dims_create`. Ułatwia ona przygotowanie parametrów wywołania funkcji `MPI_Cart_create` dla stworzenia siatki kartezjańskiej dla zadanej liczby procesów oraz liczby wymiarów. Jest użycie pokazano ma listingu 6.9. Listing 6.10 pokazuje przykład wykorzystania wspomnianych funkcji dla stworzenia dwuwymiarowej siatki procesów. Dodatkowo program wyświetla współrzędne sąsiadów każdego procesu na północ, wschód, południe i zachód. Przy tworzeniu siatki (funkcja `MPI_Cart_create`) wskazano, że każdy wymiar ma być „okresowy" (składowe tablicy `pers` równe 1), zatem przykładowo „północny" sąsiad procesu w pierwszym wierszu, to proces w tej samej kolumnie i dolnym wierszu.

```
int MPI_Dims_create(int nprocs, int ndims, int *dims)
```

int nprocs – [IN] Liczba procesów w siatce.
int ndims – [IN] Liczba wymiarów siatki procesów.
int *dims – [IN/OUT] Tablica liczby procesów w każdym wymiarze (gdy na wejściu wartość składowej jest równa 0, wówczas funkcja wyznaczy liczbę procesów dla tego wymiaru.

Listing 6.9. Użycie `MPI_Cart_create` dla stworzenia opisu siatki

```
1  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
2  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
3
4  ndims=2;
5  dims[0]=0; dims[1]=0;
6  pers[0]=1; pers[1]=1;
7
8  MPI_Dims_create(numprocs,ndims,dims);
9  MPI_Cart_create(MPI_COMM_WORLD,2,dims,pers,0,&cartcom);
10 MPI_Cart_coords(cartcom,myid,2,coord);
```

Listing 6.10. Wyznaczanie sąsiadów w topologii kartezjańskiej

```
1  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
2  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
3
4  ndims=2;
5  dims[0]=0; dims[1]=0;
6  pers[0]=1; pers[1]=1;
7
8  MPI_Dims_create(numprocs,ndims,dims);
```

```
9       MPI_Cart_create(MPI_COMM_WORLD,2,dims,pers,1,&cartcom);
10      MPI_Comm_rank(cartcom,&myid);
11      MPI_Cart_coords(cartcom,myid,2,coord);
12
13      int c0=coord[0];
14      int c1=coord[1];
15
16      int ns,es,ss,ws;
17      coord[0]=c0;
18      coord[1]=c1-1;
19      MPI_Cart_rank(cartcom,coord,&ws);
20
21      coord[0]=c0;
22      coord[1]=c1+1;
23      MPI_Cart_rank(cartcom,coord,&es);
24
25      coord[0]=c0-1;
26      coord[1]=c1;
27      MPI_Cart_rank(cartcom,coord,&ns);
28
29      coord[0]=c0+1;
30      coord[1]=c1;
31      MPI_Cart_rank(cartcom,coord,&ss);
32
33      printf("Rank= %d (%d,%d) %d %d %d %d\n",
34                                  myid,c0,c1,ns,es,ss,ws);
```

Dla realizacji komunikacji wzdłuż procesów w tym samym wymiarze można wykorzystać funkchę `MPI_Cart_shift`. Oblicza ona numery procesów wzdłuż wymiaru określonego parametrem `direction`. Przesunięcie względem wywołującego procesu określa parametr `disp`. Numery procesów wysyłających i odbierających są przekazywane w parametrach `source` i `dest`. Listing 6.11 ilustruje użycie funkcji `MPI_Cart_shift` w połączeniu z funkcją `MPI_Sendrecv_replace`. Każdy proces wysyła swój numer do sąsiad poniżej i odbiera numer od sąsiada powyżej.

```
int MPI_Cart_shift( MPI_Comm comm, int direction,
                    int disp, int *source, int *dest )
```

MPI_Comm comm – [IN] Komunikator kartezjański.
int direction – [IN] Numer współrzędnej wymiaru.
int disp – [IN] Przemieszczenie (> 0: w stronę rosnących numerów, < 0: w stronę numerów malejących).
int *source – [OUT] Numer procesu wysyłającego.
int *dest – [OUT] Numer procesu odbierającego.

Listing 6.11. Wyznaczanie przesunięć wzdłuż poszczególnych wymiarów siatki wraz z komunikacją

```
1   ndims=2;
2   dims[0]=0; dims[1]=0;
3   pers[0]=1; pers[1]=1;
4
5   MPI_Dims_create(numprocs,ndims,dims);
6   MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,pers,0,
7                                                  &cartcom);
8   MPI_Cart_coords(cartcom,myid,2,coord);
9
10  int c0=coord[0];
11  int c1=coord[1];
12  MPI_Cart_shift(cartcom,0,1,&source,&dest);
13
14  double xx=(double)myid;
15
16  printf("Rank=␣%d␣(%d,%d),␣przed:␣%lf␣\n",myid,c0,c1,xx);
17  MPI_Barrier(cartcom);
18  MPI_Cart_shift(cartcom,0,1,&source,&dest);
19  MPI_Sendrecv_replace(&xx,1,MPI_DOUBLE,dest,102,source,
20                                          102,cartcom,&stat);
21  printf("Rank=␣%d␣(%d,%d),␣po:␣%lf␣\n",myid,c0,c1,xx);
```

W ramach komunikatorów kartezjańskich wygodnie jest tworzyć komunikatory zawężające grupę procesów komunikujących się ze sobą do wybranego „podwymiaru" siatki procesów, co zilustrujemy w dalszym ciągu. Realizuje to funkcja `MPI_Cart_sub`, w której dla danego komunikatora określamy, czy ma pozostać w nowym komunikatorze. Użycie ilustruje listing 6.12.

```
int MPI_Cart_sub( MPI_Comm comm, int *remain_dims,
                  MPI_Comm *newcomm )
```

MPI_Comm comm – [IN] Komunikator kartezjański.
int *remain_dims – [IN] Tablica określająca, czy dany wymiar ma pozostać (wartość 1), czy też ma być pominięty w danym komunikatorze.
MPI_Comm *newcomm – [OUT] Nowy komunikator.

Listing 6.12. Tworzenie komunikatorów – podgrup komunikatora kartezjańskiego

```
1   dims[0]=0; dims[1]=0;
2   pers[0]=1; pers[1]=1;
3   rem[0]=1; rem[1]=0;
4
5   MPI_Dims_create(numprocs,ndims,dims);
6   MPI_Cart_create(MPI_COMM_WORLD,2,dims,pers,1,&cartcom);
7   MPI_Cart_coords(cartcom,myid,2,coord);
```

```
8
9       MPI_Cart_sub(cartcom,rem,&splitcom);
10      MPI_Comm_rank(splitcom,&newid);
11
12      printf("Rank= %d (%d,%d) - %d\n",myid,coord[0],
13                                              coord[1],newid);
```

6.4. Przykłady

Rozważmy następujący problem [25]. Należy zaprogramować operację $\mathbf{y} \leftarrow A\mathbf{x}$, $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$, na siatce $P = \sqrt{p} \times \sqrt{p}$ procesów, gdzie dla całkowitej wartości $q = n/p$, blok $A_{ij} \in \mathbb{R}^{q \times q}$ macierzy

$$A = \begin{pmatrix} A_{00} & \dots & A_{0,p-1} \\ \vdots & & \vdots \\ A_{p-1,0} & \dots & A_{p-1,p-1} \end{pmatrix}$$

jest przechowywany przez proces P_{ij}. Proces P_{i0}, $i = 0, \dots, p-1$, przechowuje $\mathbf{x}_i \in \mathbb{R}^q$ oraz $\mathbf{y}_i \in \mathbb{R}^q$. Rozważmy następujący algorytm.

1. Utworzyć komunikator kartezjański dla siatki $p \times p$ procesów oraz komunikatory podgrup dla wierszy i kolumn procesów w siatce.
2. Każdy proces P_{i0} wysyła $\mathbf{x}_i$ do procesu P_{ii}.
3. Każdy proces P_{ii} wysyła $\mathbf{x}_i$ do pozostałych procesów w kolumnie.
4. Każdy proces P_{ij} wyznacza $\mathbf{t}_{ij} \leftarrow A_{ij}\mathbf{x}_j$. Procesy tworzące wiersze wykonują operację $\mathbf{y}_i \leftarrow \sum_{j=0}^{p-1} \mathbf{t}_{ij}$, której wynik jest składowany przez proces P_{i0}, $i = 0, \dots, p-1$.
5. Zwolnić utworzone komunikatory.

Listing 6.13 kompletny program z przykładową implementację powyższego algorytmu.

Listing 6.13. Mnożenie macierzy przez wektor na kwadratowej siatce procesów

```
1  /*
2     Operacja y <- y + Ax na kwadratowej siatce
3     Proces P(i,j) przechowuje blok A(i,j) macierzy n x n
4     Proces P(i,0) przechowuje blok x(i) oraz y(i)
5  */
6  #include "mpi.h"
7  #include <stdio.h>
8  #include <math.h>
9
10 int main( int argc, char *argv[])
11 {
```

```
12      int myid, myid2d, numprocs, i, j, ndims, newid1, newid2;
13      double tim;
14
15      MPI_Status stat;
16      MPI_Comm cartcom, splitcom1, splitcom2 ;
17
18      int dims[2];
19      int pers[2];
20      int coord[2];
21      int rem[2];
22
23      int p,q,n;
24      double *a;
25      double *x;
26      double *t;
27      double *y;
28      double *w;
29
30      ndims=2;
31
32      MPI_Init(&argc,&argv);
33      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
34      MPI_Comm_rank(MPI_COMM_WORLD,&myid);
35
36      if(myid==0){
37        scanf("%d",&n);
38      }
39
40      MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
41
42
43      dims[0]=0; dims[1]=0;
44
45      pers[0]=1; pers[1]=1;
46
47  // krok 1: tworzenie siatki kartezjańskiej
48
49      MPI_Dims_create(numprocs,ndims,dims);
50      MPI_Cart_create(MPI_COMM_WORLD,2,dims,pers,1,&cartcom);
51      MPI_Comm_rank(cartcom,&myid2d);
52      MPI_Cart_coords(cartcom,myid2d,2,coord);
53
54      p=dims[0]; // dims[0]==dims[1]==sqrt(numprocs)
55      int myrow=coord[0];
56      int mycol=coord[1];
57
58  //  komunikatory dla kolumn i wierszy
59
60      rem[0]=1; rem[1]=0;
61      MPI_Cart_sub(cartcom,rem,&splitcom1);
62      MPI_Comm_rank(splitcom1,&newid1);
```

```
63
64      rem[0]=0; rem[1]=1;
65      MPI_Cart_sub(cartcom,rem,&splitcom2);
66      MPI_Comm_rank(splitcom2,&newid2);
67
68  // alokacja tablic i lokalne generowanie danych
69
70      q=n/p;
71
72      a=malloc(q*q*sizeof *a);
73      x=malloc(q*sizeof *x);
74      t=malloc(q*sizeof *t);
75      y=malloc(q*sizeof *y);
76      w=malloc(q*sizeof *w);
77
78      if(mycol==0){
79        for(i=0;i<q;i++){
80          x[i]=(myrow+1)*10+mycol+1;
81          y[i]=(myrow+1)*10+mycol+1;
82        }
83      }
84
85      for(i=0;i<q;i++)
86        for(j=0;j<q;j++)
87          a[i*q+j]=(myrow+1)*10+mycol+1;
88
89      if(myid2d==0)
90         tim=MPI_Wtime();
91
92  // krok 2: x(i) -> P(i,i)
93
94      if((mycol==0)&&(myrow!=0)){
95          MPI_Send(x,q,MPI_DOUBLE,myrow,200,splitcom2);
96      }
97
98      if((myrow==mycol)&&(myrow!=0)){
99          MPI_Recv(x,q,MPI_DOUBLE,0,200,splitcom2,&stat);
100     }
101
102 // krok 3: P(i,i) -> x(i) do P(*,i)
103
104     MPI_Bcast(x,q,MPI_DOUBLE,mycol,splitcom1);
105
106 // krok 4: obliczenia lokalne t<-A(i,j)x(j)
107
108     for(i=0;i<q;i++){
109       t[i]=0;
110       for(j=0;j<q;j++)
111         t[i]+=a[i*q+j]*x[j];
112     }
113
```

```
114 // krok 5: sumowanie t przez P(i,*) - wynik w P(i,0)
115
116     MPI_Reduce(t,w,q,MPI_DOUBLE,MPI_SUM,0,splitcom2);
117
118     if(mycol==0){
119
120         for(i=0;i<q;i++)
121           y[i]+=w[i];
122     }
123
124     if(myid2d==0){
125         tim=MPI_Wtime()-tim;
126         printf("Czas  =  %lf\n",tim);
127         printf("Mflops=  %lf\n",(2*(double)n/1000000.0)*n/tim);
128
129     }
130
131 // krok 6: zwolnienie utworzonych komunikatorów
132
133      MPI_Comm_free(&splitcom1);
134      MPI_Comm_free(&splitcom2);
135      MPI_Comm_free(&cartcom);
136      MPI_Finalize();
137      return 0;
138 }
```

Rozważmy teraz algorytm Cannona rozproszonego mnożenia macierzy [25]. Niech $A, B, C \in \mathbb{R}^{n\times n}$. Należy wyznaczyć macierz

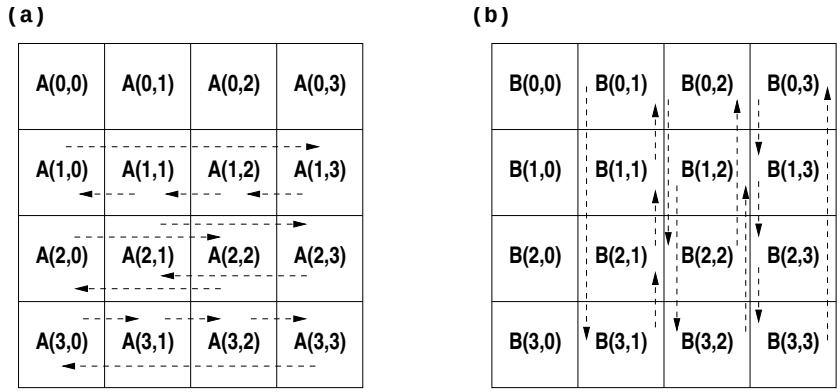
$$C \leftarrow C + AB$$

na dwuwymiarowej siatce procesów $p \times p$ przy założeniu, że p dzieli n oraz macierze A, B, C są alokowane w pamięciach lokalnych procesów P_{ij}, $i, j = 0, \ldots, p-1$ tak, że dla blokowego rozkładu macierzy postaci

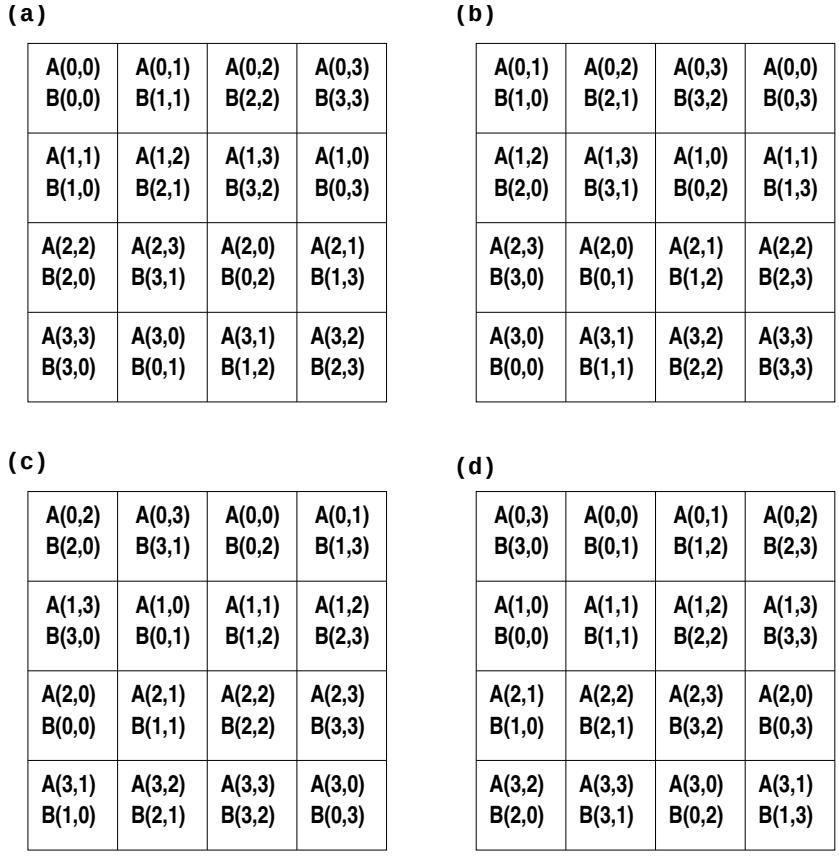
$$A = \begin{pmatrix} A_{00} & \ldots & A_{0,p-1} \\ \vdots & & \vdots \\ A_{p-1,0} & \ldots & A_{p-1,p-1} \end{pmatrix}$$

bloki $A_{ij}, B_{ij}, B_{ij} \in \mathbb{R}^{r\times r}$, $r = n/p$, znajdują się w pamięci procesu P_{ij}.

Rysunek 6.1 ilustruje początkowe przesunięcie poszczególnych bloków macierzy A i B. Na rysunku 6.2 pokazono poszczególne etapy (jest ich dokładnie p) dla wyznaczenia poszczególnych bloków C_{ij} przy użyciu przesyłanych bloków macierzy A i B. Po wykonaniu każdego etapu, wiersze bloków macierzy A są przesuwane cyklicznie w lewo, zaś kolumny bloków macierzy B cyklicznie do góry. Na listingu 6.14 pokazano implementację tego algorytmu.



Rysunek 6.1. Algorytm Cannona: początkowe przesunięcie bloków macierzy A i B



Rysunek 6.2. Etapy algorytmu Cannona rozproszonego mnożenia macierzy: (a) po początkowym przemieszczeniu, (b)–(d) po kolejnych etapach

Listing 6.14. Algorytm mnożenia macierzy na kwadratowej siatce procesów

```
/*
  Operacja C <- C + AB na kwadratowej siatce p x p
  Proces P(i,j) przechowuje bloki A(i,j)
                oraz B(i,j) macierzy
  Proces P(i,j) przechowuje blok C(i,j) wyniku
*/

#include "mpi.h"
#include "mkl.h"
#include <stdio.h>
#include <math.h>

int main( int argc, char *argv[])
{
    int myid,myid2d,numprocs,i,j,ndims,newid1,newid2;
    double tim;

    MPI_Status stat;
    MPI_Comm cartcom, splitcom1, splitcom2 ;

    int dims[2];
    int pers[2];
    int coord[2];
    int rem[2];

    int p,q,n;
    double *a;
    double *b;
    double *c;

    int up,down,left,right,shiftsource,shiftdest;
    char TRANSA='N', TRANSB='N';
    double ALPHA =1.0, BETA=1.0;

    ndims=2;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if(myid==0){
      scanf("%d",&n);
    }

    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

    dims[0]=0; dims[1]=0;
    pers[0]=1; pers[1]=1;


```

```
51 // krok 1: tworzenie siatki
52
53      MPI_Dims_create(numprocs,ndims,dims);
54      MPI_Cart_create(MPI_COMM_WORLD,2,dims,pers,1,
55                                                  &cartcom);
56      MPI_Comm_rank(cartcom,&myid2d);
57      MPI_Cart_coords(cartcom,myid2d,2,coord);
58
59      p=dims[0]; //   dims[0]==dims[1]==sqrt(numprocs)
60      int myrow=coord[0];
61      int mycol=coord[1];
62
63      MPI_Cart_shift(cartcom,1,-1,&right,&left);
64      MPI_Cart_shift(cartcom,0,-1,&down,&up);
65
66 // alokacja tablic i generowanie danych
67
68    q=n/p;
69
70    a=malloc(q*q*sizeof *a);
71    b=malloc(q*q*sizeof *b);
72    c=malloc(q*q*sizeof *c);
73
74    for(i=0;i<q;i++)
75      for(j=0;j<q;j++){
76        a[i*q+j]= myid;//(myrow+1)*10+mycol+1;
77        b[i*q+j]= myid; (myrow+1)*100+mycol+1;
78        c[i*q+j]=0;
79      }
80
81    if(myid2d==0)
82       tim=MPI_Wtime();
83
84 // krok 2: przemieszczenie  A(i,j), B(i,j)
85
86   MPI_Cart_shift(cartcom,1,-coord[0],&shiftsource,
87                                              &shiftdest);
88   MPI_Sendrecv_replace(a,q*q,MPI_DOUBLE,shiftdest,
89                  101,shiftsource,101,cartcom,&stat);
90
91   MPI_Cart_shift(cartcom,0,-coord[1],&shiftsource,
92                                              &shiftdest);
93   MPI_Sendrecv_replace(b,q*q,MPI_DOUBLE,shiftdest,
94                        1,shiftsource,1,cartcom,&stat);
95
96   // krok 3: iloczyn lokalnych bloków
97
98   for(i=0;i<dims[0];i++){
99
100    DGEMM(&TRANSA,&TRANSB,&q,&q,&q,&ALPHA,a,&q,b,
101                                          &q,&BETA,c,&q);
```

```
102       MPI_Sendrecv_replace(a,q*q,MPI_DOUBLE,left,1,
103                                  right,1,cartcom,&stat);
104       MPI_Sendrecv_replace(b,q*q,MPI_DOUBLE,up,1,
105                                  down,1,cartcom,&stat);
106     }
107
108   // krok 4: przywrócenie początkowego rozmieszczenia
109   //          A(i,j), B(i,j)
110
111     MPI_Cart_shift(cartcom,1,+coord[0],&shiftsource,
112                                              &shiftdest);
113     MPI_Sendrecv_replace(a,q*q,MPI_DOUBLE,shiftdest,
114                          1,shiftsource,1,cartcom,&stat);
115
116     MPI_Cart_shift(cartcom,0,+coord[1],&shiftsource,
117                                              &shiftdest);
118     MPI_Sendrecv_replace(b,q*q,MPI_DOUBLE,shiftdest,
119                          1,shiftsource,1,cartcom,&stat);
120
121   // informacja diagnostyczna
122
123       if(myid2d==0){
124           tim=MPI_Wtime()-tim;
125           printf("Czas   = %lf\n",tim);
126           printf("Mflops= %lf\n",(2.0*n)*n*(n/1.0e+6)/tim);
127       }
128       MPI_Comm_free(&cartcom);
129       MPI_Finalize();
130       return 0;
131   }
```

6.5. Komunikacja nieblokująca

Komunikacja nieblokująca może być wykorzystana do szybszego wykonania algorytmów rozproszonych dzięki nakładaniu na siebie (jednoczesnemu wykonaniu) obliczeń oraz przesyłania danych. Funkcje `MPI_Isend` `MPI_Irecv` inicjują operacje wysyłania i odbioru zwracając `request` reprezentującą zainicjowaną operację. Funkcja `MPI_Wait` oczekuje na zakończenie operacji, zaś funkcja `MPI_Test` testuje stan operacji.

```
int MPI_Isend( void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm,
               MPI_Request *request )
```

MPI_Request *request – [OUT] Reprezentuje zainicjowaną operację wysyłania (pozostałe parametry jak w MPI_Send).

```
int MPI_Irecv( void* buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Request *request )
```

MPI_Request *request – [OUT] Reprezentuje zainicjowaną operację wysyłania (pozostałe parametry jak w MPI_Recv).

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
```

MPI_Request *request – [IN] Reprezentuje zainicjowaną operację.
MPI_Status *status – [OUT] Status zakończenia.

```
int MPI_Test( MPI_Request *request, int *flag,
              MPI_Status *status )
```

MPI_Request *request – [IN] Reprezentuje zainicjowaną operację.
int *flag – [OUT] Informacja o zakończeniu operacji (wartość !=0 oznacza zakończenie operacji).
MPI_Status *status – [OUT] Status zakończenia.

Program przedstawiony na listingu 6.15 ilustruje wymianę wartości przy wykorzystaniu trybu standardowego komunikacji, co może prowadzić do zakleszczenia (ang. *deadlock*). Program 6.16 pokazuje rozwiązanie tego problemu przy pomocy komunikacji nieblokującej. Inne możliwe rozwiązanie może wykorzystać funkcję `MPI_Sendrecv_replace`.

Listing 6.15. Wymiana wartości przez dwa procesy – możliwy DEADLOCK

```
1  int myid, numprocs, i;
2  MPI_Status stat;
3  double a,b;
4
5  MPI_Init(&argc,&argv);
6  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
7  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
8
9  a=(double)myid;
10
11 MPI_Send(&a,1,MPI_DOUBLE,(myid+1)%numprocs,100,
12                                        MPI_COMM_WORLD);
13 MPI_Recv(&a,1,MPI_DOUBLE,(myid-1+numprocs)%numprocs,
14                                  100,MPI_COMM_WORLD,&stat);
15 printf("id=%d a=%lf\n",myid,a);
```

Listing 6.16. Wymiana wartośsci przez dwa procesy – komunikacja nieblokująca

```
int myid, numprocs, i;
MPI_Status stat;
double a,b;

MPI_Request s_req, r_req;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

a=(double)myid;

MPI_Isend(&a,1,MPI_DOUBLE,(myid+1)%numprocs,
                    100,MPI_COMM_WORLD,&s_req);
MPI_Irecv(&a,1,MPI_DOUBLE,(myid-1+numprocs)%numprocs,
                    100,MPI_COMM_WORLD,&r_req);

MPI_Wait(&r_req,&stat); // oczekiwanie na zakończenie

printf("id=%d a=%lf\n",myid,a);
```

Listing 6.17 zawiera algorytm mnożenia macierzy na kwadratowej siatce procesów z komunikacją nieblokującą.

Listing 6.17. Algorytm mnożenia macierzy na kwadratowej siatce procesów z komunikacją nieblokującą

```
/*
  Operacja C <- C + AB na kwadratowej siatce p x p
  Proces P(i,j) przechowuje bloki A(i,j) oraz B(i,j) macierzy
  Proces P(i,j) przechowuje blok C(i,j) wyniku
  Komunikacja nieblokująca
*/
#include "mpi.h"
#include "mkl.h"
#include <stdio.h>
#include <math.h>

int main( int argc, char *argv[])
{
    int myid, myid2d, numprocs, i, j, ndims, newid1, newid2;
    double tim;

    MPI_Status stat;
    MPI_Comm cartcom, splitcom1, splitcom2 ;
    MPI_Request reqs[4];

    int dims[2];
```

```
22      int pers[2];
23      int coord[2];
24      int rem[2];
25
26      int p,q,n;
27      double *a;
28      double *b;
29      double *c;
30      double *a_buff[2], *b_buff[2];
31
32      int up,down,left,right,shiftsource,shiftdest;
33
34      char TRANSA='N', TRANSB='N';
35      double ALPHA =1.0, BETA=1.0;
36
37      MPI_Init(&argc,&argv);
38      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
39      MPI_Comm_rank(MPI_COMM_WORLD,&myid);
40
41      if(myid==0){
42        scanf("%d",&n);
43      }
44
45      MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
46
47      ndims=2;
48      dims[0]=0; dims[1]=0;
49      pers[0]=1; pers[1]=1;
50
51  // krok 1: tworzenie siatki
52
53      MPI_Dims_create(numprocs,ndims,dims);
54      MPI_Cart_create(MPI_COMM_WORLD,2,dims,pers,
55                                          1,&cartcom);
56      MPI_Comm_rank(cartcom,&myid2d);
57      MPI_Cart_coords(cartcom,myid2d,2,coord);
58
59      p=dims[0];
60      int myrow=coord[0];
61      int mycol=coord[1];
62
63      MPI_Cart_shift(cartcom,1,-1,&right,&left);
64      MPI_Cart_shift(cartcom,0,-1,&down,&up);
65
66
67  // alokacja tablic i generowanie danych
68
69     q=n/p;
70
71     a=malloc(q*q*sizeof *a);
72     b=malloc(q*q*sizeof *b);
```

```
73      c=malloc(q*q*sizeof *c);
74
75      a_buff[0]=a;
76      a_buff[1]=malloc(q*q*sizeof *a);
77
78      b_buff[0]=b;
79      b_buff[1]=malloc(q*q*sizeof *b);
80
81
82      for(i=0;i<q;i++)
83        for(j=0;j<q;j++){
84          a[i*q+j]= myid;//(myrow+1)*10+mycol+1;
85          b[i*q+j]= myid; (myrow+1)*100+mycol+1;
86          c[i*q+j]=0;
87        }
88
89      if(myid2d==0)
90         tim=MPI_Wtime();
91
92  // krok 2: przemieszczenie A(i,j), B(i,j)
93
94    MPI_Cart_shift(cartcom,1,-coord[0],&shiftsource,
95                                          &shiftdest);
96    MPI_Sendrecv_replace(a_buff[0],q*q,MPI_DOUBLE,
97       shiftdest,101,shiftsource,101,cartcom,&stat);
98
99    MPI_Cart_shift(cartcom,0,-coord[1],&shiftsource,
100                                         &shiftdest);
101   MPI_Sendrecv_replace(b_buff[0],q*q,MPI_DOUBLE,
102         shiftdest,1,shiftsource,1,cartcom,&stat);
103
104 // krok 3: iloczyny lokalne
105
106   for(i=0;i<dims[0];i++){
107     MPI_Isend(a_buff[i%2],q*q,MPI_DOUBLE,left,1,
108                                   cartcom,&reqs[0]);
109     MPI_Isend(b_buff[i%2],q*q,MPI_DOUBLE,up,1,
110                                   cartcom,&reqs[1]);
111     MPI_Irecv(a_buff[(i+1)%2],q*q,MPI_DOUBLE,
112                           right,1,cartcom,&reqs[2]);
113     MPI_Irecv(b_buff[(i+1)%2],q*q,MPI_DOUBLE,
114                            down,1,cartcom,&reqs[3]);
115
116     DGEMM(&TRANSA,&TRANSB,&q,&q,&q,&ALPHA,a,&q,
117                                     b,&q,&BETA,c,&q);
118     for(j=0;j<4;j++){
119       MPI_Wait(&reqs[j],&stat);
120     }
121   }
122
123
```

```
124 // krok 4: rozmieszczenie startowe A(i,j), B(i,j)
125
126   MPI_Cart_shift(cartcom,1,+coord[0],&shiftsource,
127                                          &shiftdest);
128   MPI_Sendrecv_replace(a_buff[i%2],q*q,MPI_DOUBLE,
129         shiftdest,1,shiftsource,1,cartcom,&stat);
130
131   MPI_Cart_shift(cartcom,0,+coord[1],&shiftsource,
132                                          &shiftdest);
133   MPI_Sendrecv_replace(b_buff[i%2],q*q,MPI_DOUBLE,
134         shiftdest,1,shiftsource,1,cartcom,&stat);
135
136 // informacja diagnostyczna
137
138    if(myid2d==0){
139       tim=MPI_Wtime()-tim;
140       printf("Czas  = %lf\n",tim);
141       printf("Mflops= %lf\n",(2.0*n)*n*
142                                   (n/1.0e+6.0)/tim);
143    }
144
145   MPI_Comm_free(&cartcom);
146   MPI_Finalize();
147   return 0;
148 }
```

6.6. Zadania

Poniżej zamieściliśmy szereg zadań do samodzielnego wykonania. Do ich rozwiązania należy wykorzystać bibliotekę MPI. Ich celem jest utrwalenie wiadomości zawartych w tym rozdziale.

Zadanie 5.1.
Niech macierz $A \in \mathbb{R}^{n\times n}$ będzie rozmieszczona *blokami wierszy* w pamięciach lokalnych poszczególnych procesów, zaś kopie wektora $\mathbf{x} \in \mathbb{R}^n$ będą się znajdować w pamięci każdego procesu. Zaprogramować operację mnożenia macierzy przez wektor $\mathbf{y} \leftarrow A\mathbf{x}$. Wynik (wektor $\mathbf{y}$) należy umieścić w pamięci lokalnej każdego procesu.

Zadanie 5.2.
Niech macierz dolnotrójkątna $L \in \mathbb{R}^{n\times n}$ będzie rozmieszczona *cyklicznie wierszami* w pamięciach lokalnych poszczególnych procesów, zaś składowe

wektora $\mathbf{b}$ będą rozmieszczone cyklicznie. Zaprogramować operację rozwiązywania układu równań $L\mathbf{x} = \mathbf{b}$.

Zadanie 5.3.
Niech symetryczna i dodatnio określona macierz $A \in \mathbb{R}^{n\times n}$ będzie rozmieszczona *cyklicznie wierszami* w pamięciach lokalnych poszczególnych procesów. Zaprogramować algorytm rozkładu Choleskiego.

Zadanie 5.4.
Niech macierz $A \in \mathbb{R}^{n\times n}$ przechowywana kolumnami będzie rozmieszczona blokami wierszy w pamięciach lokalnych poszczególnych procesów. Zaprogramować operację przesłania do wszystkich procesów wiersza macierzy, który w pierwszej kolumnie ma największą wartość bezwzględną.

Zadanie 5.5.
Niech $A \in \mathbb{R}^{m\times n}$. Wyznaczyć wartość

$$\mu = \max_{0\le i\le m-1} \sum_{j=0}^{n-1} |a_{ij}|$$

na dwuwymiarowej siatce procesów $p \times q$ przy założeniu, że p, q dzielą odpowiednio m, n oraz macierz A jest alokowana w pamięciach lokalnych procesów P_{ij}, $i = 0, \ldots, p-1$, $j = 0, \ldots, q-1$ tak, że dla

$$A = \begin{pmatrix} A_{00} & \ldots & A_{0,q-1} \\ \vdots & & \vdots \\ A_{p-1,0} & \ldots & A_{p-1,q-1} \end{pmatrix}$$

blok $A_{ij} \in \mathbb{R}^{r\times s}$, $r = m/p$, $s = n/q$, znajduje się w pamięci procesu P_{ij}. Użyć następującego algorytmu.

1. Utworzyć komunikator kartezjański 2D oraz komunikatory podgrup dla wierszy i kolumn.
2. Każdy proces P_{ij} wyznacza wektor $\mathbf{x} \in \mathbb{R}^r$ taki, że $x_i = \sum_{j=0}^{s} |a_{ij}|$.
3. Wiersze wykonują operację redukcyjną SUM na wektorach $\mathbf{x}$; wynik umieszczany jest w P_{i0}, $i = 0, \ldots, p-1$ w wektorze $\mathbf{y}$.
4. Procesy P_{i0}, $i = 0, \ldots, p-1$, wyznaczają $\beta = \max_{0\le j\le r} y_j$.
5. Procesy P_{i0} wykonują operację redukcyjną MAX na wartościach β; wynik jest umieszczany w μ na P_{00}.
6. Proces P_{00} rozszyła μ do wszystkich procesów.

Bibliografia

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Kompilatory: reguły, metody i narzędzia*. WNT, Warszawa, 2002.

[2] R. Allen, K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostruchov, D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 1992.

[4] A. Baker, J. Dennis, E. R. Jessup. Toward memory-efficient linear solvers. *Lecture Notes in Computer Science*, 2565:315–238, 2003.

[5] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, S. Tomov. The impact of multicore on math software. *Lecture Notes in Computer Science*, 4699:1–10, 2007.

[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, 2001.

[7] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, R. Whaley. LAPACK working note 100: A proposal for a set of parallel basic linear algebra subprograms. http://www.netlib.org/lapack/lawns, May 1995.

[8] U. Consortium. *UPC Language Specifications*. 2005.

[9] M. J. Daydé, I. S. Duff. The RISC BLAS: a blocked implementation of level 3 BLAS for RISC processors. *ACM Trans. Math. Soft.*, 25:316–340, 1999.

[10] M. J. Daydé, I. S. Duff, A. Petitet. A parallel block implementation of level-3 BLAS for MIMD vector processors. *ACM Trans. Math. Soft.*, 20:178–193, 1994.

[11] J. Dongarra. Performance of various computer using standard linear algebra software. http://www.netlib.org/benchmark/performance.ps.

[12] J. Dongarra, J. Bunsch, C. Moler, G. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, 1979.

[13] J. Dongarra, J. DuCroz, I. Duff, S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[14] J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14:1–17, 1988.

[15] J. Dongarra, I. Duff, D. Sorensen, H. Van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1991.

[16] J. Dongarra, I. Duff, D. Sorensen, H. Van der Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, 1998.

[17] J. Dongarra, F. Gustavson, A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Rev.*, 26:91–112, 1984.

[18] J. Dongarra, S. Hammarling, D. Sorensen. Block reduction of matrices to condensed form for eigenvalue computations. *J. Comp. Appl. Math*, 27:215–227, 1989.

[19] J. Dongarra, i in. *PVM: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, 1994.

[20] E. Elmroth, F. Gustavson, I. Jonsson, B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev.*, 46:3–45, 2004.

[21] J. D. et al., redaktor. *The Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers, 2003.

[22] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C–21:94, 1972.

[23] B. Garbow, J. Boyle, J. Dongarra, C. Moler. *Matrix Eiigensystems Routines – EISPACK Guide Extension*. Lecture Notes in Computer Science. Springer-Verlag, New York, 1977.

[24] G. Golub, J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Academic Press, 1993.

[25] A. Grama, A. Gupta, G. Karypis, V. Kumar. *An Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison Wesley, 2003.

[26] J. Gustafson. Reevaluating Amdahl's law. *Comm. ACM*, 31:532–533, 1988.

[27] J. Gustafson, G. Montry, R. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9:609–638, 1988.

[28] F. G. Gustavson. New generalized data structures for matrices lead to a variety of high performance algorithms. *Lect. Notes Comput. Sci.*, 2328:418–436, 2002.

[29] F. G. Gustavson. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM J. Res. Dev.*, 47:31–56, 2003.

[30] F. G. Gustavson. The relevance of new data structure approaches for dense linear algebra in the new multi-core / many core environments. *Lecture Notes in Computer Science*, 4967:618–621, 2008.

[31] R. Hockney, C. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger Ltd., Bristol, 1981.

[32] Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 2002.

[33] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture*, 2003.

[34] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2007.

[35] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture*, 2008.

[36] L. H. Jamieson, D. B. Gannon, R. J. Douglass. *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[37] B. Kågström, P. Ling, C. V. Loan. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24:268–302, 1998.

[38] J. Kitowski. *Współczesne systemy komputerowe*. CCNS, Kraków, 2000.

[39] M. Kowarschik, C. Weiss. An overview of cache optimization techniques and cache-aware numerical algorithms. *Lecture Notes in Computer Science*, 2625:213–232, 2003.

[40] S. Kozielski, Z. Szczerbiński. *Komputery równoległe: architektura, elementy programowania*. WNT, Warszawa, 1994.

[41] B. C. Kuszmaul, D. S. Henry, G. H. Loh. A comparison of asymptotically scalable superscalar processors. *Theory of Computing Systems*, 35(2):129–150, 2002.

[42] C. Lacoursière. A parallel block iterative method for interactive contacting rigid multibody simulations on multicore pcs. *Lecture Notes in Computer Science*, 4699:956–965, 2007.

[43] M. Larid. A comparison of three current superscalar designs. *Computer Architecture News (CAN)*, 20(3):14–21, 1992.

[44] C. Lawson, R. Hanson, D. Kincaid, F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Soft.*, 5:308–329, 1979.

[45] J.-b. Lee, W. Sung, S.-M. Moon. An enhanced two-level adaptive multiple branch prediction for superscalar processors. *Lengauer, Christian (ed.) et al., Euro-par '97 parallel processing. 3rd international Euro-Par conference, Passau, Germany, August 26-29, 1997. Proceedings. Berlin: Springer. Lect. Notes Comput. Sci. 1300, 1053-1060* . 1997.

[46] C. W. McCurdy, R. Stevens, H. Simon, W. Kramer, D. Bailey, W. Johnston, C. Catlett, R. Lusk, T. Morgan, J. Meza, M. Banda, J. Leighton, J. Hules. Creating science-driven computer architecture: A new path to scientific leadership, Kwi. 16 2007.

[47] MPI Forum. MPI: A Message-Passing Interface Standard. Version 1.3. http://www.mpi-forum.org/docs/mpi21-report.pdf, 2008.

[48] U. Nagashima, S. Hyugaji, S. Sekiguchi, M. Sato, H. Hosoya. An experience with super-linear speedup achieved by parallel computing on a workstation cluster: Parallel calculation of density of states of large scale cyclic polyacenes. *Parallel Computing*, 21:1491–1504, 1995.

[49] Netwok Computer Services Inc. The AHPCRC Cray X1 primer. http://www.ahpcrc.org/publications/Primer.pdf.

[50] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Springer, 1988.

[51] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco, 1996.

[52] M. Paprzycki, C. Cyphers. Gaussian elimination on cray y-mp. *CHPC Newsletter*, 6(6):77–82, 1991.

[53] M. Paprzycki, C. Cyphers. Multiplying matrices on the cray - practical considerations. *CHPC Newsletter*, 6(4):43–47, 1991.

[54] M. Paprzycki, P. Stpiczyński. A brief introduction to parallel computing. E. Kontoghiorghes, redaktor, *Parallel Computing and Statistics*, strony 3–42. Taylor & Francis, 2006. (chapter of the mongraph).

[55] B. Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Plenum Press, 1999.

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

[57] N. Rahman. Algorithms for hardware caches and TLB. *Lecture Notes in Computer Science*, 2625:171–192, 2003.

[58] U. Schendel. *Introduction to numerical methods for parallel computers*. Ellis Horwood Limited, New York, 1984.

[59] J. Stoer, R. Bulirsh. *Introduction to Numerical Analysis*. Springer, New York, wydanie 2nd, 1993.

[60] P. Stpiczyński. Optymalizacja obliczeń rekurencyjnych na komputerach wektorowych i równoległych. *Studia Informatica*, 79, 2008. (rozprawa habilitacyjna, 184 strony).

[61] P. Stpiczyński. Solving a kind of boundary value problem for ODEs using novel data formats for dense matrices. M. Ganzha, M. Paprzycki, T. Pełech-Pilichowski, redaktorzy, *Proceedings of the International Multiconference on Computer Science and Information Technology*, wolumen 3, strony 293–296. IEEE Computer Society Press, 2008.

[62] P. Stpiczyński. A parallel non-square tiled algorithm for solving a kind of BVP for second-order ODEs. *Lecture Notes in Computer Science*, 6067:87–94, 2010.

[63] P. Stpiczyński, M. Paprzycki. Fully vectorized solver for linear recurrence systems with constant coefficients. *Proceedings of VECPAR 2000 – 4th International Meeting on Vector and Parallel Processing, Porto, June 2000*, strony 541–551. Facultade de Engerharia do Universidade do Porto, 2000.

[64] P. Stpiczyński, M. Paprzycki. Numerical software for solving dense linear algebra problems on high performance computers. M. Kovacova, redaktor, *Proceedings of Aplimat 2005, 4th International Conference, Bratislava, Slovakia, February 2005*, strony 207–218. Technical Univerity of Bratislava, 2005.

[65] R. C. Whaley, A. Petitet, J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.

[66] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison–Wesley, 1996.

[67] P. H. Worley. The effect of time constraints on scaled speedup. *SIAM J. Sci. Stat. Comput.*, 11:838–858, 1990.

[68] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.