STUDIA INFORMATICA

Formerly: *Zeszyty Naukowe Politechniki Śląskiej, seria INFORMATYKA*

Quarterly

Volume 29, Number 3B (79)

Przemysław STPICZYŃSKI

OPTYMALIZACJA OBLICZEŃ REKURENCYJNYCH NA KOMPUTERACH WEKTOROWYCH I RÓWNOLEGŁYCH

Silesian University of Technology Press

Gliwice 2008

STUDIA INFORMATICA **Volume 29, Number 3B (79)**

Formerly: *Zeszyty Naukowe Politechniki Śląskiej, seria INFORMATYKA* Nr kol. 1789

STUDIA INFORMATICA is indexed in INSPEC/IEE (London, United Kingdom)

PL ISSN 0208-7286, QUARTERLY
Printed in Poland

Pamięci mojego Ojca

SPIS TREŚCI

CONTENTS

WSTĘP

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].

Jedną z najważniejszych metod tworzenia efektywnych metod obliczeniowych jest koncepcja blokowości obliczeń [35, 33]. Algorytmy powinny być wyrażane w terminach operacji macierzowych i wektorowych, co pozwala na zastosowanie do ich implementacji podstawowych podprogramów algebry liniowej (BLAS). Dzięki temu możliwe jest efektywne wykorzystanie architektury procesorów wyposażonych w mechanizmy wektorowości i potokowości oraz przede wszystkim system hierarchii pamięci, co z kolei stanowi klucz do możliwie pełnego wykorzystania mocy obliczeniowej oferowanej przez procesor. Możliwa jest również dalsza optymalizacja algorytmów blokowych poprzez użycie nowych, wprowadzonych przez F.G. Gustavsona, sposobów reprezentacji macierzy, które znacznie poprawiają wykorzystanie pamięci podręcznej. Dodatkowo, algorytmy blokowe na ogół łatwo poddają się zrównoleglaniu na komputery wieloprocesorowe z pamięcią wspólną oraz mogą być równie łatwo adaptowane dla procesorów wielordzeniowych. Co więcej, po odpowiednim zaprojektowaniu rozmieszczenia

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

danych i obliczeń lokalnych, umożliwiają wykorzystanie mocy komputerów z pamięcią rozproszoną oraz klastrów.

Elementem znacznie ograniczającym możliwość zastosowania koncepcji algorytmów blokowych, a przez to powodującym słabsze wykorzystanie mocy obliczeniowej współczesnych procesorów oraz architektur wieloprocesorowych, są obliczenia rekurencyjne, które zwykle nie są w zadowalającym stopniu optymalizowane przez kompilatory. Celem niniejszej pracy jest przedstawienie koncepcji tworzenia algorytmów blokowych dla obliczeń, mających postać rekurencji liniowej [32, 69], które byłyby w stanie wykorzystywać w dużym stopniu moc pojedynczych procesorów oraz nadawać się do zrównoleglania na różnych typach architektur wieloprocesorowych. Dodatkowo przedstawione zostaną wybrane zastosowania omawianych algorytmów do rozwiązywania wybranych problemów obliczeniowych.

W rozdziale 1 omawiamy podstawowe zagadnienia związane ze współczesnymi architekturami komputerów dużej mocy obliczeniowej. Przedstawiamy wykorzystywane w pracy metody analizy algorytmów, formalnie definiujmy problem liniowych obliczeń rekurencyjnych oraz prezentujemy w skrócie najważniejsze znane metody jego rozwiązania. W rozdziale 2 podajemy efektywną metodę wektoryzacji obliczeń rekurencyjnych. Rozdziały 3 i 4 pokazują metodę, pozwalającą na wykorzystanie operacji macierzowych oraz jej zrównoleglenie na komputery z pamięcią wspólną i rozproszoną. Rozdział 5 omawia zastosowanie wprowadzonych we wcześniejszych rozdziałach metod dla szybkiego wyznaczania liniowych filtrów rekurencyjnych. W rozdziale 6 omawiamy zastosowanie algorytmów z poprzednich rozdziałów do rozwiązania problemu wyznaczania sum trygonometrycznych oraz optymalizacji numerycznego wyznaczania odwrotności transformanty Laplace'a. Rozdział 7 omawia algorytm szybkiego obliczania wartości wielomianów, bazujący na metodzie przedstawionej w rozdziale 2, wraz z jego analizą numeryczną. Na koniec w rozdziale 8 podajemy nową metodę rozmieszczenia danych dla problemu rozwiązywania układów równań liniowych o macierzach trójkątnych, która pozwala na oszczędniejsze wykorzystanie pamięci oraz konstrukcję szybszych algorytmów.

Wszystkie najważniejsze wyniki prezentowane w pracy zostały opublikowane przez autora niniejszej rozprawy. Podkreślmy jednak, że niniejsza rozprawa zawiera ich ujednolicenie oraz często ulepszenie. Część zawartości wstępnego rozdziału 1 opublikowano w rozdziale książki [92] oraz pracy [120]. Metody obliczeniowe opisywane w rozdziałach 2, 3 i 4 opublikowano odpowiednio w pracach [119], [113, 112] i [108, 111]. Praca [114] prezentuje zastosowanie omówionych metod do wyznaczania filtrów rekurencyjnych, zastosowanie zaś nowego sposobu reprezentacji macierzy przedstawiono w pracy [117].

Wykorzystanie metod rozwiązywania równań rekurencyjnych do wyznaczania sum trygonometrycznych opublikowano w pracach [106, 107, 115]. Problem obliczania wartości wielomianów opisano w pracach [109, 110]. Nowa metoda dystrybucji danych została opublikowana w pracy [116].

Wszystkie prezentowane w tej rozprawie nowe algorytmy zostały zaimplementowane przez autora oraz uruchomione na różnych rodzajach sprzętu komputerowego. Poszczególne rozdziały zawierają wyniki oraz wnioski płynące z tych eksperymentów. Komputery Cray C90 oraz Cray SV1 wykorzystane do eksperymentów w rozdziale 2 zostały udostępnione odpowiednio przez *Mississippi Center for Supercomputing Research* (MCSR) w Oxford, Mississippi, oraz *National Partnership for Advanced Computational Infrastructure* (NPACI), Austin, Texas. Komputery Sun UltraSPARC II, Cray SV1 oraz Cray X1 zainstalowane w Interdyscyplinarnym Centrum Modelowania Matematycznego i Komputerowego (ICM) Uniwersytetu Warszawskiego posłużyły do wykonania obliczeń prezentowanych w rozdziałach 3, 4, 5, 6 i 8. Klaster oparty na procesorach Itanium 2 zainstalowany w ośrodku obliczeniowym Uniwersytetu Marii Curie-Skłodowskiej w Lublinie, w ramach projektu Clusterix - Krajowy Klaster Linuksowy, posłużył do wykonania obliczeń prezentowanych w rozdziałach 4, 5, 6 i 8. Pozostały sprzęt informatyczny wykorzystany do obliczeń przedstawianych w niniejszej rozprawie jest zainstalowany w Instytucie Matematyki UMCS w Lublinie.

1. Metody optymalizacji programów dla współczesnych architektur komputerowych

W pierwszym rozdziale przedstawimy krótki przegląd najistotniejszych 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 na te tematy można znaleźć w książkach [40, 44, 66, 68, 93]. Sformułujemy również problem liniowych obliczeń rekurencyjnych oraz przedstawimy w skrócie najważniejsze, opisane w literaturze, metody jego rozwiązania.

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 [68]. 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* [75]).

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 $\alpha\mathbf{x}$ [32]. 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 [32]. 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 [72, 70], 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* [79]) 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. 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 [28]. Warto wspomnieć, że podobną ideę zastosowano przy budowie superkomputera Cray X1, gdzie połączono cztery procesory SSP (ang. *single-*

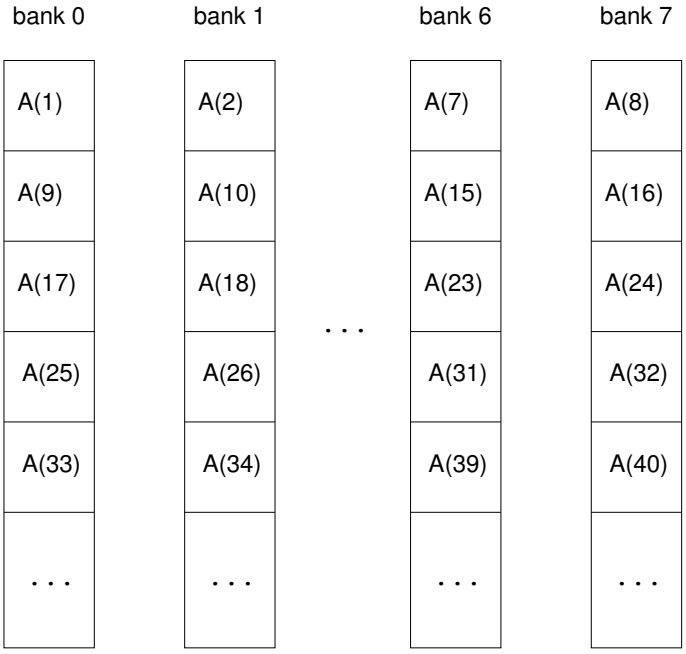
streaming processor) w procesor MSP (ang. *multi-streaming processor*). Więcej informacji na temat architektury tego komputera zostanie podanych w podrozdziale 1.9.

Idea wektorowości została wykorzystana w popularnych procesorach Intela, które począwszy od modelu Pentium III zostały wyposażone w mechanizm SSE (ang. *streaming SIMD extensions* [58, 57]), 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 [41]. 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 [60, 59] 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.

xmm0:	**x3**	**x2**	**x1**	**x0**
	+	**+**	**+**	**+**
xmm1:	**y3**	**y2**	**y1**	**y0**
	=	**=**	**=**	**=**
xmm0:	**x3+y3**	**x2+y2**	**x1+y1**	**x0+y0**

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

Fig. 1.1. Adding two vectors using the instruction `addps xmm0,xmm1`



Rys. 1.2. Rozmieszczenie składowych tablicy w ośmiu bankach pamięci
Fig. 1.2. Allocation of an array in the eight-way interleaved memory

1.2. Wykorzystanie pamięci komputera

Kolejnym 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 napisaną w języku Fortran.

```
do i=1,N,K
  A(i)=A(i)+1
end do
```

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 [34, 87, 88]. 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 [96]. 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 [127]. 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 przesła-

nia określonego obszaru pamięci w stronę procesora, jak to ma miejsce w przypadku rozszerzeń SSE [58]. 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 [57, 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

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

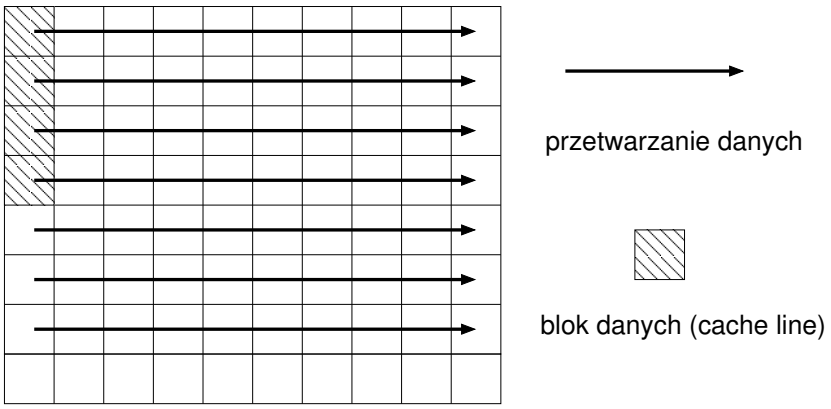
Przy rozmieszczeniu elementów kolumnami istotny jest parametr `LDA` (ang. *leading dimension of array*), określający liczbę wierszy tablicy dwuwymiarowej, w której przechowywana jest macierz (1.2). Zwykle przyjmuje się `LDA` $= m$, choć w pewnych przypadkach z uwagi na możliwe lepsze wykorzystanie pamięci podręcznej, korzystniej jest zwiększyć wiodący rozmiar tablicy (ang. *leading dimension padding*), przyjmując za `LDA` liczbę nieparzystą większą niż m [67], 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. Algorytmy, które zostaną opisane w rozdziałach 2 i 3, wymagają przetwarzania tablic zarów-

no wierszami, jak i kolumnami, a zatem ewentualna zamiana porządku nie przyniesie poprawy wykorzystania pamięci podręcznej.

```
      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
      *  *  *  *  *  *  *  *  *  *
```

Rys. 1.3. Kolumnowe rozmieszczenie składowych tablicy dwuwymiarowej 7×10 dla `LDA=8`
Fig. 1.3. Standard column major storage of a 7×10 array with `LDA=8`



Rys. 1.4. Zjawisko *cache miss* przy rozmieszczeniu kolumnowym
Fig. 1.4. Cache miss in the standard column major storage

1.2.3. Nowe sposoby reprezentacji macierzy

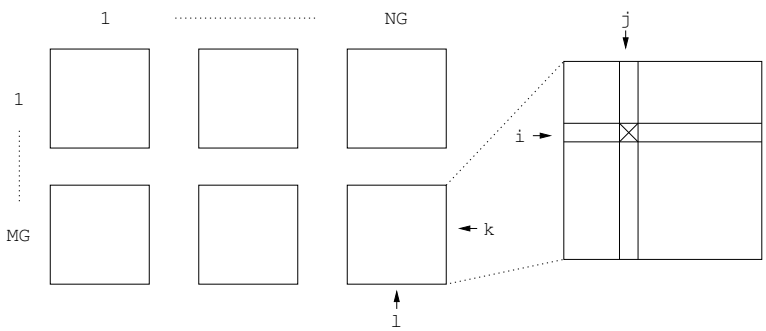
W celu ograniczenia opisanych w poprzednim punkcie niekorzystnych zjawisk, związanych ze stosowaniem tablic dwuwymiarowych, zaproponowano nowe sposoby reprezentacji macierzy [47, 48], które prowadzą do konstrukcji bardzo szybkich algorytmów [39]. Podstawową ideą nowego sposobu jest podział macierzy na bloki według następującego schematu

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

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, 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 [47]. Postuluje się również implementację wsparcia nowych sposobów reprezentacji na poziomie kompilatora, co znacznie ułatwiłoby konstrukcję efektywnych i szybkich algorytmów [39].

```
      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  *  *
      *  *  *  * |  *  *  *  * |  *  *  *  *
```

Rys. 1.5. Nowy blokowy sposób reprezentacji macierzy
Fig. 1.5. New square blocked full data format

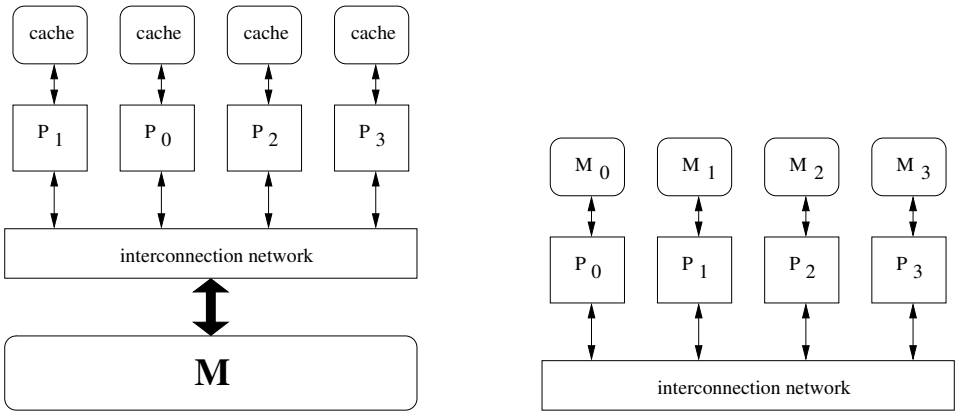


Rys. 1.6. Nowy blokowy sposób rozmieszczenia składowych w czterowymiarowej tablicy
Fig. 1.6. Square blocked data format in a four dimensional array

Najdogodniejszym sposobem implementacji nowego sposobu reprezentacji macierzy są tablice wielowymiarowe. Rysunek 1.6 pokazuje sposób użycia tablic o czterech wymiarach w języku Fortran. Symbol ⊠ wskazuje składową `A(i,j,k,l)` o lokalnych współrzędnych (i, j) w bloku $A_{k,l}$. W podrozdziale 5.4 podamy przykład wykorzystania nowego sposobu reprezentacji macierzy dla obliczania wartości filtra rekurencyjnego.

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 [41], który to model obejmuje większość współczesnych komputerów wieloprocesorowych. Z punktu widzenia programisty najistotniejszy będzie jednak dalszy podział wynikający z typu zastosowanej pamięci (rysunek 1.7). Będziemy zatem zajmować się komputerami wieloprocesorowymi wyposażonymi we wspólną pamięć (ang. *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.



Rys. 1.7. Komputery klasy MIMD z pamięcią wspólną i rozproszoną
Fig. 1.7. MIMD computers with shared and distributed memory

1.3.1. Komputery z pamięcią wspólną

W tym modelu liczba procesorów będzie na ogół niewielka, 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 [68].

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 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 [44, 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 [66] (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 [83] 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 wieloprocesoro-

we 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 [44, 66].

1.3.3. Procesory wielordzeniowe

W ostatnich latach ogromną popularność zdobyły procesory wielordzeniowe, których pojawienie się stanowi wyzwanie dla twórców oprogramowania [14, 71]. 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 (wiosna 2008) 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 [49] wykazano, że w przypadku obliczeń z zakresu algebry liniowej szczególnie dobre wyniki daje wykorzystanie nowych sposobów reprezentacji macierzy opisanych w podrozdziale 1.2.3, co zostanie również pokazane w podrozdziale 5.4.

1.4. Optymalizacja 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* [39]), 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 [2, 3]. 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.

```
do i=1,N
   a(i)=i*(1+b)*(c+d)
end do
```

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.

```
temp1=(1+b)*(c+d)
do i=1,N
  a(i)=i*temp1
end do
```

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 (jak na przykład SSE) 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 [130]. 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.

```
do i=1,n-1
  a(i+1)=a(i)+b(i)
end do
```

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, co zostanie dokładnie omówione w podrozdziale 1.6.1. W pracy [83] 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 [3, 128, 130], 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, na ogół nie daje zadowalających rezultatów.

Zwykle konieczne staje się rozważenie czterech aspektów tworzenia efektywnych programów na komputery równoległe [40, rozdział 3.2], [44].

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 [17]. 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 gru-

pę wątków działających równolegle aż do miejsca złączenia. W programie może wystąpić wiele regionów równoległych.

```
      h=(b-a)/n
      s=0.0

! region równoległy - pętla

!$omp parallel do reduction(+:s)
      do i=1,n-1
       s=s+f(a+i*h)
      end do
!$omp end parallel do

      s=h*(s+0.5*(f(a)+f(b)))
```

Rys. 1.8. Obliczenia według wzoru (1.4) przy użyciu OpenMP
Fig. 1.8. Computing of (1.4) using OpenMP

Architektury wieloprocesorowe z pamięcią rozproszoną programuje się zwykle wykorzystując środowisko MPI (ang. *Message Passing Interface* [85]), wspierające model programu typu SPMD (ang. *Single Program, Multiple Data*) lub powoli wychodzące z użycia środowisko PVM (ang. *Parallel Virtual Machine* [37]). Program SPMD w MPI zakłada wykonanie pojedynczej instancji programu (procesu) na każdym procesorze biorącym udział w obliczeniach [61]. 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). W przypadku komunikacji *point-to-point* używa się komplementarnych operacji **send** i **receive** odpowiadających blokującemu wysyłaniu wiadomości i blokującemu odbiorowi. Operacja **send** powoduje wstrzymanie procesu wywołującego do momentu zwolnienia bufora, a operacja **receive** wstrzymanie do momentu dotarcia danych do bufora procesu wywołującego.

Rysunki 1.8 i 1.9 pokazują przykładową realizację obliczania przybliżonej wartości całki oznaczonej na podstawie wzoru złożonego trapezów [65, rozdział 7]:

$$\int_a^b f(x)dx \approx h\left(\frac{1}{2}f(x_0) + f(x_1) + \ldots + f(x_{n-1}) + \frac{1}{2}f(x_n)\right) \tag{1.4}$$

```
! zainicjowanie środowiska MPI

      call MPI_INIT( ierr )

! pobranie własnego numeru i liczby procesów

      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

! proces 0 pobiera n

      if (myid .eq. 0) then
         read (*,*) n
      endif

! operacja kolektywna - rozesłanie n do wszystkich procesów

      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

! obliczenia lokalne
      h = (b-a)/n
      s = 0.0
      do i = myid+1, n-1, numprocs
         s = s + f(a+i*h)
      end do

! operacja redukcyjna - wyznaczenie sumy obliczonych sum
! częściowych

      call MPI_REDUCE(s,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $     MPI_COMM_WORLD,ierr)

! proces 0 wyznacza ostateczną wartość

      if (myid .eq. 0) then
         sum = h * (sum + 0.5 * (f(a) + f(b)))
      endif

! koniec pracy w ramach środowiska MPI

      call MPI_FINALIZE(ierr)
```

Rys. 1.9. Obliczenia według wzoru (1.4) przy użyciu MPI

Fig. 1.9. Computing of (1.4) using MPI

gdzie $h = (b - a)/n$, $x_i = a + ih$ dla $i = 0, 1, \ldots, n$, w postaci pętli zrównoleglonej przy użyciu OpenMP oraz programu MPI w stylu SPMD. Dalsze przykłady oraz specyficzne cechy programowania równoległego przy użyciu OpenMP i MPI można znaleźć w książce [95].

Programowanie algorytmów numerycznych przy użyciu MPI może być znacznie uproszczone dzięki zastosowaniu biblioteki BLACS (ang. *Basic Linear Algebra Communication Subroutines* [38]) zbudowanej na bazie MPI. BLACS pozwala na logiczną organizację procesów w postaci dwuwymiarowej siatki $P \times Q$, gdzie każdy proces jest identyfikowany parą współrzędnych (p, q), $p = 0, \ldots, P - 1$, $q = 0, \ldots, Q - 1$. Wymiana komunikatów może dotyczyć pary procesów (nadawca i odbiorca) bądź też obejmować wiersz, kolumnę lub całą siatkę (jeden nadawca i wielu odbiorców). Podstawowym strukturalnym typem przesyłanych informacji jest blok tablicy dwuwymiarowej (prostokątny, o kształcie trapezu lub trójkątny). Oprócz tego dostępne są podprogramy do inicjowania siatki oraz odczytywania informacji o procesach w ramach siatki. Rysunek 1.10 pokazuje szkielet programu SPMD napisany przy użyciu operacji z biblioteki BLACS, w którym N procesów jest zorganizowanych w postaci siatki $P \times Q$, gdzie $N = P \cdot Q$.

```
! odczytanie informacji o własnym numerze
! oraz liczbie procesów

    call blacs_pinfo(iam,nprocs)

! inicjowanie sieci PxQ

    call blacs_get(0,0,cntx)
    call blacs_gridinit(cntx,'C',P,Q)
    ..........

! bariera - oczekiwanie na wszystkie procesy

    call blacs_barrier(cntx,'A')

! koniec działania w ramach BLACS-a

    call blacs_exit(0)
```

Rys. 1.10. Organizacja dwuwymiarowej siatki procesów przy użyciu BLACS-a
Fig. 1.10. Two-dimmensional process grid with BLACS

Innymi mniej popularnymi narzędziami programowania komputerów z pamięcią rozproszoną są języki Co-array Fortran (w skrócie CAF [40, podrozdział 12.4]) oraz Unified Parallel C (w skrócie UPC [22]). Oba rozszerzają standardowe języki Fortran i C o mechanizmy umoż-

liwiające 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, umieszczając przy odwołaniu do zmiennej właściwy numer obrazu w nawiasach kwadratowych. Przykładowo, rozesłanie wartości zmiennej do wszystkich obrazów może być zrealizowane jedną prostą instrukcją przypisania o postaci `y[:]=x`. 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. W ramach UPC zdefiniowano nową konstrukcję `upc_forall` rozszerzającą funkcjonalność standardowej pętli `for` języka C. Nakazuje ona wykonanie pętli przez grupę wątków. Przykładowo, pętla o postaci

```
upc_forall(i=0;i<n;i++;i) {
  ...
}
```

będzie wykonana w ten sposób, że iteracja dla wartości zmiennej sterującej równej `i` będzie wykonana przez wątek o numerze `i%THREADS`, gdzie poszczególne wątki są numerowane od zera do wartości `THREADS`-1. W przypadku bardziej skomplikowanym, ostatni parametr instrukcji `upc_forall` może być referencją do miejsca w pamięci wspólnej. Przykładowo, każda i-ta iteracja pętli

```
upc_forall(i=0;i<n;i++;&x[i]) {
  x[i]++;
}
```

będzie wykonywana przez ten wątek, w którego pamięci lokalnej jest alokowana dana składowa `x[i]`.

Jak wykażemy w punkcie 1.7, aktualnie jedną z najbardziej obiecujących metod konstrukcji bardzo szybkich algorytmów blokowych (operujących na macierzach), które wykorzystywałyby w dużym stopniu możliwości współczesnych procesorów, jest zastosowanie do ich konstrukcji podprogramów z biblioteki BLAS, które umożliwiają efektywne wykorzystanie hierarchii pamięci [24, 63] i zapewniają przenośność kodu między różnymi architekturami [7, 35]. Podejście to zostało z sukcesem zastosowane przy konstrukcji biblioteki LAPACK [4], zawierającej zestaw podprogramów rozwiązujących typowe zagadnienia algebry liniowej (rozwiązywanie układów równań liniowych o macierzach pełnych i pasmowych oraz algebraiczne zagadnienie

własne), jej równoległego odpowiednika – bliblioteki PLAPACK [122], zawierającej równoległe wersje algorytmów algebry liniowej oraz biblioteki ScaLAPACK [12, 55], zawierającej podprogramy realizujące najważniejsze algorytmy algebry liniowej – odpowiednik biblioteki LAPACK dla środowiska rozproszonego.

1.6. Podstawowa analiza wydajności obliczeniowej współczesnych komputerów

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 [66], [84] oraz [32] najważniejsze pojęcia z tym związane.

Definicja 1.1 ([66]). *Przyspieszeniem bezwzględnym algorytmu równoległego nazywamy wielkość*

$$s_p^\star = \frac{t_1^\star}{t_p}, \tag{1.5}$$

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 1.2 ([66]). *Przyspieszeniem względnym algorytmu równoległego nazywamy wielkość*

$$s_p = \frac{t_1}{t_p}, \tag{1.6}$$

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 1.3 ([84]). *Przyspieszeniem algorytmu wektorowego względem najszybszego algorytmu skalarnego nazywamy wielkość*

$$s_v^\star = \frac{t_s^\star}{t_v}, \tag{1.7}$$

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 1.4 ([84]). *Algorytm wektorowy rozwiązujący problem o rozmiarze n nazywamy zgodnym z najszybszym algorytmem skalarnym, gdy*

$$\lim_{n\to\infty} \frac{V(n)}{S(n)} = C < +\infty \tag{1.8}$$

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 1.2 i 1.3 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 [40, 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 1.5 ([84]). *Przyspieszeniem algorytmu A względem algorytmu B nazywamy wielkość*

$$s = \frac{t_B}{t_A}, \tag{1.9}$$

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* [32]) będziemy podawać w milionach operacji zmiennopozycyjnych na sekundę (Mflops) i definiować jako

$$r = \frac{N}{t} \text{ Mflops}, \tag{1.10}$$

gdzie N oznacza liczbę operacji zmiennopozycyjnych wykonanych w czasie t mikrosekund. Producenci sprzętu podają opierając się na wzorze (1.10) 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 (1.10) 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 (1.10) stanowi ważne kryterium porównania algorytmów przy jednoczesnym wskazaniu na stopień wykorzystania możliwości sprzętu.

[1] Innym polskim tłumaczeniem tego terminu jest *szybkość komputerów w zakresie obliczeń numerycznych* [66].

Przekształcając wzór (1.10), wnioskujemy, że czas wykonania programu spełnia następującą zależność

$$t = \frac{N}{r} \ \mu\text{s}, \tag{1.11}$$

co jest równoważne

$$t = \frac{N}{10^6 r} \ \text{s}. \tag{1.12}$$

Jak zaznaczyliśmy w podrozdziale 1.1, dla poszczególnych fragmentów programu może być osiągnięta różna wydajność, a zatem wzór (1.10) opisuje tylko średnią wydajność obliczeniową danej architektury. Poniżej przedstawimy najważniejsze modele, które znacznie lepiej oddają specyfikę wykonania programów na współczesnych komputerach.

1.6.1. 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 (1.11), otrzymujemy łączny czas wykonywania obliczeń obu części programu

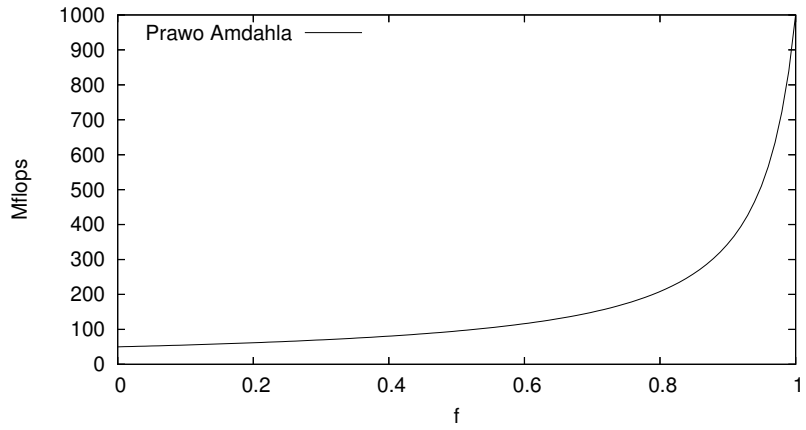
$$t = f\frac{N}{V} + (1-f)\frac{N}{S} = N(\frac{f}{V} + \frac{1-f}{S})$$

oraz uwzględniając (1.10) otrzymujemy wydajność obliczeniową komputera, który wykonuje dany program

$$r = \frac{1}{\frac{f}{V} + \frac{(1-f)}{S}} \ \text{Mflops}. \tag{1.13}$$

Wzór (1.13) nosi nazwę *prawo Amdahla* [32, 33] i opisuje wpływ optymalizacji fragmentu programu na wydajność obliczeniową danej architektury.

Jako przykład rozważmy sytuację, gdy $V = 1000$ oraz $S = 50$ Mflops. Rysunek 1.11 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.



Rys. 1.11. Prawo Amdahla dla $V = 1000$ oraz $S = 50$ Mflops
Fig. 1.11. Amdahl's Law for $V = 1000$ and $S = 50$ Mflops

1.6.2. Model Hockneya-Jesshope'a

Innym modelem, który dokładniej charakteryzuje obliczenia wektorowe jest model Hockneya - Jesshope'a obliczeń wektorowych [54, 32], pokazujący wydajność komputera wykonującego obliczenia w postaci pętli. Rozważmy pętlę o N iteracjach. Wydajność, jaką osiąga komputer wykonując takie obliczenia, wyraża się wzorem

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \text{ Mflops}, \tag{1.14}$$

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.

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

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

$$T_{DOT}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N). \tag{1.15}$$

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

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

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

$$T_{AXPY}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N). \tag{1.16}$$

Oczywiście, wielkości r_∞ oraz $n_{1/2}$ występujące odpowiednio we wzorach (1.15) i (1.16) 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.

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

$$L\mathbf{x} = \mathbf{b}, \tag{1.17}$$

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 [103]:

$$\begin{cases} x_1 = b_1/a_{11} \\ x_i = \left(b_i - \sum_{k=1}^{i-1} a_{ik}x_k\right)/a_{ii} \quad \text{dla } i = 2, \ldots, N. \end{cases} \tag{1.18}$$

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

[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*).

wzorem

$$
\begin{aligned}
T_1(N) &= \sum_{k=1}^{N-1} T_{DOT}(k) = \frac{2\cdot 10^{-6}}{r_\infty}\left(n_{1/2}(N-1) + \sum_{k=1}^{N-1} k\right) \\
&= \frac{2\cdot 10^{-6}}{r_\infty}(N-1)\left(n_{1/2} + \frac{N}{2}\right).
\end{aligned} \tag{1.19}
$$

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

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

oraz

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

Oczywiście zachodzi

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

Stąd otrzymujemy następujący wzór [100]:

$$
\begin{cases} \mathbf{y}_0 = \mathbf{b} \\ \mathbf{y}_i = L_i^{-1}\mathbf{y}_{i-1} \quad \text{dla } i = 1, \ldots, N \\ \mathbf{x} = \mathbf{y}_N \end{cases} \tag{1.20}
$$

Operacja mnożenia macierzy L_i^{-1} przez wektor $\mathbf{y}_{i-1}$ nie wymaga jawnego wyznaczania postaci macierzy. Istotnie, rozpisując wzór (1.20) otrzymujemy

$$\mathbf{y}_i = \begin{pmatrix} y_1^{(i)} \\ y_2^{(i)} \\ \vdots \\ y_i^{(i)} \\ \vdots \\ y_{N-1}^{(i)} \\ y_N^{(i)} \end{pmatrix} = L_i^{-1} \begin{pmatrix} y_1^{(i-1)} \\ y_2^{(i-1)} \\ \vdots \\ y_i^{(i-1)} \\ \vdots \\ y_{N-1}^{(i-1)} \\ y_N^{(i-1)} \end{pmatrix} = \begin{pmatrix} y_1^{(i-1)} \\ \vdots \\ y_{i-1}^{(i-1)} \\ y_i^{(i-1)}/a_{ii} \\ y_{i+1}^{(i-1)} - a_{i+1,i} y_i^{(i-1)}/a_{ii} \\ \vdots \\ y_N^{(i-1)} - a_{N,i} y_i^{(i-1)}/a_{ii} \end{pmatrix} \tag{1.21}$$

W algorytmie opartym na wzorach (1.20) i (1.21) 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 (1.21) przyjmie postać sekwencji operacji skalarnej

$$y_i \leftarrow y_i/a_{ii}, \tag{1.22}$$

a następnie wektorowej

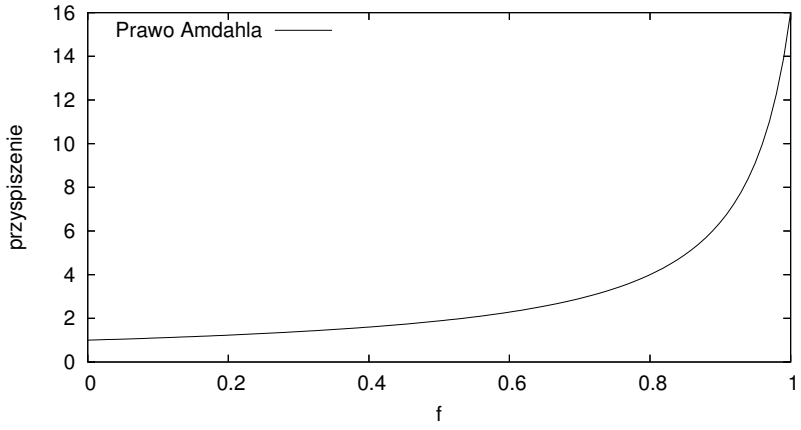
$$\begin{pmatrix} y_{i+1} \\ \vdots \\ y_N \end{pmatrix} \leftarrow \begin{pmatrix} y_{i+1} \\ \vdots \\ y_N \end{pmatrix} - y_i \begin{pmatrix} a_{i+1,i} \\ \vdots \\ a_{N,i} \end{pmatrix}. \tag{1.23}$$

Zauważmy, że (1.23) to właśnie operacja `AXPY`. Stąd podobnie jak w przypadku poprzedniego algorytmu, pomijając czas potrzebny do wykonania dzielenia (1.22), otrzymujemy

$$\begin{aligned} T_2(N) &= \sum_{k=1}^{N-1} T_{AXPY}(N-k) = \frac{2 \cdot 10^{-6}}{r_\infty} \left(n_{1/2}(N-1) + \sum_{k=1}^{N-1} (N-k) \right) \\ &= \frac{2 \cdot 10^{-6}}{r_\infty} (N-1) \left(n_{1/2} + \frac{N}{2} \right). \end{aligned} \tag{1.24}$$

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

Algorytm DOT	$r_\infty = 18$	$n_{1/2} = 36$	$T_1(1000) = 0.059$ s.
Algorytm AXPY	$r_\infty = 16$	$n_{1/2} = 26$	$T_2(1000) = 0.066$ s.



Rys. 1.12. Prawo Amdahla dla obliczeń równoległych, $p = 16$
Fig. 1.12. Amdahl's Law – parallel computing, $p = 16$

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

1.6.3. Prawo Amdahla dla obliczeń równoległych

Prawo Amdahla ma swój odpowiednik również dla obliczeń równoległych [32]. Przypuśćmy, że czas wykonania programu na jednym procesorze wynosi t_1. Niech f oznacza część programu, która może być idealnie zrównoleglona na p procesorach. Pozostała sekwencyjna część programu $(1 - f)$ będzie wykonywana na jednym procesorze. Łączny czas wykonania programu równoległego przy użyciu p procesorów wynosi

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

Stąd przyspieszenie w sensie definicji 1.2 wyraża się wzorem [32]:

$$s_p = \frac{t_1}{t_p} = \frac{p}{f + (1 - f)p}. \tag{1.25}$$

Rysunek 1.12 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 (1.13).

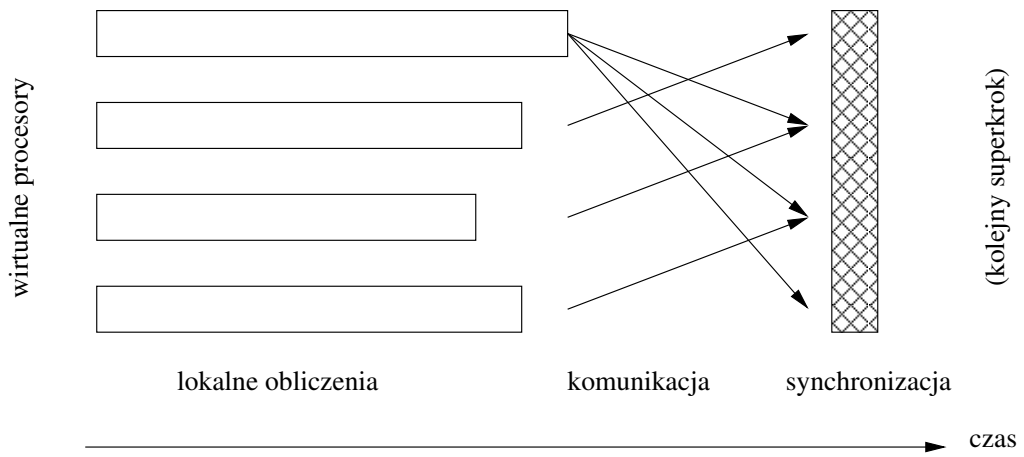
1.6.4. Model BSP

W celu przeprowadzania analizy wykonania programów w środowisku rozproszonym bez wspólnej pamięci rozważmy następujący model BSP (ang. *Bulk Synchronous Parallel Archi-*

tecture) [11, 53]. Program równoległy składa się z pewnej liczby superkroków (rysunek 1.13). Każdy superkrok składa się z obliczeń wykonywanych przez procesory na danych znajdujących się w ich pamięciach lokalnych, globalnej wymiany danych (komunikacji) oraz na koniec synchronizacji. Model BSP charakteryzuje się następującymi parametrami: liczbą dostępnych procesorów p, czasem g (liczonym w jednostkach równych czasowi wykonania jednej operacji zmiennopozycyjnej) potrzebnym do wysłania bądź odbioru jednego słowa maszynowego oraz czasem l (również liczonym w jednostkach określonych przez czas wykonania operacji zmiennopozycyjnej) potrzebnym do synchronizacji wszystkich procesorów. Wykonanie operacji synchronizacji na koniec superkroku gwarantuje, że wszystkie wysyłane dane dotarły do miejsca przeznaczenia. Złożoność (inaczej koszt) superkroku definiujemy jako wielkość

$$C_{step} = w_{\max} + gh_{\max} + l, \tag{1.26}$$

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



Rys. 1.13. Model obliczeniowy BSP: struktura superkroku
Fig. 1.13. The BSP model of computations: the structure of a superstep

Tabela 1.1
BLAS: odwołania do pamięci, liczba operacji arytmetycznych oraz ich stosunek, przy założeniu że $n = m = k$ [32]

BLAS (operacja)		pamięć	operacje	*ratio*
$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$	(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)	$2mn + mk + kn$	$2mkn + 2mn$	$2:n$

1.7. BLAS: podstawowe podprogramy algebry liniowej

W roku 1979 zaproponowano standard dla podprogramów realizujących podstawowe operacje algebry liniowej (ang. *Basic Linear Algebra Subprograms* – BLAS) [74]. 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 [29], 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 [34].

Następnym krokiem w rozwoju standardu BLAS były prace nad biblioteką LAPACK [4], wykorzystującą algorytmy blokowe, której funkcjonalność pokryła bibliotekę LINPACK oraz EISPACK [43], zawierającą podprogramy do rozwiązywania algebraicznego zagadnienia własnego. Zdefiniowano zbiór podprogramów zawierających działania typu macierz - wektor (biblioteka BLAS poziomu drugiego [31]) oraz macierz - macierz (inaczej BLAS poziomu trzeciego [30]), 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.

Tabela 1.1 pokazuje zalety użycia wyższych poziomów BLAS-u [32]. Dla reprezentatywnych operacji z poszczególnych poziomów (kolumna 1) podaje liczbę odwołań do pamięci (ko-

lumna 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. Aby zilustrować wpływ tego faktu na szybkość obliczeń, rozważmy cztery równoważne sobie algorytmy mnożenia macierzy, każdy wykonujący identyczną liczbę działań arytmetycznych. Algorytm 1.1, to klasyczne mnożenie macierzy, a algorytmy 1.2, 1.3, 1.4 wykorzystują odpowiednie operacje z kolejnych poziomów BLAS-u. 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. Dodatkowo przyjmijmy, że $M_{i:j,*} = M_{i:j,1:n}$, $M_{*,k:l} = M_{1:m,k:l}$ oraz $M_{i:j,k} = M_{i:j,k:k}$, $M_{i,k:l} = M_{i:i,k:l}$.

Algorytm 1.1. 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: $t \leftarrow 0$
4: **for** $l = 1$ to k **do**
5: $t \leftarrow t + a_{il} b_{lj}$
6: **end for**
7: $c_{ij} \leftarrow \beta c_{ij} + \alpha t$
8: **end for**
9: **end for**

Algorytm 1.2. 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: $C_{*j} \leftarrow \beta C_{*j}$ {operacja `SCAL`}
3: **for** $i = 1$ to k **do**
4: $C_{*j} \leftarrow C_{*j} + (\alpha b_{ij}) A_{*i}$ {operacja `AXPY`}
5: **end for**
6: **end for**

Tabela 1.2 pokazuje czas działania i wydajność osiąganą przy wykonaniu poszczególnych algorytmów na trzech różnych komputerach, przy czym $m = n = k = 1000$. W każdym przy-

Algorytm 1.3. 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: $C_{*j} \leftarrow \alpha AB_{*j} + \beta C_{*j}$ {operacja `GEMV`}

3: **end for**

Algorytm 1.4. 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`}

padku widać, że algorytm wykorzystujący wyższy poziom BLAS-u jest istotnie szybszy. Co więcej, wykonanie algorytmu 1.4 odbywa się z wydajnością bliską teoretycznej maksymalnej. W przypadku procesorów Pentium czas wykonania każdego algorytmu wykorzystującego BLAS jest mniejszy niż czas wykonania algorytmu 1.1, który wykorzystuje jedynie kilka procent wydajności procesorów. W przypadku komputera Cray X1 czas wykonania algorytmu 1.1 utrzymuje się na poziomie czasu wykonania algorytmu wykorzystującego BLAS poziomu 2, co jest wynikiem doskonałej optymalizacji prostego kodu algorytmu 1.1, realizowanej przez kompilator Cray Fortran, który jest powszechnie uznawany za jeden z najlepszych kompilatorów optymalizujących. Zauważmy, że stosunkowo słabą optymalizację kodu przeprowadza kompilator Intel Fortran, 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 1.3 pokazuje wydajność, czas działania oraz przyspieszenie w sensie definicji 1.2 dla algorytmów 1.1, 1.2, 1.3 oraz 1.4 na dwuprocesorowym komputerze Quad-Core Xeon. Można zauważyć, że algorytmy 1.1, 1.2 i 1.3 wykorzystują niewielki procent wydajności komputera, a dla algorytmu 1.4 osiągana jest bardzo duża wydajność. Wszystkie algorytmy dają się dobrze zrównoleglić, choć najlepsze przyspieszenie względne jest osiągnięte dla algorytmów 1.1 i 1.2.

Na zakończenie dodajmy, 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 [25]. Biblioteka PBLAS (ang. *parallel BLAS*, [21]) zawiera podprogramy realizujące podstawowe operacje algebry liniowej w środowisku rozproszonym, wykorzystuje lokalnie BLAS oraz BLACS w warstwie komunikacyjnej.

Tabela 1.2
Wydajność i czas wykonania algorytmów mnożenia macierzy na różnych procesorach dla wartości $m = n = k = 1000$

	PIII 866MHz		P4 3GHz HT		Cray X1, 1 MSP	
alg.	Mflops	sec.	Mflops	sec.	Mflops	sec.
1.1	93.98	21.28	282.49	7.08	7542.29	0.27
1.2	94.65	21.13	1162.79	1.72	587.41	3.40
1.3	342.46	5.83	1418.43	1.40	7259.48	0.28
1.4	1398.60	1.43	7692.30	0.26	16369.89	0.12

Tabela 1.3
Wydajność, czas wykonania i przyspieszenie względne algorytmów mnożenia macierzy na dwuprocesorowym komputerze Xeon Quad-Core dla wartości $m = n = k = 1000$

	1 core		Quad-Core			2x Quad-Core		
alg.	Mflops	czas (s)	Mflops	czas (s)	s_p	Mflops	czas (s)	s_p
1.1	350.9	5.6988	1435.0	1.3937	4.09	2783.4	0.7185	7.93
1.2	1573.8	1.2708	6502.0	0.3076	4.13	12446.1	0.1607	7.90
1.3	4195.5	0.4767	13690.1	0.1461	3.26	22326.9	0.0896	5.32
1.4	16026.3	0.1248	54094.9	0.0370	3.38	86673.5	0.0231	5.41

1.8. Liniowe równania rekurencyjne

W podrozdziale1.5 wskazaliśmy na istnienie fragmentów kodu mających postać rekurencji, jako na jedną z ważniejszych przyczyn ograniczających pełne wykorzystanie mocy obliczeniowych oferowanych przez współczesne komputery wektorowe i wieloprocesorowe. W niniejszej pracy zajmujemy się efektywnymi metodami optymalizacji obliczeń rekurencyjnych na architektury wektorowe i równoległe. W odróżnieniu od stosownych metod transformacji pętli [13], które prowadzą co najwyżej do wektoryzacji obliczeń, wykorzystamy metody algebry liniowej wychodząc od matematycznego modelu tego typu konstrukcji algorytmicznych.

Rozważmy zatem następujący problem [80, 100]. Dla danych współczynników f_k oraz a_{kj}, $k = 1, \ldots, n$, $j = 1, \ldots, m$, gdzie $n \gg m$, należy wyznaczyć n liczb x_k, $k = 1, \ldots, n$, spełniających :

$$x_k = \begin{cases} 0 & \text{dla } k \leq 0 \\ f_k + \sum\limits_{j=1}^{m} a_{kj} x_{k-j} & \text{dla } 1 \leq k \leq n, \end{cases} \tag{1.27}$$

bądź w szczególnym przypadku

$$x_k = \begin{cases} 0 & \text{dla } k \leq 0 \\ f_k + \sum_{j=1}^{m} a_j x_{k-j} & \text{dla } 1 \leq k \leq n. \end{cases} \tag{1.28}$$

Równanie postaci (1.27) będziemy za książką [69] nazywać *liniowym równaniem rekurencyjnym rzędu m*, a w przypadku (1.28) *liniowym równaniem rekurencyjnym rzędu m o stałych współczynnikach*. Znalezienie liczb x_k będziemy nazywać wyznaczeniem rozwiązania liniowego równania rekurencyjnego.

Równania (1.27) oraz w szczególności (1.28) występują jako element składowy wielu algorytmów rozwiązujących ważne problemy obliczeniowe. Należy tutaj wymienić schemat Hornera wyznaczania wartości wielomianu [103], wyznaczanie wartości wielomianów ortogonalnych definiowanych rekurencyjnie [9], obliczanie sum trygonometrycznych [103] wykorzystywanych w interpolacji trygonometrycznej oraz przy numerycznym wyznaczaniu odwrotności transformanty Laplace'a [121, 82], rozwiązywaniu układów równań liniowych o macierzach pasmowych [36, 62], algorytmach wyznaczania wartości własnych [103] i wielu innych. Równanie (1.28) jest również znane jako część filtra rekurencyjnego, mającego ważne znaczenie w teorii i praktyce analizy sygnałów [101]. Przykładem nienumerycznego algorytmu realizującego obliczenia w postaci rekurencyjnej (1.28) jest sortowanie przez zliczanie [23].

Oczywiście zakodowanie najprostszego algorytmu wyznaczania rozwiązania równania postaci (1.27) lub (1.28) nie przedstawia z pozoru żadnych trudności. Algorytm 1.5 stanowi przykład rozwiązania problemu (1.28). Jednak taki sekwencyjny (skalarny) algorytm w postaci dwóch pętli, jedna zagnieżdżona w drugiej, ma kilka wad. Po pierwsze, nie nadaje się on do bezpośredniego zrównoleglenia, a zatem jego wykonanie nie będzie mogło w pełni wykorzystać sprzętu wieloprocesorowego. Po drugie, wykonanie takiego algorytmu na jednym procesorze będzie się wiązać z bardzo niewielkim wykorzystaniem teoretycznej, maksymalnej wydajności procesora. Wektoryzacja pętli wewnętrznej jest na ogół nieopłacalna ze względu na raczej niewielkie wartości m.

Warto tutaj wspomnieć, że kompilatory optymalizujące zwykle nie są w stanie wygenerować skalowalnego kodu maszynowego, który umożliwiłby efektywne obliczenia dla algorytmów rekurencyjnych [128, 130]. Istnieje zatem potrzeba sformułowania szybkich algorytmów, wykorzystujących w dużym stopniu możliwości oferowane przez procesory i jednocześnie dających się zrównoleglić, co umożliwiłoby efektywne wykorzystanie architektury współczesnych komputerów wieloprocesorowych. Bardzo pożądaną cechą takich algorytmów byłaby również przenośność między różnymi typami architektur.

Algorytm 1.5. Sekwencyjne (skalarne) wyznaczanie rozwiązania liniowego równania rekurencyjnego o stałych współczynnikach (1.28).

Wejście: liczby $f_1, \ldots, f_n$ oraz $a_1, \ldots, a_m$, przy czym początkowo $x_k = f_k$ dla $k = 1, \ldots, n$

Wyjście: $x_1, \ldots, x_n$ spełniające (1.28)

1: **for** $k = 1$ to n **do**
2: **for** $j = 1$ to $\min\{m, k-1\}$ **do**
3: $x_k \leftarrow x_k + a_j x_{k-j}$
4: **end for**
5: **end for**

Przedstawimy teraz pokrótce uzyskane przez innych autorów najważniejsze wyniki dotyczące rozważanego problemu rozwiązywania liniowych równań rekurencyjnych. Podstawowy wynik dotyczący złożoności obliczeniowej równoległego rozwiązywania liniowych równań rekurencyjnych pochodzi z pracy [19] i mówi, że wyznaczanie rozwiązania równania (1.27) wymaga

$$T = (2 + \log m) \log n - \frac{1}{2}(1 + \log m) \log m \tag{1.29}$$

kroków pracy komputera równoległego typu MIMD, a wymagana do tego liczba procesorów p spełnia ograniczenie

$$p \leq \begin{cases} m(m+1)n/2 + O(m^3) & \text{dla } 1 \leq m \leq n/2 \\ n^3/68 & \text{dla } n/2 \leq m \leq n-1. \end{cases} \tag{1.30}$$

Oczywiście zacytowany wynik ma charakter wyłącznie teoretyczny i efektywna realizacja algorytmu zaprezentowanego w pracy [19] na współczesnych komputerach wieloprocesorowych jest praktycznie niemożliwa ze względu na dużą liczbę wymaganych procesorów w stosunku do rozmiaru rozwiązywanego problemu. Podobnie teoretyczne znaczenie mają wyniki opisane w pracach, [45], [56], [99], [125]. Z kolei w pracach [42], [15], [16] zaprezentowano techniki konstrukcji algorytmów rozwiązywania liniowych równań rekurencyjnych na specyficzne architektury wieloprocesorowe typu *mesh-connected* oraz procesory macierzowe typu SIMD, które wyszły z użycia.

W pracy [20] przedstawiono modyfikację wyniku dotyczącego optymalnej złożoności problemu opisanej wzorem (1.30). Dla liczby procesorów $p = km \ll n$, równanie rekurencyjne (1.27) może być rozwiązane w czasie

$$(2m^2 n/p) + (3mn/p) + O(m^2 \log(p/m)),$$

a m ostatnich składowych może być wyznaczone w liczbie kroków określonej wzorem

$$(2m^2n/p) + (mn/p) + O(m^2 \log(p/m)),$$

również przy użyciu $p = km$ procesorów. Przyspieszenie algorytmu zaprezentowanego w pracy [20] wynosi $p/(2m)$, a zatem spada wraz ze wzrostem rzędu rozwiązywanego równania. Oznacza to, że proponowany algorytm nie spełnia postulatu skalowalności, czyli zachowania lub nawet wzrostu efektywności wraz ze wzrostem rozmiaru rozwiązywanego problemu.

Z praktycznego punktu widzenia najbardziej przydatne są trzy metody oraz ich modyfikacje. Pierwsza z nich *recursive doubling* [32, 80, 86], w przypadku $m = 1$ polega na dokonywaniu przekształceń, które przyjmą postać

$$x_k = f_k + a_{k,1}x_{k-1} = f_k + a_{k,1}(f_{k-1} + a_{k-1,1}x_{k-2}) = \tilde{\tilde{f}}_k + \tilde{a}_{k,1}x_{k-2}.$$

Nowe współczynniki $\tilde{\tilde{f}}_k$, $\tilde{a}_{k,1}$ mogą być obliczane równolegle bądź wektorowo, co raczej nie jest zalecane ze względu na niewielką długość wektoryzowanych pętli. Powyższe przekształcenie może być powtarzane: w pierwszym kroku zastępujemy we wzorze na x_n wyraz x_{n-1} za pomocą wzoru na x_{n-1}, czyli przy użyciu oraz x_{n-2}, podobnie wyrażamy x_{n-2} przez x_{n-3} itd. W kolejnym kroku zastępujemy we wzorze na x_n wyraz x_{n-2} wzorem na x_{n-2}, czyli przez x_{n-3} itd.

Kolejna metoda, bardziej efektywna z praktycznego punktu widzenia (szczególnie w przypadku obliczeń wektorowych) to *cyclic reduction* [73, 78], w której pierwszym kroku obliczamy x_k, gdzie k jest parzyste, a w kroku drugim obliczamy pozostałe x_k za pomocą jednej operacji wektorowej.

Trzecia metoda znana jest pod nazwą metody podziału (*divide and conquer*) lub metody Wanga [124, 73, 104]. Polega ona na zastąpieniu równania (1.27) układem równań liniowych. Następnie taki układ jest zapisywany w postaci blokowej dwudiagonalnej

$$\begin{pmatrix} L_1 & & & \\ U_2 & L_2 & & \\ & \ddots & \ddots & \\ & & U_p & L_p \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_p \end{pmatrix} \tag{1.31}$$

lub w przypadku (1.28) blokowej dwudiagonalnej typu Toeplitza [91]

$$\begin{pmatrix} L & & & \\ U & L & & \\ & \ddots & \ddots & \\ & & U & L \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_p \end{pmatrix}, \tag{1.32}$$

gdzie w obu przypadkach p oznacza liczbę procesorów. Stąd otrzymujemy

$$\begin{cases} \mathbf{x}_1 = L_1^{-1}\mathbf{f}_1 \\ \mathbf{x}_j = L_j^{-1}\mathbf{f}_j - L_j^{-1}U_j\mathbf{x}_{j-1} \quad \text{dla } j = 2, \ldots, p. \end{cases} \tag{1.33}$$

Algorytm przebiega w trzech fazach. Po pierwsze, równolegle wyznaczane są rozwiązania układów równań

$$L_j\mathbf{z}_j = \mathbf{f}_j, \tag{1.34}$$

co jest równoważne wyznaczeniu rozwiązań p liniowych równań rekurencyjnych rzędu m i może być dokonane za pomocą skalarnego algorytmu (1.27). Następnie sekwencyjnie wyznacza się m ostatnich składowych wektorów $\mathbf{x}_2, \ldots, \mathbf{x}_p$, później wyznacza się równolegle pierwsze $q - m$ składowych tych wektorów, gdzie $pq = n$.

Szczególne przypadki omówionych algorytmów, poświęcone ich wektoryzacji dla równań rzędu pierwszego, opisano w pracach [5], [73], [123], [126]. Prace [64, 129] przedstawiają metodę analizy zależności danych w obliczaniu liniowych równań rekurencyjnych rzędu pierwszego, co umożliwia sformułowanie algorytmów wykorzystujących mechanizmy odpowiedniego szeregowania rozkazów maszynowych, co jednak wiąże się z trudnościami przy ich przenoszeniu na inne rodzaje architektur. Należy tutaj podkreślić, że wszystkie omówione algorytmy wymagają na ogół dużej liczby procesorów dla osiągnięcia istotnego przyspieszenia oraz ich zastosowanie na komputerach z niewielką liczbą procesorów może wiązać się z wolniejszym czasem działania, niż ma to miejsce dla prostego skalarnego algorytmu opartego na wzorze (1.27) lub (1.28). Jest to spowodowane tym, że wspomniane algorytmy równoległe bądź wektorowe wymagają większej liczby działań niż algorytm skalarny. Dla przypadku $m = 1$, algorytm *recursive doubling* wymaga łącznie $3n \log_2 n$ działań arytmetycznych, czyli nie jest zgodny z algorytmem 1.5 w sensie definicji 1.4. Algorytm *cyclic reduction* wymaga $5n - 3$ działań [5]. W obu przypadkach liczba operacji jest zatem istotnie większa od liczby operacji $2n - 2$, których wymaga prosty algorytm oparty na wzorze (1.27).

Również wspomniane algorytmy wektorowe nie dają dużego przyspieszenia w stosunku do algorytmu skalarnego. Dodatkowo, nie działają szybciej na popularnych procesorach Intela i AMD, gdyż nie wykorzystują w odpowiednim stopniu pamięci podręcznej, a przyspieszenie uzyskane dzięki wektoryzacji przy wykorzystaniu mechanizmów SSE jest „pochłaniane” przez większą liczbę operacji.

W dalszych rozdziałach podamy metody konstrukcji szybkich algorytmów realizujących obliczenia typu (1.28) oraz pewnych szczególnych, mających praktyczne zastosowanie, obliczeń typu (1.27). Algorytmy zostaną wyrażone w terminach operacji z różnych poziomów biblioteki BLAS. Dzięki temu będą się charakteryzować łatwą przenośnością między różnymi

rodzajami procesorów, możliwością dobrego wykorzystania mechanizmów wektorowych oraz mechanizmów użycia pamięci podręcznej oraz łatwym zrównoleglaniem na komputery z pamięcią wspólną i rozproszoną. Będą się również charakteryzować dobrą skalowalnością, jeśli chodzi o rozmiar problemu (wartości n, m), a zatem będą spełniać wszystkie postulaty konstrukcji efektywnych algorytmów omówionych w tym rozdziale.

1.9. Przykłady architektur komputerowych

Przedstawimy teraz przykładowe architektury komputerów, które były wykorzystane do testów algorytmów prezentowanych w dalszych rozdziałach niniejszej pracy.

Dwuprocesorowy komputer Pentium III

Działający pod kontrolą systemu operacyjnego Linux komputer z dwoma procesorami Pentium III 866 MHz (pamięć podręczna L1 po 16 KB na instrukcje i dane, 512 KB pamięci podręcznej L2, rozszerzenia MMX i SSE), wyposażony w 512 MB pamięci SDRAM. W testach wykorzystano kompilator *Intel Fortran 7.0* wspierający standard OpenMP i bibliotekę *Intel Math Kernel Library 5.2* zawierającą implementację podprogramów z biblioteki BLAS.

Komputer Pentium 4

Działający pod kontrolą systemu operacyjnego Linux komputer z procesorem Pentium 4 HT 3.2 GHz (256 KB pamięci podręcznej L2, rozszerzenia MMX, SSE i SSE2) ze sprzętową implementacją wielowątkowości (ang. *Hyper-Threading*), dzięki której dwa niezależne wątki mogą jednocześnie korzystać z jednostek wykonawczych procesora, co w przypadku obliczeń numerycznych z zakresu algebry liniowej daje wzrost wydajności około 20%. Z poziomu systemu operacyjnego procesor z technologią HT jest rozpoznawany jako dwa logiczne procesory. Komputer był wyposażony w 512 MB pamięci SDRAM. W testach wykorzystano kompilator *Intel Fortran 7.0*.

Dwuprocesorowy komputer Quad-Core Xeon

Działający pod kontrolą systemu operacyjnego Linux komputer z dwoma procesorami Quad-Core Xeon 2.33 GHz z pamięcią podręczną L2 o pojemności 8 MB (2x4 MB) i pamięcią operacyjną 4 GB. Z poziomu systemu operacyjnego komputer jest rozpoznawany jako wyposażony w osiem procesorów. Do testów użyto kompilatora GNU Fortran g95 wspierającego standard OpenMP oraz biblioteki ATLAS w wersji 3.6.0, zawierającej podprogramy BLAS.

Dwunastoprocesorowy komputer SUN UltraSPARC II

Komputer działający pod kontrolą systemu operacyjnego SunOS z rodziny systemów Unix, wyposażony w dwanaście procesorów UltraSPARC II z zegarem 400 MHz. Do testów użyto dostępnego na komputerze kompilatora języka Fortran ze wsparciem dla OpenMP oraz biblioteki zawierającej implementację BLAS.

Klaster piętnastu komputerów Pentium III 667 MHz

Klaster piętnastu działających pod kontrolą systemu operacyjnego Linux komputerów z procesorami Intel Pentium III 667 MHz, wyposażonych w 256 MB SDRAM, połączonych siecią Fast Ethernet (100 Mb/s). Do testów użyto kompilatora GNU g77, biblioteki ATLAS w wersji 3.6.0 zawierającej podprogramy BLAS oraz środowiska MPICH w wersji 1.2.7p1 będącego implementacją standardu MPI [46].

Klaster dwudziestu czterech procesorów Itanium 2

Klaster składał się z dwunastu węzłów dwuprocesorowych Intel Itanium 2 połączonych siecią Gigabit Ethernet (1 Gb/s). Każdy węzeł posiadał 4 GB pamięci operacyjnej. Procesory Itanium 2 charakteryzowały się zegarem 1.4 GHz oraz posiadały 3 MB pamięci podręcznej L2. Trzeba tutaj podkreślić, że architektura procesora Itanium 2 bazuje na koncepcji bardzo długiego słowa maszynowego (ang. *Very Long Instruction Word* – VLIW). Każde słowo zawiera trzy instrukcje i zadaniem kompilatora jest właściwe rozdzielanie ich sekwencji do wykonania na poszczególnych jednostkach wykonawczych procesora (więcej informacji na ten temat można znaleźć w książce [66] oraz na stronie internetowej producenta[3]). Do testów użyto kompilatora *Intel Fortran 8.0*, biblioteki *Intel Math Kernel Library 7.2*, środowiska MPICH w wersji 1.2.7p1 oraz bibliotek BLACS i PBLAS dostępnych w postaci źródeł[4] i skompilowanych na tę architekturę. Dla testów algorytmów na pojedynczych węzłach wykorzystano standard OpenMP, którego wsparcie oferuje kompilator *Intel Fortran*.

Cray C90

Komputer wyposażony w szesnaście procesorów wektorowych, każdy o teoretycznej maksymalnej wydajności 1000 Mflops. Komputer działał pod kontrolą systemu operacyjnego UniCOS z rodziny Unix. Do testów użyto kompilatora *Cray Fortran* oraz biblioteki *Cray scilib*. Testy algorytmów wektorowych z rozdziału 2 były przeprowadzane dla pojedynczego procesora.

[3] `http://www.intel.com/design/itanium/arch_spec.htm`
[4] `http://www.netlib.org`

Cray SV1

Działający pod kontrolą systemu operacyjnego UniCOS, wyposażony w maksymalnie trzydzieści dwa procesory komputer wektorowy. W testach algorytmów wektorowych z rozdziału 2 wykorzystano jeden procesor w starszej wersji SV1 (zegar 300 MHz) oraz dla testów algorytmów z rozdziału 3, czterech procesorów nowszej wersji SV1ex z zegarem 500 MHz o teoretycznej maksymalnej wydajności 2000 Mflops dla każdego procesora. System kolejkowy na tym komputerze umożliwiał rezerwację dla programu maksymalnie czterech procesorów, a próba użycia większej ich liczby (przykładowo ośmiu) nie dawała spodziewanego przyspieszenia (osiem procesów było fizycznie wykonywanych na czterech procesorach). Do testów użyto kompilatora *Cray Fortran* oraz biblioteki *Cray scilib*.

Cray X1

Ciekawym przykładem komputerowej architektury wieloprocesorowej jest jest Cray X1, który pojawił się na rynku w roku 2003. Każdy procesor MSP komputera składa się z czterech procesorów SSP. Możliwa jest kompilacja programu do pracy w trybie MSP oraz SSP. Każdy procesor typu SSP jest wyposażony w szybką jednostkę wektorową oraz bardzo wolną (400 MHz) jednostkę do obliczeń skalarnych. Cztery procesory MSP działają w trybie SMP (ang. *symmetric multiprocessing*) i mają dostęp do wspólnej pamięci, tworząc węzeł obliczeniowy. Poszczególne węzły połączone są szybką magistralą. Aby uzyskać dostęp do więcej niż jednego węzła, trzeba się posługiwać systemami programowania rozproszonego MPI, CAF, UPC. Teoretyczna maksymalna wydajność jednego procesora MSP wynosi 12.8 Gflops, a wydajność jednostki skalarnej procesora SSP to zaledwie 100 Mflops. Stosowanie MSP staje się sensowne, gdy wykorzystujemy co najmniej drugi poziom biblioteki BLAS. W pracy [83] zawarto sugestię, że jeśli w programie dominują obliczenia skalarne, należy raczej zrezygnować z zastosowania komputera Cray X1. Do testów użyto kompilatora *Cray Fortran* oraz biblioteki *Cray scilib*.

2. Wektoryzacja obliczeń rekurencyjnych

W niniejszym rozdziale przedstawimy metodę efektywnej wektoryzacji algorytmu *divide and conquer* wyznaczania rozwiązania liniowych równań rekurencyjnych o stałych współczynnikach (1.28), wykorzystującą BLAS poziomu pierwszego, a ściślej operację `AXPY` oraz przy pewnych dodatkowych założeniach, również operację `GEMV` z poziomu drugiego biblioteki BLAS. Pokażemy, że zastosowane podejście umożliwia osiągnięcie znacznego przyspieszenia na procesorach wektorowych.

2.1. Wariant metody *divide and conquer*

W pracy [104] przedstawiliśmy modyfikację metody *divide and conquer* [20] wyznaczania rozwiązania (1.27), polegającą na uwzględnieniu postaci macierzy U_j, a co za tym idzie, innym sposobie wyznaczania wektorów $\mathbf{x}_j$. Mianowicie, macierze U_j mogą być zapisane jako

$$\begin{aligned} U_j &= \begin{pmatrix} & -a_{(j-1)q+1,(j-1)q-m+1} & \cdots & -a_{(j-1)q+1,(j-1)q} \\ & & \ddots & \vdots \\ & & & -a_{(j-1)q+m,(j-1)q} \\ 0 & & & \end{pmatrix} \\ &= -\sum_{k=1}^{m}\sum_{l=k}^{m} a_{(j-1)q+k,(j-1)q-m+l}\mathbf{e}_k\mathbf{e}_{q-m+l}^T, \end{aligned} \tag{2.1}$$

przy czym wektory $\mathbf{e}_k$ mają następującą postać

$$\mathbf{e}_k = (\underbrace{0,\ldots,0}_{k-1},1,0,\ldots,0)^T \in \mathbb{R}^q.$$

Stąd wzór (1.33) przyjmie postać [104]:

$$\begin{cases} \mathbf{x}_1 = L_1^{-1}\mathbf{f}_1 \\ \mathbf{x}_j = L_j^{-1}\mathbf{f}_j + \sum\limits_{k=1}^{m} \alpha_j^k \mathbf{y}_j^k \quad \text{dla } j = 2,\ldots,p, \end{cases} \tag{2.2}$$

gdzie

$$\alpha_j^k = \sum_{l=k}^{m} a_{(j-1)q+k,(j-1)q-m+l}\mathbf{e}_{q-m+l}^T\mathbf{x}_{j-1}$$

oraz wektory $\mathbf{y}_j^k$ są rozwiązaniami układów równań liniowych o postaci

$$L_j \mathbf{y}_j^k = \mathbf{e}_k \text{ dla } k = 1, \ldots, m.$$

W cytowanej pracy podano również obliczenia pokazujące, że na architekturze typu tablica procesorów, w przypadku niewielkiej liczby dostępnych procesorów, algorytm powinien działać nieco szybciej niż klasyczna metoda Wanga. Potwierdziły to wyniki eksperymentów przeprowadzonych na komputerze Sequent Symmetry, które przedstawiono szczegółowo w pracy [90]. Niestety, wyniki eksperymentów przeprowadzonych na wektorowym komputerze Cray Y-MP pokazały [89], że w przypadku równań rzędu pierwszego, klasyczny algorytm *divide and conquer* działa szybciej niż opisany algorytm zmodyfikowany. Jednakże w przypadku równań o stałych współczynnikach, algorytm zmodyfikowany wykazuje pewną przewagę. Istotnie, w tym przypadku wszystkie macierze L_j, $U_j \in \mathbb{R}^{q \times q}$ przyjmują postać

$$L_j = L = \begin{pmatrix} 1 & & & & & \\ -a_1 & \ddots & & & & \\ \vdots & \ddots & \ddots & & & \\ -a_m & & \ddots & \ddots & & \\ & \ddots & & \ddots & \ddots & \\ & & -a_m & \cdots & -a_1 & 1 \end{pmatrix} \tag{2.3}$$

oraz

$$U_j = U = \begin{pmatrix} & -a_m & \cdots & -a_1 \\ & & \ddots & \vdots \\ & & & -a_m \\ 0 & & & \end{pmatrix}. \tag{2.4}$$

Stąd uwzględniając postać macierzy

$$U = -\sum_{k=1}^{m} \sum_{l=k}^{m} a_{m+k-l} \mathbf{e}_k \mathbf{e}_{q-m+l}^T,$$

oraz kładąc $L\mathbf{z}_j = \mathbf{f}_j$, otrzymujemy następujący wzór stanowiący szczególny przypadek wzoru (2.2)

$$\begin{cases} \mathbf{x}_1 = \mathbf{z}_1 \\ \mathbf{x}_j = \mathbf{z}_j + \sum\limits_{k=1}^{m} \alpha_j^k \mathbf{y}_k \quad \text{dla } j = 2, \ldots, p, \end{cases} \tag{2.5}$$

gdzie

$$\alpha_j^k = \sum_{l=k}^{m} a_{m+k-l} x_{(j-1)q-m+l} \tag{2.6}$$

oraz wektory $\mathbf{y}_k$ spełniają zależności

$$L\mathbf{y}_k = \mathbf{e}_k \text{ dla } k = 1, \ldots, m.$$

W pracy [89] pokazaliśmy również, że do wyznaczenia wszystkich wektorów $\mathbf{y}_k$ wystarczy jedynie wcześniejsze wyznaczenie rozwiązania układu $L\mathbf{y}_1 = \mathbf{e}_1$. Pozostałe wektory $\mathbf{y}_k$ mogą być obliczone z następującego wzoru

$$\mathbf{y}_k = (\underbrace{0, \ldots, 0}_{k-1}, 1, y_2, \ldots, y_{q-k+1})^T, \tag{2.7}$$

przy czym

$$\mathbf{y}_1 = (1, y_2, \ldots, y_q)^T.$$

Oznacza to, że w celu rozwiązania liniowego równania rekurencyjnego rzędu m trzeba wyznaczyć rozwiązania $p + 1$ układów równań liniowych $L\mathbf{z}_j = \mathbf{f}_j$, $j = 1, \ldots, p$ oraz $L\mathbf{y}_1 = \mathbf{e}_1$, czyli ich liczba nie zależy od rzędu równania (wartości m). Przedstawione w pracy [89] wyniki testów pokazują, że przyspieszenie algorytmu względem algorytmu skalarnego nie maleje wraz ze wzrostem wartości m oraz algorytm jest dobrze skalowalny, to znaczy zwiększenie liczby procesorów powoduje wzrost przyspieszenia. Trzeba jednak podkreślić, że mimo wszystko algorytm wykorzystuje niewielką część mocy obliczeniowej komputera z procesorem wektorowym.

2.2. Szybki algorytm wektorowy

Bazując na opisanych wyżej spostrzeżeniach, przedstawimy teraz algorytm wektorowy wykorzystujący BLAS poziomu 1, rozwiązujący równania o stałych współczynnikach. W tym celu wybierzmy najpierw dwie liczby całkowite dodatnie r i s takie, że $rs \leq n$ oraz $s > m$. Przedstawiona poniżej metoda będzie służyła dla wyznaczenia niewiadomych $x_1, \ldots, x_{rs}$. Pozostałe wartości $x_{rs+1}, \ldots, x_n$ mogą być wyznaczone bezpośrednio z równania (1.28), o ile oczywiście zachodzi taka potrzeba, czyli $n \neq rs$.

W tym celu zauważmy, że liczby x_k, f_k stanowiące odpowiednio niewiadome oraz prawą stronę równania (1.28) mogą być potraktowane jako elementy macierzy zbudowanych z wektorów – kolumn $\mathbf{x}_1, \ldots, \mathbf{x}_r$ oraz $\mathbf{f}_1, \ldots, \mathbf{f}_r$, gdzie

$$\mathbf{x}_j = (x_{(j-1)s+1}, \ldots, x_{js})^T, \ \mathbf{f}_j = (f_{(j-1)s+1}, \ldots, f_{js})^T \in \mathbb{R}^s \tag{2.8}$$

dla $j = 1, \ldots, r$. Zauważmy, że wektor $\mathbf{z}_1$ jest tożsamy z wektorem $\mathbf{x}_1$. Wyznaczenie liczb $x_1, \ldots, x_{rs}$ stanowiących rozwiązanie równania (1.28) będzie polegało na znalezieniu elementów macierzy

$$X = (\mathbf{x}_1, \ldots, \mathbf{x}_r) \in \mathbb{R}^{s \times r} \tag{2.9}$$

przy wykorzystaniu działań na wektorach (dokładniej operacji `AXPY`) zamiast stosowanych w przypadku algorytmu 1.5 operacji na liczbach (skalarach). W tym celu zdefiniujmy pomocnicze macierze

$$Z = (\mathbf{z}_1, \ldots, \mathbf{z}_r, \mathbf{y}_1), \;\; F = (\mathbf{f}_1, \ldots, \mathbf{f}_r, \mathbf{e}_1) \in \mathbb{R}^{s \times (r+1)}. \tag{2.10}$$

Pierwszy krok algorytmu *dziel i zwyciężaj* opisanego w poprzednim paragrafie sprowadza się do wyznaczenia wszystkich wektorów $\mathbf{z}_j$ oraz wektora $\mathbf{y}_1$, a zatem macierzy Z. Zamiast wykorzystać do tego prosty algorytm 1.5 oparty na (1.28), a zatem wyznaczać poszukiwane kolumny macierzy Z rozwiązując kolejne układy równań liniowych $L\mathbf{z}_j = \mathbf{f}_j$, $j = 1, \ldots, r$, oraz układ $L\mathbf{y}_1 = \mathbf{e}_1$, będziemy wyznaczać kolejne wiersze macierzy Z spełniającej równanie

$$LZ = F, \tag{2.11}$$

stosując wzór

$$Z_{k,*} = \begin{cases} \mathbf{0} & \text{dla } k \leq 0 \\ F_{k,*} + \sum\limits_{j=1}^{m} a_j Z_{k-j,*} & \text{dla } 1 \leq k \leq s. \end{cases} \tag{2.12}$$

Istotnie (2.12) stanowi uogólnienie wzoru (1.28) dla poszczególnych wierszy macierzy Z. Zatem jeśli przyjmiemy, że początkowo $Z = F$, wówczas działania opisane wzorem (2.12), czyli wyznaczenie wiersza $Z_{k,*}$, $k = 1, \ldots, s$, będzie można zapisać w postaci ciągu operacji `AXPY`, czyli dla kolejnych wartości $j = 1, \ldots, \min\{m, k-1\}$

$$Z_{k,*} \leftarrow Z_{k,*} + a_j Z_{k-j,*}. \tag{2.13}$$

Następnym krokiem algorytmu jest wyznaczenie kolejno wektorów $\mathbf{x}_j$, $j = 2, \ldots, r$, czyli kolumn macierzy X. Na początek należy wyznaczyć współczynniki α_j^k, $k = 1, \ldots, m$, w sposób umożliwiający wektoryzację obliczeń. W tym celu zapiszmy wzór (2.6) w następującej postaci

$$\begin{cases} \alpha_j^1 = a_m x_{(j-1)s-m+1} + \ldots + a_2 x_{(j-1)s-1} + a_1 x_{(j-1)s} \\ \alpha_j^2 = a_m x_{(j-1)s-m+2} + \ldots + a_2 x_{(j-1)s} \\ \vdots \\ \alpha_j^m = a_m x_{(j-1)s} \end{cases} \tag{2.14}$$

Powyższy wzór (2.14) może być zapisany również w postaci wektorowej

$$\begin{pmatrix} \alpha_j^1 \\ \alpha_j^2 \\ \vdots \\ \alpha_j^m \end{pmatrix} = a_m \begin{pmatrix} x_{(j-1)s-m+1} \\ x_{(j-1)s-m+2} \\ \vdots \\ x_{(j-1)s} \end{pmatrix} + a_{m-1} \begin{pmatrix} x_{(j-1)s-m+2} \\ \vdots \\ x_{(j-1)s} \\ 0 \end{pmatrix}$$

$$+ \ldots + a_2 \begin{pmatrix} x_{(j-1)s-1} \\ x_{(j-1)s} \\ 0 \\ \vdots \\ 0 \end{pmatrix} + a_1 \begin{pmatrix} x_{(j-1)s} \\ 0 \\ \vdots \\ \vdots \\ 0 \end{pmatrix} \tag{2.15}$$

Wykorzystamy teraz (2.15) do określenia ciągu operacji, które posłużą do obliczenia wektora współczynników α_j^k. Dla $k = 1, \ldots, m$ wykonujemy operację `AXPY` dla wektorów o długości $m - k + 1$, a mianowicie

$$\begin{pmatrix} \alpha_j^1 \\ \alpha_j^2 \\ \vdots \\ \alpha_j^{m-k+1} \end{pmatrix} \leftarrow \begin{pmatrix} \alpha_j^1 \\ \alpha_j^2 \\ \vdots \\ \alpha_j^{m-k+1} \end{pmatrix} + a_{m+1-k} \begin{pmatrix} x_{(j-1)s-m+k} \\ x_{(j-1)s-m+k+1} \\ \vdots \\ x_{(j-1)s} \end{pmatrix} \tag{2.16}$$

Oczywiście początkowo musi być $\alpha_j^k = 0$ dla $k = 1, \ldots, m$. Każda operacja o postaci (2.16) aktualizuje $m - k + 1$ pierwszych składowych wektora współczynników. Następnie, wykorzystując wzór (2.5), możemy zastosować operację `AXPY` dla wyznaczenia wektora $\mathbf{x}_j$, czyli dla $k = 1, \ldots, m$ wykonujemy

$$\mathbf{x}_j \leftarrow \mathbf{x}_j + \alpha_j^k \mathbf{y}_k. \tag{2.17}$$

Przedstawiona metoda wymaga dodatkowego miejsca w pamięci dla reprezentacji wektora $\mathbf{y}_1$ oraz m współczynników α_j^k. Oznacza to konieczność alokacji dodatkowej pamięci dla $s + m$ liczb zmiennopozycyjnych. Jako podsumowanie przedstawionych wyżej rozważań otrzymujemy algorytm 2.1. Schemat wyznaczania poszczególnych elementów rozwiązania przedstawia rysunek 2.1.

Istnieje możliwość przyspieszenia algorytmu 2.1 dzięki użyciu podprogramu z biblioteki BLAS poziomu drugiego, kosztem użycia dodatkowego miejsca w pamięci potrzebnego do przechowywania $ms + m$ liczb rzeczywistych. W tym celu, wykorzystując (2.7), zdefiniujmy

Algorytm 2.1. Wektorowe wyznaczanie rozwiązania liniowego równania rekurencyjnego o stałych współczynnikach (1.28) przy ustalonych wartościach całkowitych $r > 1$, $s > 1$ takich, że $rs = n$.

Wejście: liczby $a_1, \ldots, a_m$ oraz $f_1, \ldots, f_n$ tworzące wektory $\mathbf{f}_1, \ldots, \mathbf{f}_r$ zdefiniowane przez (2.8).

Wyjście: $X_{1:s,1:r} = (\mathbf{x}_1, \ldots \mathbf{x}_r)$.

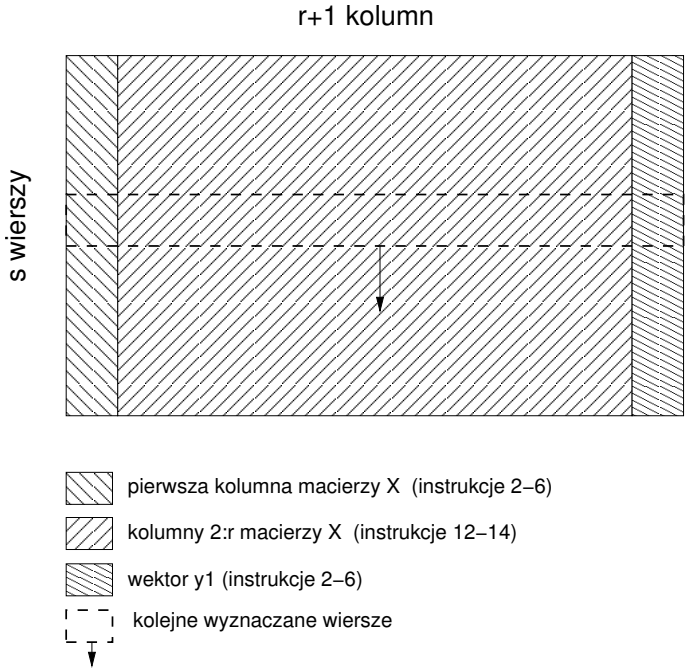
1: $X \leftarrow (\mathbf{f}_1, \ldots \mathbf{f}_r, \mathbf{e}_1)$
2: **for** $k = 2$ to s **do**
3: **for** $j = 1$ to $\min\{m, k-1\}$ **do**
4: $X_{k,*} \leftarrow X_{k,*} + a_j X_{k-j,*}$ { `AXPY`}
5: **end for**
6: **end for**{ $X_{*,r+1} = \mathbf{y}_1$}
7: **for** $j = 2$ to r **do**
8: $\left(\alpha_j^1, \ldots, \alpha_j^m\right)^T \leftarrow (0, \ldots, 0)^T$
9: **for** $k = 1$ to m **do**
10: $\begin{pmatrix} \alpha_j^1 \\ \vdots \\ \alpha_j^{m-k+1} \end{pmatrix} \leftarrow \begin{pmatrix} \alpha_j^1 \\ \vdots \\ \alpha_j^{m-k+1} \end{pmatrix} + a_{m+1-k} X_{s-m+k:s,j-1}$ { `AXPY`}
11: **end for**
12: **for** $k = 1$ to m **do**
13: $X_{*,j} \leftarrow X_{*,j} + \alpha_j^k \mathbf{y}_k$ {operacja `AXPY`, stosując (2.7)}
14: **end for**
15: **end for**

macierz

$$Y = (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_m) = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ y_2 & 1 & \ddots & \vdots \\ y_3 & y_2 & \ddots & 0 \\ \vdots & y_3 & \ddots & 1 \\ \vdots & \vdots & \ddots & y_2 \\ \vdots & \vdots & & \vdots \\ y_s & y_{s-1} & \ldots & y_{s-m+1} \end{pmatrix}. \tag{2.18}$$

Wówczas zamiast ciągu (2.17) operacji `AXPY` realizującego

$$\mathbf{x}_j \leftarrow \mathbf{x}_j + \alpha_j^1 \mathbf{y}_1 + \ldots + \alpha_j^m \mathbf{y}_m, \tag{2.19}$$



Rys. 2.1. Wyznaczanie elementów $X_{1:s,1:r}$ w algorytmie 2.1

Fig. 2.1. Finding the elements of $X_{1:s,1:r}$ using Algorithm 2.1

można wyznaczyć wektor $\mathbf{x}_j$ za pomocą jednego mnożenia macierzy przez wektor, czyli operacji GEMV z biblioteki BLAS poziomu 2. Istotnie (2.19) jest równoważne operacji

$$\mathbf{x}_j \leftarrow \mathbf{x}_j + Y \begin{pmatrix} \alpha_j^1 \\ \vdots \\ \alpha_j^m \end{pmatrix}. \tag{2.20}$$

Oczywiście macierz Y musi być wcześniej zbudowana, a zatem po wyznaczeniu rozwiązania układu równań $LZ = F$ należy wykorzystując operację COPY z biblioteki BLAS poziomu pierwszego, skopiować $m - 1$ razy (z odpowiednim przesunięciem) ostatnią kolumnę macierzy Z oraz wyzerować odpowiednie składowe tablicy, by w pamięci przechowywać macierz

$$(Z, Y) \in \mathbb{R}^{s\times(r+m)}.$$

Wówczas po obliczeniu współczynników α_j^k wyznaczenie każdego wektora $\mathbf{x}_j$, $j = 2, \ldots, r$, czyli instrukcje 12–14, będzie mogło być realizowane operacją GEMV z biblioteki BLAS poziomu drugiego. Zauważmy, że ta modyfikacja nie zmienia liczby działań arytmetycznych w algorytmie 2.1. Otrzymamy w ten sposób algorytm 2.2.

Algorytm 2.2. Wektorowo-macierzowe wyznaczanie rozwiązania liniowego równania rekurencyjnego o stałych współczynnikach (1.28) przy ustalonych wartościach całkowitych $r > 1$, $s > 1$ takich, że $rs = n$.

Wejście: liczby $a_1, \ldots, a_m$ oraz $f_1, \ldots, f_n$ tworzące wektory $\mathbf{f}_1, \ldots, \mathbf{f}_r$ zdefiniowane przez (2.8).

Wyjście: $X_{1:s,1:r} = (\mathbf{x}_1, \ldots \mathbf{x}_r)$.

1: $X \leftarrow (\mathbf{f}_1, \ldots \mathbf{f}_r, \mathbf{e}_1)$

2: **for** $k = 2$ to s **do**

3: **for** $j = 1$ to $\min\{m, k-1\}$ **do**

4: $X_{k,*} \leftarrow X_{k,*} + a_j X_{k-j,*}$ {operacja `AXPY`}

5: **end for**

6: **end for**$\{ X_{*,r+1} = \mathbf{y}_1 \}$

7: $Y \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_m)$ {stosując (2.18)}

8: **for** $j = 2$ to r **do**

9: $\left(\alpha_j^1, \ldots, \alpha_j^m\right)^T \leftarrow (0, \ldots, 0)^T$

10: **for** $k = 1$ to m **do**

11: $\begin{pmatrix} \alpha_j^1 \\ \vdots \\ \alpha_j^{m-k+1} \end{pmatrix} \leftarrow \begin{pmatrix} \alpha_j^1 \\ \vdots \\ \alpha_j^{m-k+1} \end{pmatrix} + a_{m+1-k} X_{s-m+k:s,j-1}$ { operacja `AXPY`}

12: **end for**

13: $X_{*,j} \leftarrow X_{*,j} + Y \begin{pmatrix} \alpha_j^1 \\ \vdots \\ \alpha_j^m \end{pmatrix}$ {operacja `GEMV`}

14: **end for**

2.3. Analiza algorytmu i optymalny wybór parametrów

W przedstawionych algorytmach istotny problem stanowi właściwe wyznaczenie wartości parametrów r i s. Oczywiście musi być $rs \leq n$. Gdy $rs < n$, wówczas stosujemy algorytm 2.1 lub 2.2 dla wyznaczenia wartości $x_1, \ldots, x_{rs}$, a następnie algorytm 1.5, aby wyznaczyć $x_{rs+1}, \ldots, x_n$. Wartości r i s wybieramy tak, by algorytm działał jak najkrócej. W tym celu wyznaczymy teraz czas działania algorytmu, opierając się na omówionym w podrozdziale 1.6.2 modelu Hockneya - Jesshope'a obliczeń wektorowych [54, 32].

W algorytmie 2.1 wszystkie obliczenia na liczbach zmiennopozycyjnych realizowane są w postaci operacji `AXPY`, a w przypadku algorytmu 2.2 możemy przyjąć, że operacja `GEMV` może być zrealizowana jako sekwencja operacji `AXPY`. Stąd otrzymujemy następujące twierdzenie.

Twierdzenie 2.1. *Niech $T_{AXPY}(k)$ będzie czasem wykonania operacji* `AXPY` *dla wektorów o liczbie składowych równej k. Czas wykonania algorytmów 2.1 i 2.2 wyraża się wzorem*

$$\begin{aligned} T(n,m,r,s) &= m(s-\frac{m+1}{2})T_{AXPY}(r+1) \\ &+ (r-1)\left(\sum_{k=1}^{m} T_{AXPY}(m+1-k) + mT_{AXPY}(s)\right). \end{aligned} \tag{2.21}$$

Dowód. Przeanalizujmy liczbę operacji `AXPY` wywoływanych w algorytmie 2.1 oraz długości wektorów (czyli również pętli), na których one działają. We wzorze (2.13) wykonujemy operację `AXPY` dla wektorów o długości $r+1$, przy czym z uwagi na to, że nie ma potrzeby działać na wektorach składających się z samych zer, dla $k = 1,\ldots,m$ liczba wywołań wynosi $k-1$, a dla $k = m+1,\ldots,s$ wynosi m. Stąd całkowita liczba wywołań `AXPY` w pierwszym kroku wyraża się jako

$$\begin{aligned} \sum_{k=1}^{m-1} k + (s-m)m &= (s-m)m + \frac{m-1}{2}m \\ &= m\left(s-\frac{m+1}{2}\right). \end{aligned}$$

Stąd czas działania tej części algorytmu wynosi

$$m(s-\frac{m+1}{2})T_{AXPY}(r+1). \tag{2.22}$$

Następnie w instrukcjach 7–15 (albo 8–14 w algorytmie 2.2) obliczane są poszczególne wektory $\mathbf{x}_j$, przy czym koszt wyznaczenia każdego wektora jest identyczny i składa się na niego wyznaczenie współczynników α_j^k oraz zastosowanie wzoru (2.17) lub (2.20). Zatem, uwzględniając długości wektorów i liczbę wywołań operacji `AXPY`, otrzymujemy czas działania potrzebny dla wyznaczenia jednego wektora $\mathbf{x}_j$

$$\sum_{k=1}^{m} T_{AXPY}(m+1-k) + mT_{AXPY}(s). \tag{2.23}$$

Sumując wielkości (2.22) oraz (2.23) przemnożone przez $r-1$, czyli liczbę wektorów stanowiących kolumny macierzy X zdefiniowanej wzorem (2.9), otrzymujemy tezę twierdzenia. ■

Wykorzystując powyższe twierdzenie oraz wzór (1.16), możemy wyznaczyć czas działania algorytmu 2.1 przy użyciu parametrów r_∞ oraz $n_{1/2}$. Istotnie, wstawiając (1.16) do (2.21) otrzymujemy

$$\begin{aligned} T(n,m,r,s) &= \frac{2\cdot 10^{-6}}{r_\infty}\left(m(s-\frac{m+1}{2})(n_{1/2}+r+1)\right. \\ &+ \left.(r-1)\left(\sum_{k=1}^{m}(n_{1/2}+m+1-k) + m(n_{1/2}+s)\right)\right). \end{aligned}$$

Stąd po dokonaniu stosownych przekształceń otrzymujemy

$$T(n,m,r,s) = \frac{2 \cdot 10^{-6}}{r_\infty} m(2rs + 2n_{1/2}r + n_{1/2}s - 2.5n_{1/2} - 0.5n_{1/2}m - m - 1). \tag{2.24}$$

Kolejnym elementem analizy algorytmów 2.1 i 2.2 będzie wyznaczenie optymalnych wartości parametrów r i s. Minimalizując funkcję $T(n,m,r,s)$ przy warunku $rs = n$ oraz ustalonych rozmiarów problemu n, m, otrzymujemy wniosek, że minimalna wartość czasu działania algorytmów jest osiągnięta dla

$$(r,s) = \left(\sqrt{n/2}, \sqrt{2n}\right).$$

Oczywiście, wartości parametrów r i s muszą być całkowite, zatem jako optymalne wartości parametrów r i s należy przyjąć

$$(r,s) = \left(\left\lfloor\sqrt{n/2}\right\rfloor, \left\lfloor\sqrt{2n}\right\rfloor\right). \tag{2.25}$$

Zauważmy, że wartość parametru s określona przez (2.25) nie zależy od m i co najważniejsze, nie zależy również od r_∞ oraz $n_{1/2}$. Oznacza to, że w kodzie źródłowym algorytmu można zaprogramować obliczanie r i s ze wzoru (2.25) jako optymalny wybór tych parametrów. Pozostanie on słuszny niezależnie od konkretnych wartości r_∞, $n_{1/2}$ dla danego komputera.

Wybór optymalnych wartości parametrów algorytmu określony wzorem (2.25) powinien być „dostrojony”, aby uzyskać jak najlepszą szybkość algorytmu poprzez eliminację konfliktów w dostępie do banków pamięci. Instrukcje 2–6 algorytmów 2.1 i 2.2 zapisane w języku Fortran z wykorzystaniem podprogramu `SAXPY`, czyli `AXPY` dla pojedynczej precyzji, przyjmą następującą postać

```
do k=1,s
  do j=1,min(m,k-1)
    call saxpy(r+1,a(j),x(k-j),s,x(k),s)
  end do
end do
```

Parametry wywołania podprogramu `AXPY` opisują kolejno: długość wektorów, skalar, wektor mnożony, odległość między jego kolejnymi składowymi, wektor wynikowy oraz odległość między kolejnymi składowymi (ang. *stride*). W obu przypadkach odległość jest równa s, gdyż wektory te stanowią wiersze dwuwymiarowej tablicy `x` odpowiadającej macierzy X, gdzie $LDA = s$. Zatem jeśli wartość s będzie wielokrotnością potęgi liczby 2, może dojść do opisanego w podrozdziale 1.1 niekorzystnego zjawiska konfliktów w dostępie do banków pamięci. Stąd należy tak dobrać parametry metody, aby wartość s była nieparzysta, czyli

$$s = \begin{cases} \left\lfloor\sqrt{2n}\right\rfloor - 1 & \text{dla } \left\lfloor\sqrt{2n}\right\rfloor \text{ parzystych,} \\ \left\lfloor\sqrt{2n}\right\rfloor & \text{w pozostałych przypadkach,} \end{cases} \tag{2.26}$$

a następnie

$$r = \lfloor n/s \rfloor. \tag{2.27}$$

Na zakończenie zbadajmy liczbę operacji zmiennopozycyjnych wykonywanych w ramach algorytmów wektorowych 2.1, 2.2 oraz algorytmu skalarnego 1.5.

Twierdzenie 2.2. *Liczba operacji zmiennopozycyjnych w algorytmach 2.1 i 2.2 wynosi*

$$4mrs - 2m(m+1). \tag{2.28}$$

Dowód. Zamieniając we wzorze (2.21) czas działania odpowiednich operacji `AXPY` na liczbę operacji zmiennopozycyjnych, otrzymujemy wzór na łączną liczbę operacji w rozważanych algorytmach, czyli

$$\begin{aligned} &2(r+1)m(s - \frac{m+1}{2}) + 2(r-1)\left(\sum_{k=1}^{m}(m+1-k) + ms\right) \\ &= 4rsm - rm(m+1) + 2(r-1)\left(m(m+1) - m\frac{(m+1)}{2}\right) \end{aligned}$$

Stąd po prostych przekształceniach otrzymujemy (2.28). ■

Twierdzenie 2.3 ([104]). *Liczba operacji zmiennopozycyjnych w algorytmie 1.5 wynosi*

$$2mn - m(m+1). \tag{2.29}$$

Oczywiście, wyznaczenie rozwiązania równania (1.28) algorytmem 2.1 lub 2.2 dla dowolnego n przy ustalonych wartościach r i s wymaga na ogół zastosowania algorytmu 1.5 dla wyznaczenia liczb $x_{rs+1}, \ldots, x_n$. Stąd otrzymujemy następujące wnioski.

Wniosek 2.1. *Wyznaczenie rozwiązania równania (1.28) za pomocą algorytmu 2.1 lub 2.2 wymaga wykonania*

$$2mn + 2mrs - 2m(m+1) \tag{2.30}$$

operacji zmiennopozycyjnych.

Wniosek 2.2. *Algorytmy 2.1 oraz 2.2 są zgodne z algorytmem 1.5 w sensie definicji 1.4.*

Zauważmy, że jeśli przyjmiemy optymalne wartości parametrów r i s, to zastosowanie algorytmów wektorowych 2.1 i 2.2 będzie się charakteryzować dwukrotnie większą liczbą operacji

arytmetycznych niż w przypadku prostego algorytmu 1.5. W przypadku algorytmu *cyclic reduction* łączna liczba operacji w algorytmie wynosi $2m^2n$ [20], a zatem jest m-krotnie większa niż optymalna określona wzorem (2.29) dla algorytmu 1.5. W przypadku algorytmu *recursive doubling* nadmiarowość jest jeszcze większa i wynosi $\log_2 n$ [5, 100].

Oczywiście z uwagi na to, że w algorytmach w pełni wektoryzowalnych (jak przedstawione wyżej algorytmy 2.1 i 2.2) wszystkie operacje mogą być wykonywane w trybie wektorowym, należy oczekiwać, że takie właśnie algorytmy będą działały znacznie szybciej na komputerach oferujących mechanizmy wektorowości i to zrównoważy konieczność wykonania większej niż określona wzorem (2.29) optymalnej liczby operacji. Poszczególne fragmenty algorytmu będą wykonywane na ogół z różnymi prędkościami (jak w prawie Amdahla), a zatem należy oczekiwać, że liczba wykonywanych operacji nie musi dokładnie odzwierciedlać czasu działania algorytmu.

2.4. Wyniki eksperymentów – porównanie algorytmów

Wprowadzone w poprzednim punkcie algorytmy porównamy ze skalarnym algorytmem 1.5 oraz dwoma algorytmami 2.3 i 2.4, wykorzystującymi podprogramy bibliotek BLAS i LAPACK.

W algorytmie 2.4 wykorzystano pojedyncze wywołanie podprogramu `TBSV` z BLAS-u poziomu 2, rozwiązującego równanie (1.27). W algorytmie 2.3 zastosowano podprogram `TBTRS` z biblioteki LAPACK rozwiązujący (1.27) dla wielu prawych stron, który wykorzystano do rozwiązania układu (2.11), dalsze zaś kroki zrealizowano jak w algorytmie 2.1. Liczba operacji zmiennopozycyjnych w operacji `TBSV` jest określona przez (2.29) [32]. Po analizie kodu źródłowego podprogramu `TBTRS` z biblioteki BLAS[1] przyjmujemy, że liczba operacji zmiennopozycyjnych w algorytmie 2.3 jest identyczna jak w algorytmie 2.1, a liczba operacji w algorytmie 2.4 odpowiada wielkości (2.29), choć nie można mieć pewności co do algorytmu zastosowanego w bibliotece scilib dostarczanej wraz z komputerami Cray, gdyż nie są udostępniane jej kody źródłowe.

Wszystkie algorytmy zostały zaimplementowane w języku Fortran oraz przetestowane na dwóch komputerach wektorowych Cray C90 oraz Cray SV1, które charakteryzują się różnymi wartościami r_∞, $n_{1/2}$ dla występujących w rozważanych algorytmach pętli. Użyto kompilatora optymalizującego (z opcją automatycznej wektoryzacji) oraz zoptymalizowanych podprogramów BLAS-u i LAPACK-a z biblioteki Cray scilib. Każdy algorytm testowano dla wielu róż-

[1] `http://www.netlib.org/blas/stbtrs.f`

Algorytm 2.3. Wektorowe wyznaczanie rozwiązania liniowego równania rekurencyjnego o stałych współczynnikach (1.28) przy ustalonych wartościach całkowitych $r > 1$, $s > 1$ takich, że $rs = n$.

Wejście: liczby $a_1, \ldots, a_m$ oraz $f_1, \ldots, f_n$ tworzące wektory $\mathbf{f}_1, \ldots, \mathbf{f}_r$ zdefiniowane przez (2.8).

Wyjście: $X_{1:s,1:r} = (\mathbf{x}_1, \ldots \mathbf{x}_r)$.

1: zastosuj podprogram `TBTRS` do wyznaczenia $X = L^{-1}F$, przyjmując $a_{kj} \equiv a_j$ dla $k = 1, \ldots, n$
2: **for** $j = 2$ to r **do**
3: $\left(\alpha_j^1, \ldots, \alpha_j^m\right)^T \leftarrow (0, \ldots, 0)^T$
4: **for** $k = 1$ to m **do**
5: $\begin{pmatrix} \alpha_j^1 \\ \vdots \\ \alpha_j^{m-k+1} \end{pmatrix} \leftarrow \begin{pmatrix} \alpha_j^1 \\ \vdots \\ \alpha_j^{m-k+1} \end{pmatrix} + a_{m+1-k} X_{s-m+k:s,j-1}$ { `AXPY`}
6: **end for**
7: **for** $k = 1$ to m **do**
8: $X_{*,j} \leftarrow X_{*,j} + \alpha_j^k \mathbf{y}_k$ { `AXPY`, stosując (2.7)}
9: **end for**
10: **end for**

Algorytm 2.4. Wyznaczania rozwiązania liniowego równania rekurencyjnego o stałych współczynnikach (1.28) przy użyciu podprogramu `TBSV`.

Wejście: liczby $f_1, \ldots, f_n$ oraz $a_1, \ldots, a_m$, przy czym początkowo $x_k = f_k$ dla $k = 1, \ldots, n$

Wyjście: obliczone $x_1, \ldots, x_n$

1: zastosuj podprogram `TBSV`, przyjmując $a_{kj} \equiv a_j$ dla $k = 1, \ldots, n$

nych rozmiarów (wartości n i m) oraz parametrów r i s, mierząc rzeczywisty czas działania przy użyciu funkcji systemowej `timef()`.

Rysunki 2.2 i 2.3 ilustrują zależność między różnymi wartościami parametru s a mierzonym w sekundach czasem działania algorytmu 2.1 dla różnych wartości m oraz n. Jak zaznaczyliśmy w podrozdziale 2.3, z teoretycznego punktu widzenia, optymalny wybór parametrów metody nie zależy od charakterystyki konkretnego procesora. Wyniki eksperymentów potwierdzają tę tezę. Dla obu komputerów metoda jest wykonywana najszybciej dla wartości s bliskich optymalnej wartości określonej wzorem (2.25), która nie zależy od wartości m. Jednocześnie można zaobserwować bardzo charakterystyczne wzrosty czasu działania algorytmu dla pewnych, regularnie powtarzających się wartości s. Są to właśnie wielokrotności potęgi liczby 2. Są one tym większe, im większy jest wykładnik potęgi. Potwierdza to rozważania teoretyczne z pod-

rozdziału 2.3. Zatem, wybór parametrów s i r określony odpowiednio wzorami (2.26) i (2.27) należy traktować jako bardzo dobre praktycznie określenie wartości, które należy stosować.

Rysunek 2.4 pokazuje zależność między rozmiarem problemu (wartość m) a czasem działania rozważanych algorytmów: 1.5, 2.1, 2.2, 2.3, 2.4 dla wybranych wartości n. W przypadku algorytmów 2.1, 2.2, 2.3 zastosowano optymalny dobór parametrów metody zdefiniowany wzorami (2.26) i (2.27). Na początek zaobserwujmy, że zachowanie się algorytmów jest podobne dla małych ($n = 64000$) i dużych ($n = 1024000$) rozmiarów problemu oraz obu komputerów. Dla $m = 1$ największą szybkość działania osiąga algorytm 2.1, a dla $m > 1$ większą szybkość osiąga algorytm 2.2 dzięki zastosowaniu podprogramu mnożenia macierz - wektor z drugiego poziomu BLAS-u. Rysunek 2.5 pokazuje wydajność obu komputerów wykonujących rozważane algorytmy. Wydajność rośnie wraz ze wzrostem wartości m oraz n. Dla algorytmów 2.3 oraz 2.4 komputery osiągają wydajność większą niż dla prostego algorytmu 1.5, ale znacznie mniejszą niż dla algorytmów 2.1 i 2.2. Odnotujmy również fakt, że wydajność obu komputerów dla wszystkich algorytmów jest podobna dla mniejszych wartości n, ale prawie dwukrotnie większa na nowszym komputerze Cray SV1 przy większych wartościach n.

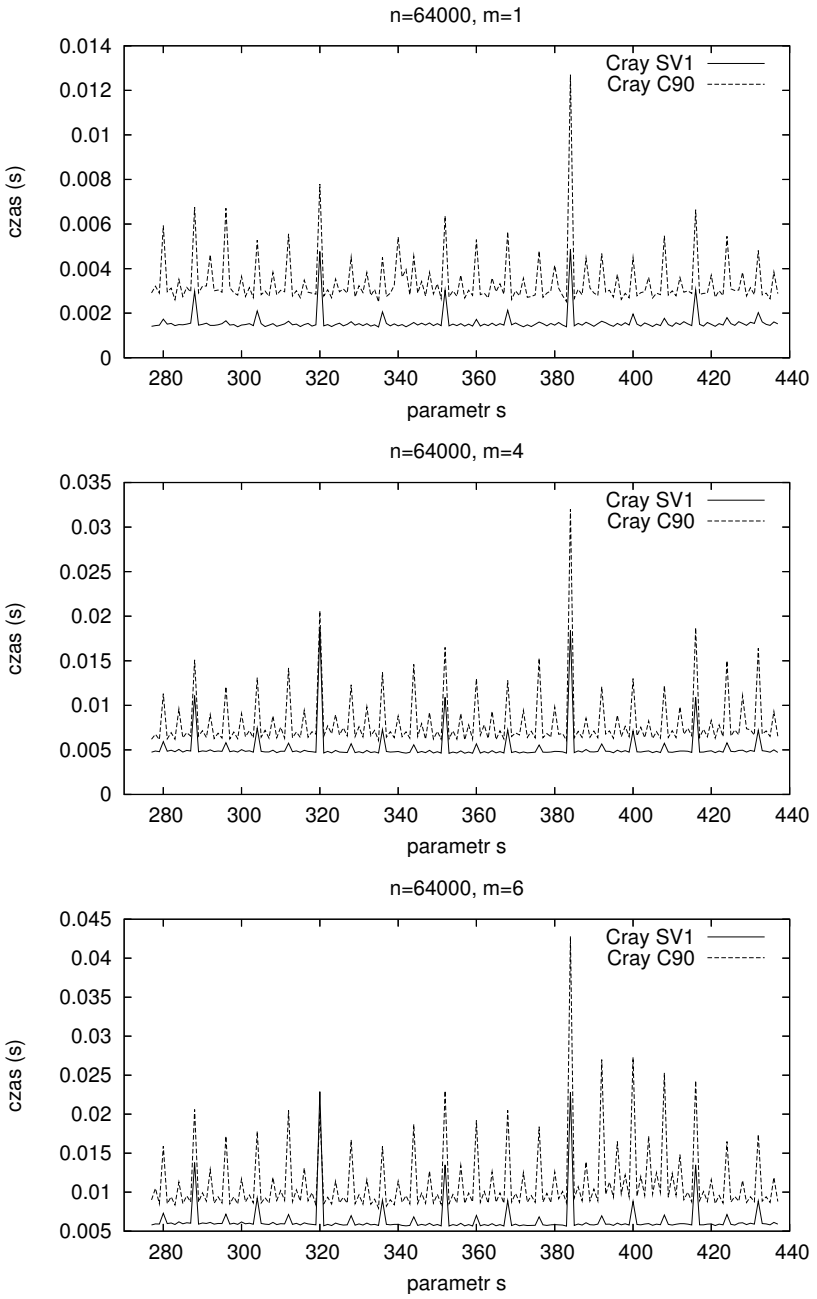
Z punktu widzenia użytkownika najistotniejsze jest przyspieszenie algorytmów zoptymalizowanych 2.1, 2.2, 2.3, 2.4 względem najprostszego i jednocześnie charakteryzującego się najmniejszą liczbą operacji zmiennopozycyjnych algorytmu 1.5 (w sensie definicji 1.5). Rysunek 2.6 pokazuje zależność między rozmiarem problemu n a przyspieszeniem względem algorytmu 1.5 dla wybranych wartości m. Przyspieszenie algorytmów wektorowych 2.1 i 2.2 na ogół rośnie wraz ze wzrostem wartości n. Jednak wzrost rozmiaru problemu z 1024000 do 2048000 powoduje niewielki wzrost przyspieszenia na komputerze Cray SV1, a nawet w przypadku komputera Cray C90 oraz $m = 1$ jego spadek. Jednocześnie przyspieszenie maleje wraz ze wzrostem wartości m, co jest spowodowane wzrostem wydajności wykonania algorytmu 1.5, gdyż dla większych wartości m pętla wewnętrzna w algorytmie 1.5 może być wektoryzowana. Algorytmy 2.3 i 2.4 osiągają niewielkie przyspieszenie niezależne od wartości n i są do sześciu razy wolniejsze niż algorytmy wektorowe. Jako podsumowanie powyższych rozważań odnotujmy następujące wnioski.

- Algorytmy wektorowe 2.1 i 2.2 osiągają największą szybkość działania dla wartości s i r zbliżonych do optymalnych, zdefiniowanych przez (2.26) i (2.27). Wybór parametrów zależy jedynie od n.

- Użycie algorytmu 2.2 jest wskazane dla $m > 1$, ale wymaga dodatkowego miejsca dla reprezentacji macierzy Y (sm słów maszynowych).

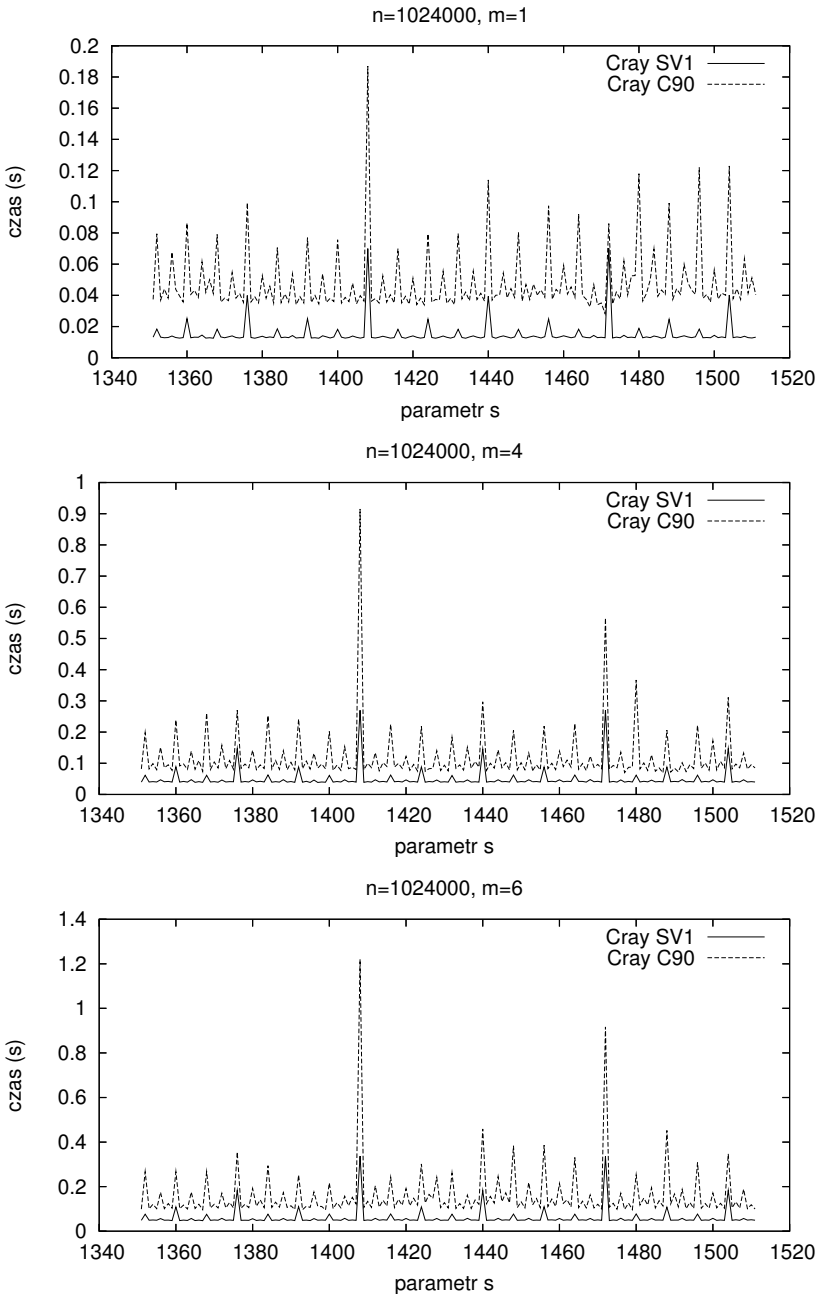
- Przyspieszenie względem algorytmu 1.5 oraz wydajność komputerów dla algorytmów wektorowych na ogół rośnie wraz ze wzrostem rozmiaru problemu (wartości n i m).
- Dla $s = a2^k$ szybkość działania algorytmów wektorowych drastycznie spada, co jest efektem konfliktów w dostępie do banków pamięci. Im większa wartość k, tym większy spadek szybkości.
- Szybkość działania algorytmów 2.3 i 2.4 wykorzystujących algorytmy z biblioteki Cray scilib jest bardzo niewielka. Nie wykorzystują one specjalnej struktury macierzy L.
- Na komputerze Cray SV1, dla $m = 1$ i odpowiednio dużych wartości n, algorytm 2.1 osiąga przyspieszenie około 60 względem algorytmu 1.5.

Na zakończenie dodajmy, że zrównoleglenie algorytmów wektorowych 2.1 i 2.2 oparte na zastosowaniu równoległej wersji operacji `AXPY` nie jest efektywne. W następnym rozdziale przedstawimy inną metodę, która będzie mogła być efektywnie zrównoleglona.



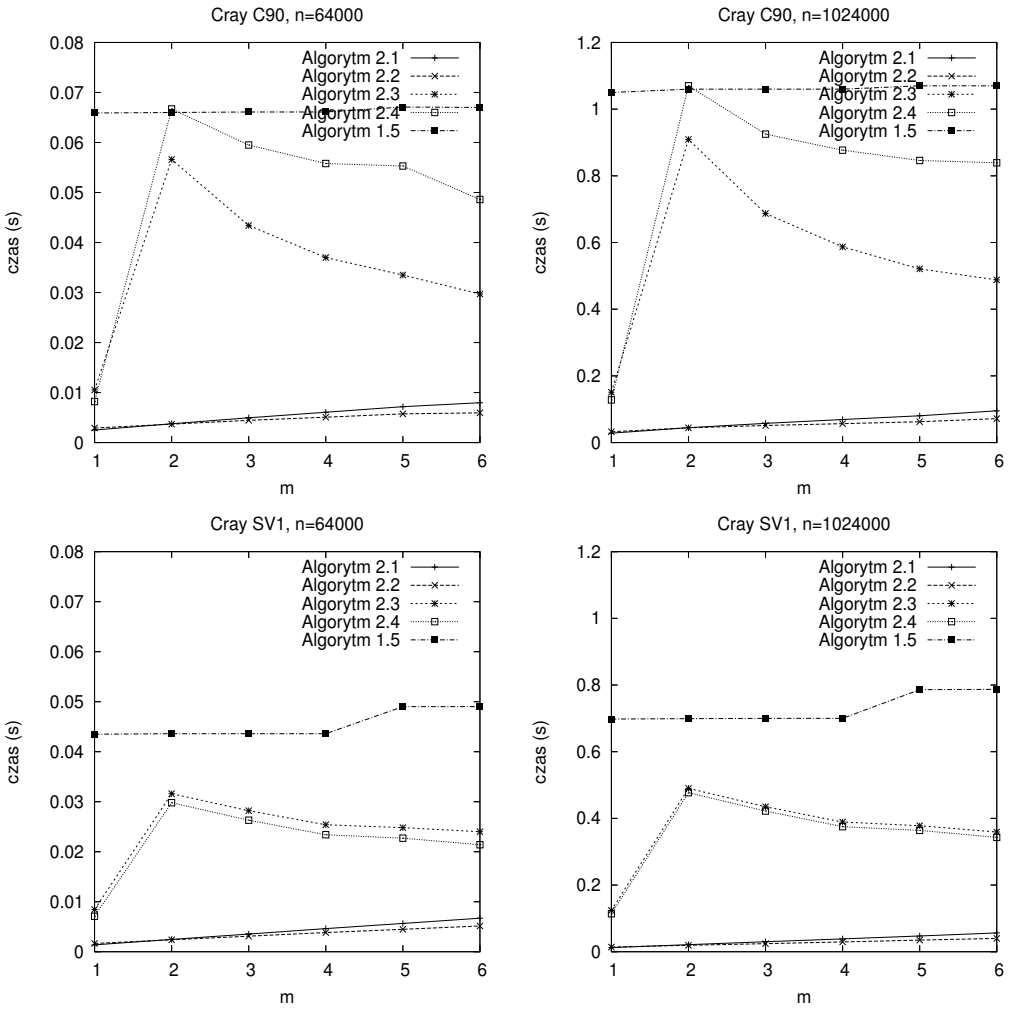
Rys. 2.2. Czas wykonania algorytmu 2.1 dla $n = 64000$ i różnych wartości s. Optymalna wartość parametru s wynosi 357

Fig. 2.2. Execution time of Algorithm 2.1 for $n = 64000$ and various s. The optimal value of s is 357



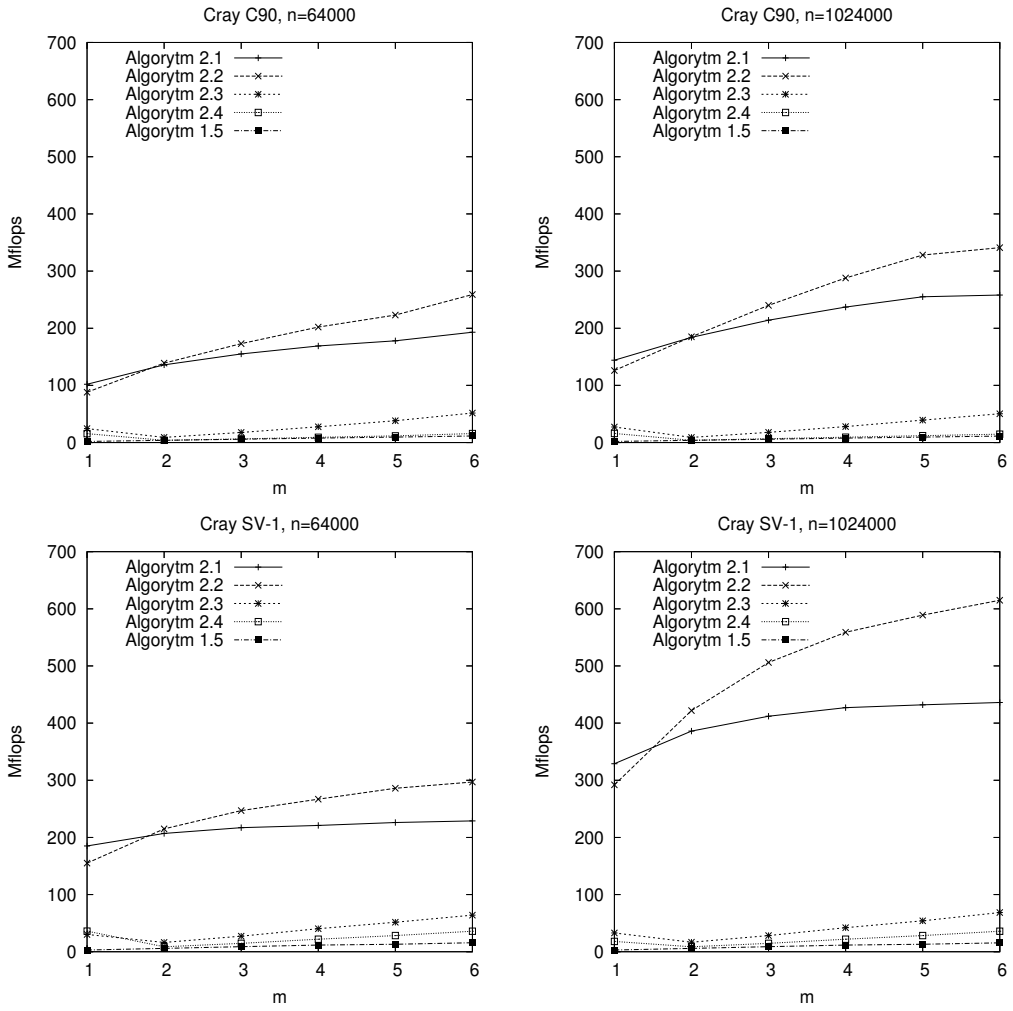
Rys. 2.3. Czas wykonania algorytmu 2.1 dla $n = 1024000$ i różnych wartości s. Optymalna wartość parametru s wynosi 1431

Fig. 2.3. Execution time of Algorithm 2.1 for $n = 1024000$ and various s. The optimal value of s is 1431



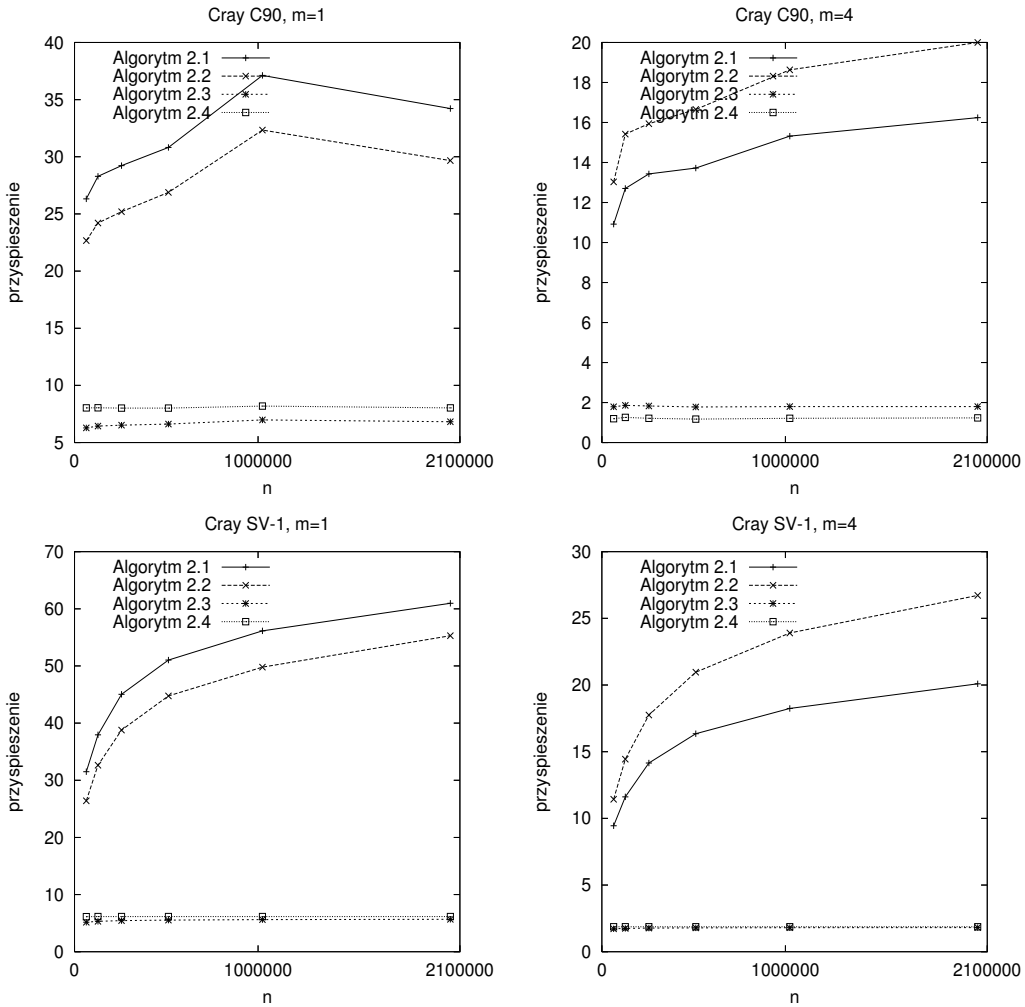
Rys. 2.4. Czas wykonania rozważanych algorytmów na komputerach Cray C90 i SV1

Fig. 2.4. Execution time of the considered algorithms on a Cray C90 and SV1



Rys. 2.5. Wydajność komputerów Cray C90 i SV1 wykonujących rozważane algorytmy

Fig. 2.5. Pereformance of the considered algorithms on a Cray C90 and SV1



Rys. 2.6. Przyspieszenie rozważanych algorytmów względem algorytmu 1.5 w sensie definicji 1.5
Fig. 2.6. Speedup of the algorithms relative to Algorithm 1.5 defined by Definition 1.5

3. Blokowe algorytmy liniowych obliczeń rekurencyjnych

Efektywność algorytmu 2.1 przedstawionego w poprzednim rozdziale zależy w dużej mierze od efektywności wykonania operacji `AXPY`. Co więcej, wraz ze wzrostem rzędu równania (wartość m) wydajność komputerów wektorowych wykonujących algorytm 2.1 rośnie nieznacznie. W takim przypadku istotne ulepszenie algorytmu można osiągnąć jedynie przez zastosowanie podprogramów z wyższych poziomów biblioteki BLAS, co może przyczynić się też do wzrostu efektywności obliczeń rekurencyjnych na komputerach z procesorami innymi niż wektorowe, gdzie szybkość wykonania obliczeń w ramach operacji `AXPY` na ogół znacznie odbiega od teoretycznej maksymalnej wydajności komputera. Dodatkową zaletą będzie możliwość efektywnego zrównoleglenia takiego algorytmu.

3.1. BLAS poziomów 2 i 3 w rozwiązywaniu liniowych równań rekurencyjnych

Za punkt wyjścia dla wyznaczenia rozwiązania równania (1.28) przyjmijmy założenia dotyczące algorytmu 2.1. Dodatkowo używając liczb $a_1, \ldots, a_m$ wprowadzonych w (1.28), zdefiniujmy następującą macierz górnotrójkątną

$$C = \begin{pmatrix} a_m & a_{m-1} & \cdots & a_1 \\ 0 & a_m & \cdots & a_2 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & a_m \end{pmatrix} \in \mathbb{R}^{m\times m}. \tag{3.1}$$

Podobnie jak w przypadku algorytmu 2.1 rozważmy na początek rozwiązanie układu równań $LZ = F$, gdzie macierz $L \in \mathbb{R}^{s\times s}$ jest postaci (2.3), a macierze Z i F zdefiniowano wzorem (2.10). Każdy wiersz $Z_{k,*}$ macierzy Z może być wyznaczany za pomocą następującego wzoru stanowiącego uogólnienie (2.12)

$$Z_{k,*} \leftarrow Z_{k,*} + C_{1,\max\{1,m-k+2\}:m} Z_{\max\{1,k-m\}:k-1,*}, \tag{3.2}$$

gdzie $C_{1,\max\{1,m-k+2\}:m}$ oznacza pewien, zależny od wartości k, fragment pierwszego wiersza macierzy C dla $k = 2, \ldots, m$ oraz cały pierwszy wiersz dla $k = m+1, \ldots, s$, czyli inaczej liczby

$$a_{\min\{m,k-1\}}, a_{\min\{m,k-1\}-1}, \ldots, a_1.$$

Istotnie, zapisując (3.2) w postaci rozwiniętej, otrzymamy wzór (2.12). Operacja określona wzorem (3.2) może być zaprogramowana za pomocą operacji mnożenia macierzy przez wektor, czyli operacji GEMV z BLAS-u poziomu drugiego. Transpozycja iloczynu

$$C_{1,\max\{1,m-k+2\}:m} Z_{\max\{1,k-m\}:k-1,*}$$

może być zapisana w postaci

$$(C_{1,\max\{1,m-k+2\}:m} Z_{\max\{1,k-m\}:k-1,*})^T = Z^T_{\max\{1,k-m\}:k-1,*} C^T_{1,\max\{1,m-k+2\}:m},$$

czyli właśnie jako odpowiedni wariant operacji GEMV. Ostatecznie podsumujmy, że wyznaczenie rozwiązania układu $LZ = F$ może być zrealizowane jako sekwencja operacji mnożenia macierzy przez wektor.

W następnym kroku wyznaczamy m ostatnich składowych wektorów $\mathbf{x}_2, \ldots, \mathbf{x}_r$, czyli posługując się terminologią macierzową, elementy macierzy $X_{s-m+1:s,2:r}$. W tym celu zdefiniujmy macierz, której elementami są współczynniki α_j^k zdefiniowane wzorem (2.6) jako

$$A = \begin{pmatrix} \alpha_2^1 & \cdots & \alpha_r^1 \\ \vdots & & \vdots \\ \alpha_2^m & \cdots & \alpha_r^m \end{pmatrix} \in \mathbb{R}^{m\times(r-1)}. \tag{3.3}$$

Elementy macierzy A będą wyznaczane kolumnami, łącznie z poszukiwanymi liczbami stanowiącymi m ostatnich składowych kolejnych wektorów $\mathbf{x}_j$ dla $j = 2, \ldots, r$. W tym celu zapiszmy wzory (2.14) w następującej postaci, gdzie poszczególne liczby $x_{(j-1)s-l}$, $l = 0, \ldots, m-1$, celowo zapisano w kolumnach jedna pod drugą:

$$\begin{cases} \alpha_j^1 = & a_m x_{(j-1)s-m+1} + \ldots + a_2 x_{(j-1)s-1} + a_1 x_{(j-1)s} \\ \alpha_j^2 = & a_m x_{(j-1)s-m+2} + \ldots + a_2 x_{(j-1)s} \\ \quad\vdots & \\ \alpha_j^m = & a_m x_{(j-1)s} \end{cases} \tag{3.4}$$

W zapisie wektorowym wzór (3.4) przyjmie postać

$$\begin{pmatrix} \alpha_j^1 \\ \alpha_j^2 \\ \vdots \\ \alpha_j^m \end{pmatrix} = x_{(j-1)s-m+1} \begin{pmatrix} a_m \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \ldots + x_{(j-1)s} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix},$$

skąd otrzymujemy

$$A_{*,j-1} = \begin{pmatrix} a_m & a_{m-1} & \cdots & a_1 \\ 0 & a_m & \cdots & a_2 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & a_m \end{pmatrix} \begin{pmatrix} x_{(j-1)s-m+1} \\ x_{(j-1)s-m+2} \\ \vdots \\ x_{(j-1)s} \end{pmatrix} = CX_{s-m+1:s,j-1}. \tag{3.5}$$

Wyznaczenie m ostatnich składowych wektorów $\mathbf{x}_j$, $j = 2, \ldots, r$ może być zrealizowane za pomocą operacji mnożenia macierzy przez wektor o postaci (2.20), zawężonej do m ostatnich składowych wyznaczanego wektora. Niech zatem macierz Y będzie zdefiniowana przez (2.18) oraz niech $X = Z$ będzie macierzą otrzymywaną jako rozwiązanie układu $LZ = F$. Wówczas wyznaczenie $X_{s-m+1:s,2:r}$ może być zrealizowane przez powtórzenie dla $j = 2, \ldots, r$ sekwencji dwóch operacji

$$\begin{cases} A_{*,j-1} \leftarrow CX_{s-m+1:s,j-1} \\ X_{s-m+1:s,j} \leftarrow X_{s-m+1:s,j} + Y_{s-m+1:s,*}A_{*,j-1}. \end{cases} \tag{3.6}$$

Pierwsza to mnożenie macierzy górnotrójkątnej przez wektor (operacja `TRMV` z biblioteki BLAS poziomu drugiego). Druga to zwykłe mnożenie macierzy przez wektor (operacja `GEMV`). Zauważmy, że w trakcie powtarzania czynności (3.6) dla kolejnych wektorów zachowujemy wyznaczane kolejno współczynniki α_j^k, stanowiące kolumny macierzy A, co wymaga dodatkowego miejsca w pamięci dla $m(r-1)$ liczb, ale jest niezbędne dla efektywnego wykonania kolejnego kroku algorytmu, w którym wyznaczymy $s-m$ pierwszych składowych wektorów $\mathbf{x}_j$, $j = 2, \ldots, r$. W tym miejscu warto zaznaczyć, że wykonanie tego kroku nie zawsze jest konieczne. W pewnych zastosowaniach interesuje nas jedynie wyznaczenie liczb $x_{n-m+1}, \ldots, x_n$, które spełniają (1.28), co będziemy nazywać cząstkowym rozwiązaniem równania.

Zauważmy teraz, że dla wyznaczenia $s-m$ pierwszych składowych wektorów $\mathbf{x}_j$, $j = 2, \ldots, r$, czyli macierzy $X_{1:s-m,2:r}$, za pomocą wzoru (2.20) okrojonego do pierwszych $s-m$ składowych wyznaczanego wektora, czyli

$$X_{1:s-m,j} \leftarrow X_{1:s-m,j} + Y_{1:s-m,*}A_{*,j-1}$$

potrzebne są jedynie odpowiednie współczynniki α_j^k (kolumny macierzy A) wyznaczone w poprzednim kroku metody oraz macierz Y. Co więcej, składowe poszczególnych wektorów $\mathbf{x}_j$ mogą być wyznaczane niezależnie. Stąd, stosując powyższy wzór jednocześnie dla $j = 2, \ldots, r$, otrzymujemy następujący wzór na wyznaczenie brakujących składowych rozwiązania (czyli elementów macierzy $X_{1:s-m,2:r}$):

$$X_{1:s-m,2:r} \leftarrow X_{1:s-m,2:r} + Y_{1:s-m,*}A. \tag{3.7}$$

Operacja o postaci (3.7) jest mnożeniem dwóch macierzy prostokątnych i jest dostępna w bibliotece BLAS poziomu 3 jako `GEMM`.

Opisane wyżej postępowanie zapiszmy w postaci algorytmu 3.1, gdzie dla poszczególnych operacji, podanych w notacji matematycznej, wskazano również ich odpowiedniki z biblioteki BLAS poziomów 2 i 3. Schemat wyznaczania poszczególnych elementów rozwiązania przedstawia rysunek 3.1. Wyznaczymy teraz liczbę operacji zmiennopozycyjnych w algorytmie 3.1.

Algorytm 3.1 . Macierzowe wyznaczanie rozwiązania liniowego równania rekurencyjnego o stałych współczynnikach (1.28) przy ustalonych wartościach całkowitych $r > 1$, $s > 1$ takich, że $rs = n$.

Wejście: liczby $a_1, \ldots, a_m$ oraz $f_1, \ldots, f_n$ tworzące wektory $\mathbf{f}_1, \ldots, \mathbf{f}_r$ zdefiniowane przez (2.8)

Wyjście: $X_{1:s,1:r} = (\mathbf{x}_1, \ldots \mathbf{x}_r)$.

1: $X \leftarrow (\mathbf{f}_1, \ldots, \mathbf{f}_r, \mathbf{e}_1)$

2: $C \leftarrow \begin{pmatrix} a_m & \cdots & a_1 \\ & \ddots & \vdots \\ 0 & & a_m \end{pmatrix}$

3: **for** $k = 2$ to s **do**

4: $\quad X_{k,*} \leftarrow X_{k,*} + C_{1,\max\{1,m-k+2\}:m} X_{\max\{1,k-m\}:k-1,*}$ {operacja `GEMV`}

5: **end for**{ $X_{*,r+1} = \mathbf{y}_1$ }

6: $Y \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_m)$ {stosując (2.18)}

7: **for** $j = 2$ to r **do**

8: $\quad A_{*,j-1} \leftarrow C X_{s-m+1:s,j-1}$ {operacja `TRMV`}

9: $\quad X_{s-m+1:s,j} \leftarrow X_{s-m+1:s,j} + Y_{s-m+1:s,*} A_{*,j-1}$ {operacja `GEMV`}

10: **end for**

11: $X_{1:s-m,2:r} \leftarrow X_{1:s-m,2:r} + Y_{1:s-m,*} A$ {operacja `smallGEMM`}

Twierdzenie 3.1. *Liczba operacji zmiennopozycyjnych w algorytmie 3.1 wynosi*

$$4mrs - 2m^2 - mr - m. \tag{3.8}$$

Dowód. Liczba operacji zmiennopozycyjnych w instrukcjach 3-5 wynosi

$$2\left(s - \frac{m+1}{2}\right)(r+1)m. \tag{3.9}$$

W instrukcjach 7-10 powtarzamy $r - 1$ razy operację `TRMV` wymagającą m^2 operacji zmiennopozycyjnych oraz `GEMV` wymagającą w tym konkretnym przypadku $2m^2$ operacji. Stąd łącznie

otrzymujemy liczbę

$$3m^2(r-1) \tag{3.10}$$

operacji zmiennopozycyjnych. Instrukcja 11, czyli operacja `GEMM`, wymaga w tym wypadku

$$2(s-m)m(r-1) \tag{3.11}$$

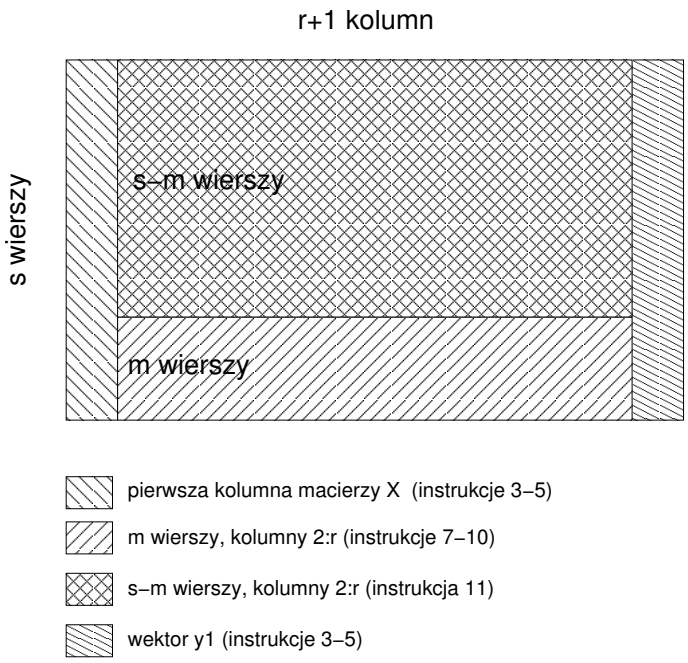
operacji. Sumując liczby (3.9), (3.10) i (3.11), otrzymujemy po prostych przekształceniach wielkość opisaną wzorem (3.8). ■

Uwzględniając konieczność wyznaczenia liczb $x_{rs+1}, \ldots, x_n$ ze wzoru (1.28), co wymaga $2(n-rs)m$ operacji, otrzymujemy następujący wniosek.

Wniosek 3.1. *Wyznaczenie rozwiązania równania (1.28) za pomocą algorytmu 3.1 wymaga wykonania*

$$2mn + 2mrs - 2m^2 - mr - m \tag{3.12}$$

operacji zmiennopozycyjnych.



Rys. 3.1. Wyznaczanie elementów rozwiązania równiania (1.28) przy użyciu algorytmu 3.1
Fig. 3.1. Finding the elements of the solution of (1.28) using Algorithm 3.1

Zatem, liczba działań arytmetycznych jest zbliżona do liczby działań algorytmów 2.1 i 2.2. Podobnie jak w przypadku algorytmów wektorowych zajmiemy się teraz wyborem optymalnych wartości parametrów r i s. Zauważmy, że funkcja opisana wzorem (3.12) nie posiada minimum lokalnego. Jednocześnie instrukcja 11 jako jedyna będzie realizowana przez wywołanie operacji mnożenia macierzy `GEMM` z trzeciego poziomu BLAS-u. Należy zatem oczekiwać, że ten fragment algorytmu będzie wykonywany z największą wydajnością. Zatem, przyjmijmy jako parametry metody wartości, które minimalizują liczbę operacji w instrukcjach 3-10, czyli wykonywane wolniej za pomocą BLAS-u poziomu drugiego, to jest wartość

$$f(r,s) = 2\left(s - \frac{m+1}{2}\right)(r+1)m + 3m^2(r-1).$$

Wyznaczając minimum lokalne funkcji $f(r,s)$ przy warunku $rs = n$ oraz obcinając optymalne wartości do liczb całkowitych, otrzymujemy

$$r^* = \left\lfloor \sqrt{\frac{2n}{2m-1}} \right\rfloor, \quad s^* = \left\lfloor \sqrt{\frac{2mn-n}{2}} \right\rfloor. \tag{3.13}$$

Podobnie jak w przypadku algorytmów wektorowych, wybór (3.13) należy tak skorygować, aby uniknąć parzystych wartości parametru s, zatem

$$s = \begin{cases} s^* - 1 & \text{dla } s^* \text{ parzystych,} \\ s^* & \text{w przeciwnym przypadku,} \end{cases} \tag{3.14}$$

a następnie

$$r = \lfloor n/s \rfloor. \tag{3.15}$$

Zauważmy również, że jeśli przyjmiemy optymalne wartości parametrów s i r zdefiniowane przez (3.14) i (3.15), to algorytm blokowy 3.1 będzie się charakteryzować dwukrotnie większą liczbą operacji arytmetycznych niż prosty algorytm 1.5. Należy jednak oczekiwać zadowalającego przyspieszenia względem algorytmu 1.5, a to z uwagi na użycie podprogramów BLAS-u wyższych poziomów.

3.2. Algorytm równoległy na komputery z pamięcią wspólną

Algorytm 3.1 może być łatwo zrównoleglony na komputery wieloprocesorowe z pamięcią wspólną. Do wykonania równoległego nadają się instrukcje 3-5 oraz 11. Instrukcje 7-9 będą stanowić sekwencyjną część algorytmu. Każdy procesor będzie działał na bloku kolumn macierzy X. Szczegóły przedstawiamy w postaci algorytmu 3.2, który może być zaimplementowany za pomocą standardu OpenMP [17].

Algorytm 3.2. Równoległe (macierzowe) wyznaczanie rozwiązania liniowego równania rekurencyjnego o stałych współczynnikach (1.28) przy ustalonych wartościach całkowitych $r > 1$, $s > 1$ takich, że $rs = n$, na p procesorach z pamięcią wspólną.

Wejście: liczby $a_1, \ldots, a_m$ oraz $f_1, \ldots, f_n$ tworzące wektory $\mathbf{f}_1, \ldots, \mathbf{f}_r$ zdefiniowane przez (2.8)

Wyjście: $X_{1:s,1:r} = (\mathbf{x}_1, \ldots \mathbf{x}_r)$.

1: $X \leftarrow (\mathbf{f}_1, \ldots, \mathbf{f}_r, \mathbf{e}_1)$

2: $C \leftarrow \begin{pmatrix} a_m & \cdots & a_1 \\ & \ddots & \vdots \\ 0 & & a_m \end{pmatrix}$

3: $t \leftarrow \lfloor (r+1)/p \rfloor$

4: **for** $i = 0$ to $p - 1$ **do in parallel**

5: $\quad t_1 \leftarrow it + 1;\ \ t_2 \leftarrow (i+1)t$

6: $\quad$ **if** $i = p - 1$ **then**

7: $\quad\quad t_2 \leftarrow r + 1$

8: $\quad$ **end if**

9: $\quad$ **for** $k = 2$ to s **do**

10: $\quad\quad X_{k,t_1:t_2} \leftarrow X_{k,t_1:t_2} + C_{1,\max\{1,m-k+2\}:m} X_{\max\{1,k-m\}:k-1,t_1:t_2}$ { `GEMV`}

11: $\quad$ **end for**

12: **end for**{ $X_{*,r+1} = \mathbf{y}_1$}

13: $Y \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_m)$ {stosując (2.18)}

14: **for** $j = 2$ to r **do**

15: $\quad A_{*,j-1} \leftarrow CX_{s-m+1:s,j-1}$ {operacja `TRMV`}

16: $\quad X_{s-m+1:s,j} \leftarrow X_{s-m+1:s,j} + Y_{s-m+1:s,*} A_{*,j-1}$ {operacja `GEMV`}

17: **end for**

18: $t \leftarrow \lfloor r/p \rfloor$

19: **for** $i = 0$ to $p - 1$ **do in parallel**

20: $\quad t_1 \leftarrow it + 1;\ \ t_2 \leftarrow (i+1)t$

21: $\quad$ **if** $i = 0$ **then**

22: $\quad\quad t_1 \leftarrow t_1 + 1$

23: $\quad$ **end if**

24: $\quad$ **if** $i = p - 1$ **then**

25: $\quad\quad t_2 \leftarrow r$

26: $\quad$ **end if**

27: $\quad X_{1:s-m,t_1:t_2} \leftarrow X_{1:s-m,t_1:t_2} + Y_{1:s-m,*} A$ {operacja `GEMM`}

28: **end for**

Rysunek 3.2 przedstawia sposób organizacji obliczeń równoległych w pętli opisanej w liniach 4–12. Najpierw wywoływany jest podprogram ustalający liczbę działających równolegle wątków na wartość równą liczbie dostępnych procesorów. Następnie pętlę, która ma być wykonana równolegle (dla każdego procesora jedna iteracja), poprzedzamy dyrektywą `parallel do`, w której wskazujemy zmienne lokalne, służące do definiowania numerów kolumn przetwarzanych przez dany procesor. Domyślnie, wszystkie zmienne, które nie zostały wyspecyfikowane w klauzuli `private`, są traktowane jako wspólne. Dla każdego wątku ustalane są wartości zmiennych lokalnych `t1`, `t2` oraz zmiennej `d` opisującej liczbę kolumn przetwarzanych przez dany procesor. Następnie w pętli odpowiadającej liniom 9–11 wywoływana jest operacja `SGEMV`, czyli mnożenie macierzy przez wektor dla pojedynczej precyzji. Organizacja pętli 19-28 jest podobna.

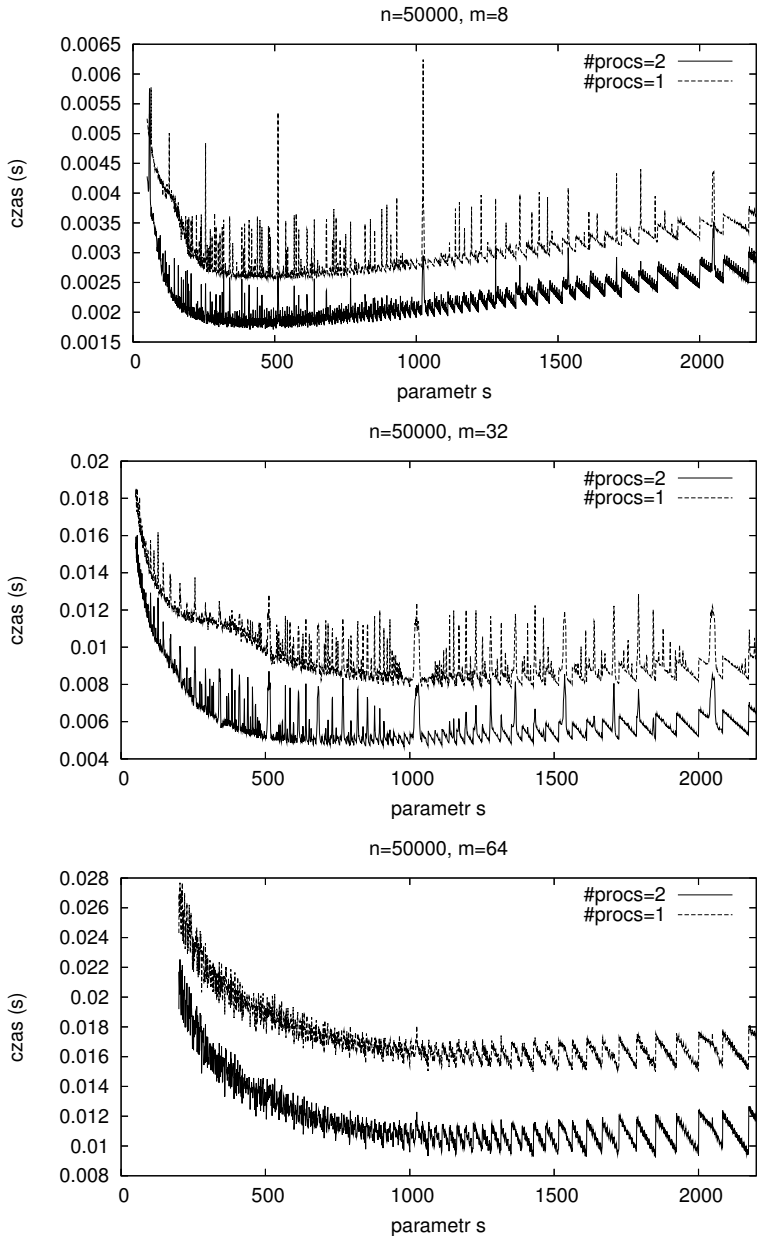
```
      call omp_set_num_threads(p)
!$omp parallel do private(k,t1,t2,d) default(shared)
      do i=0,p-1
        t1=i*t+1
        t2=(i+1)*t
        if (i.eq.p-1) then
          t2=r+1
        end if
        d=t2-t1+1
        do k=2,s
          call sgemv('T',min(k-1,m),d,1.0,x(max(1,k-m),t1),
     &               s,c(1,max(1,m-k+2)),maxm,1.0,x(k,t1),s)
        end do
      end do
```

Rys. 3.2. Organizacja obliczeń równoległych w pętli 4–12 algorytmu 3.2 przy użyciu OpenMP
Fig. 3.2. Parallelizing of the loop 4–12 in Algorithm 3.2 using OpenMP

3.3. Wyniki eksperymentów

Algorytmy 3.1 i 3.2 zostały zaimplementowane w języku Fortran z wykorzystaniem standardu OpenMP na czterech platformach: dwuprocesorowym komputerze Intel Pentium III, dwunastoprocesorowym Sun UltraSPARC II, czterech procesorach komputera Cray SV1 oraz dwuprocesorowym komputerze Quad-Core Xeon.

Rysunek 3.3 przedstawia czas działania obu algorytmów na komputerze Intel Pentium dla różnych wartości m, względem różnych warości parametru s. Szybkość działania zależy od rzędu równania (m) oraz wartości s. Podobnie jak w przypadku algorytmu 2.1 wykonywanego na komputerach Cray, czas działania algorytmów znacznie rośnie dla $s = a2^k$, co jest efektem zjawiska *cache miss*.



Rys. 3.3. Czas działania algorytmów 3.1 (#procs=1) i 3.2 (#procs=2) na dwuprocesorowym komputerze Pentium III 866 MHz dla $n = 50000$, $m = 8, 32, 64$ i różnych wartości s. Optymalne wartości parametru s wynoszą odpowiednio 611, 1253, 1787

Fig. 3.3. Execution time of the algorithms 3.1 (#procs=1) and 3.2 (#procs=2) on a dual-processor Pentium III 866 MHz for $n = 50000$, $m = 8, 32, 64$ and various s. The optimal values of s are 611, 1253, 1787 respectively

Wybór parametrów s i r określony wzorami (3.14) i (3.15) należy uznać za względnie dobre przybliżenie optymalnych wartości tych parametrów, choć w praktyce za wartość s lepiej jest przyjąć około 4/5 wartości określonej wzorem (3.13), gdyż rozmiar tablic ma wpływ na efektywny sposób wykorzystania pamięci podręcznej.

Rysunek 3.4 pokazuje przykładowy czas wykonania algorytmów 3.1 i 3.2 na komputerze z procesorem Pentium III dla rozwiązania pełnego (F) oraz cząstkowego (P), gdzie wyznaczamy tylko m ostatnich składowych rozwiązania. W obu przypadkach zastosowanie algorytmu 3.2, czyli użycie dwóch procesorów, daje istotny przyrost przyspieszenia, które na ogół rośnie wraz ze wzrostem m, choć dla pewnych wartości $m = a2^k$ efektywność algorytmu drastycznie spada, co jest związane ze zjawiskiem braku potrzebnych danych w pamięci podręcznej w trakcie wykonywania podprogramów z biblioteki BLAS (*cache miss*). Rysunek 3.5 pokazuje wydajność komputera z procesorem Pentium III wykonującego algorytmy 3.1 i 3.2. Dla algorytmu 3.2 zastosowanego do rozwiązania pełnego wynosi ona maksymalnie 1400 Mflops, co stanowi około 50% teoretycznej maksymalnej wydajności komputera, podczas gdy wydajność dla algorytmu 1.5 to zaledwie około 145 Mflops.

Rysunki 3.6, 3.7, 3.8 pokazują odpowiednio czas działania, wydajność i przyspieszenie algorytmów 3.1 i 3.2 względem algorytmu 1.5 na komputerze Sun UltraSPARC II dla mniejszego ($n = 1000000$) i większego ($n = 4000000$) rozmiaru problemu dla wyznaczania rozwiązania pełnego i cząstkowego, przy z zmieniającej się wartości m. Algorytm 3.1 działa nieco wolniej niż algorytm 1.5, ale sytuacja poprawia się dla algorytmu równoległego 3.2 przy odpowiednio dużych wartościach n i m. Przyspieszenie jest na ogół większe niż liczba procesorów, co jest spowodowane efektywniejszym wykonywaniem operacji BLAS-u na większych macierzach (lepsze wykorzystanie pamięci podręcznej). Użycie większej liczby procesorów jest korzystne dla większych wartości n. Dla $n = 1000000$ użycie dwunastu procesorów na ogół nie daje wzrostu przyspieszenia w porównaniu do przyspieszenia na ośmiu procesorach. Oczywiście, przyspieszenie jest znacznie większe dla wyznaczania rozwiązania cząstkowego, gdyż wówczas nie są wykonywane fragmenty algorytmów, które zawierają wywołania podprogramu `GEMM`, służącego do wyznaczenia $s - m$ pierwszych składowych kolumn macierzy X.

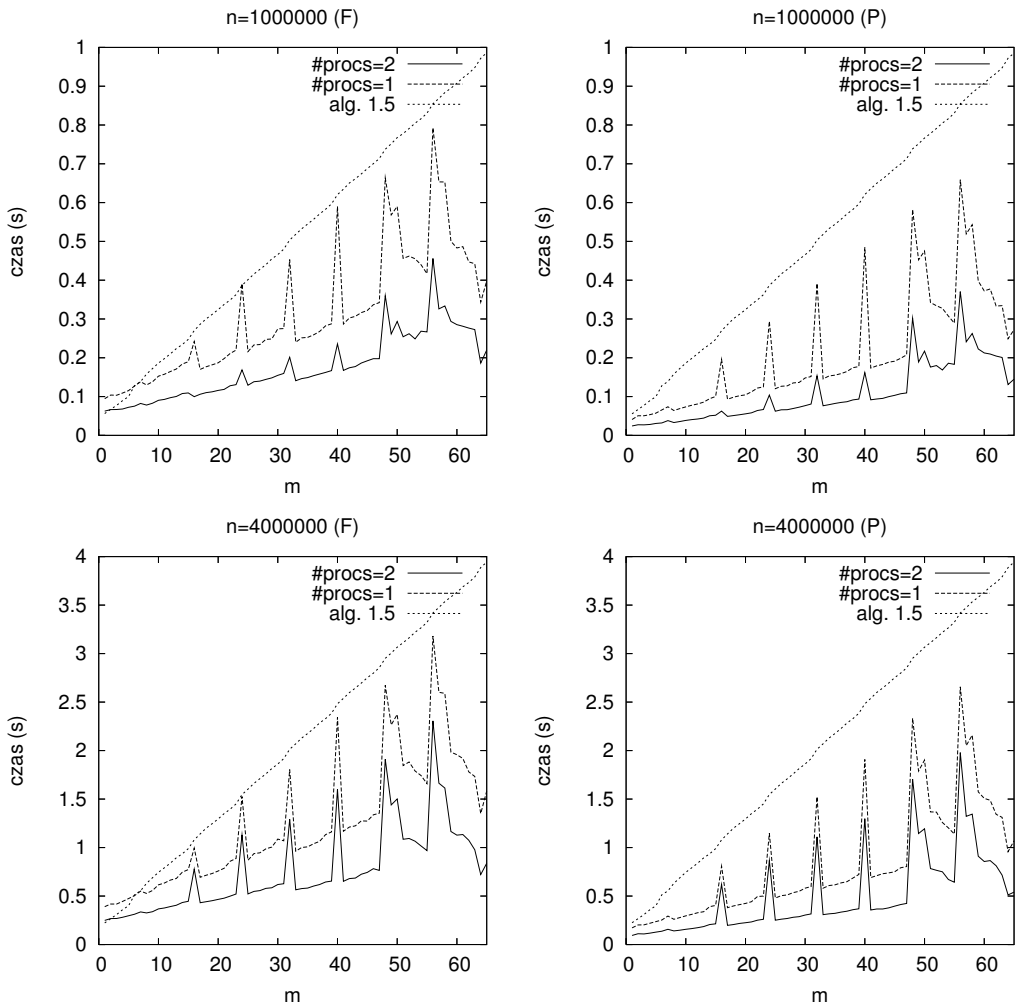
W trakcie testów przyjęto za s wartość określoną przez (3.14). Jednak w wielu przypadkach może być ona tak dostrojona, by algorytmy działały szybciej. Szczególnie dla większych wartości m korzystnie jest przyjąć za s wartość mniejszą, nawet $s^*/2$. Na rysunku 3.8 można zaobserwować, że dla $n = 1000000$ i wartości $m \in [4, 10]$ algorytm osiąga znaczne przyspieszenie, a dla $n = 4000000$ i $m > 48$ można zaobserwować pewien spadek przyspieszenia. W pierwszym

przypadku wzór (3.14) bardzo dobrze określa optymalną wartość parametru, a drugim znacznie gorzej.

Rysunki 3.9, 3.10, 3.11 pokazują odpowiednio czas działania, wydajność i przyspieszenie rozważanych algorytmów względem algorytmu 1.5 na komputerze Cray SV1. Można zaobserwować spadek przyspieszenia przy rosnącej wartości m, przy jednoczesnym wzroście wydajności. Jest to spowodowane wzrostem wydajności osiąganej dla algorytmu 1.5 przy większej wartości m, co jest z kolei spowodowane wektoryzacją pętli wewnętrznej w algorytmie 1.5. Można również zaobserwować wzrost przyspieszenia przy rosnącej wartości n oraz znacznie większe przyspieszenia przy wyznaczaniu rozwiązania cząstkowego. W przypadku rozwiązania pełnego algorytmy są wykonywane znacznie bardziej wydajnie niż przy wyznaczaniu rozwiązania częściowego. Jest to spowodowane bardzo szybkim wykonaniem operacji `GEMM`, która stanowi prawie połowę ogólnej liczby operacji zmiennopozycyjnych wykonywanych w obu algorytmach. Użycie większej liczby procesorów staje się korzystniejsze dla większych rozmiarów problemu.

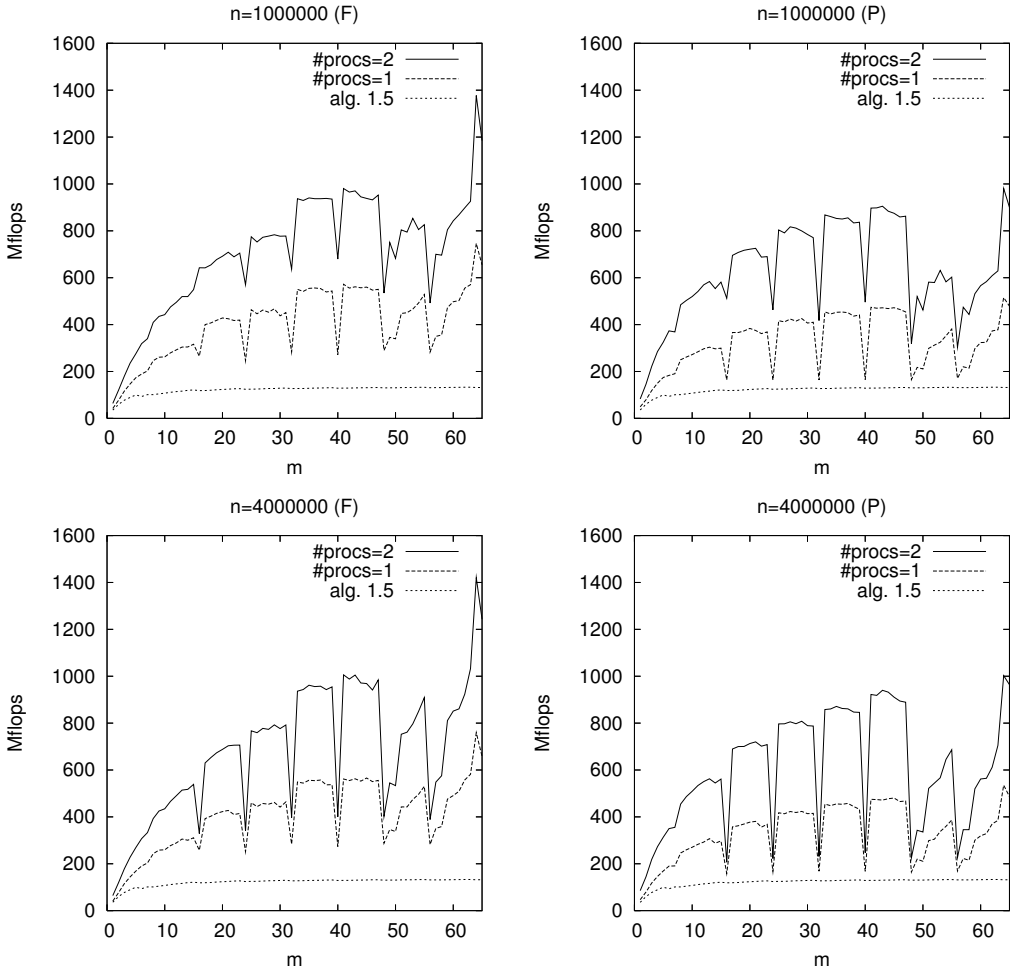
Rysunki 3.12–3.14 pokazują analogiczne wyniki eksperymentów na dwuprocesorowym komputerze Quad-Core Xeon (odpowiednio czas, wydajność i przyspieszenie). Można zaobserwować, że czas działania algorytmu 3.1 jest zbliżony do czasu działania algorytmu 1.5. Użycie większej liczby rdzeni (odpowiednio 4 i 8) i zastosowanie algorytmu 3.2 daje istotne skrócenie czasu obliczeń. Przyspieszenie oraz wydajność na ogół rosną wraz ze wzrostem wartości n i m.

Na zakończenie odnotujmy fakt, że rozważane w tym rozdziale algorytmy 3.1 i 3.2 charakteryzują się znacznie lepszym przyspieszeniem niż inne znane algorytmy [9, 89, 90]. W porównaniu do zaprezentowanej w pracy [90] oryginalnej i zmodyfikowanej metody Wanga, algorytm 3.2 osiąga znacznie większe przyspieszenie względem algorytmu 1.5. Metoda Wanga dla liczby procesorów od 10 do 12 daje przyspieszenie między 1 a 4 tylko dla niewielkich wartości m ($m \leq 2$). Dla większych wartości przyspieszenie jest jeszcze mniejsze. W pracy [89] pokazano, że metoda Wanga zastosowana do wyznaczenia rozwiązania (1.28) daje na ośmiu procesorach komputera Cray Y-MP przyspieszenie nie większe niż 3. Znacznie nowsze wyniki zaprezentowano w pracy [9]. Algorytm zaproponowany przez R. Barrio i współautorów daje dobre przyspieszenie na komputerze Cray T3D. Jednak do osiągnięcia przyspieszenia na poziomie wartości 5 potrzeba aż szesnastu procesorów, a na czterech procesorach przyspieszenie nie przekracza wartości 2, nawet dla niewielkich wartości m. Kluczowe dla osiąganego dużego przyspieszenia algorytmów 3.1 i 3.2 względem algorytmu 1.5 jest użycie podprogramów z poziomów drugiego i trzeciego biblioteki BLAS oraz możliwość wykorzystania mechanizmów wektorowości.



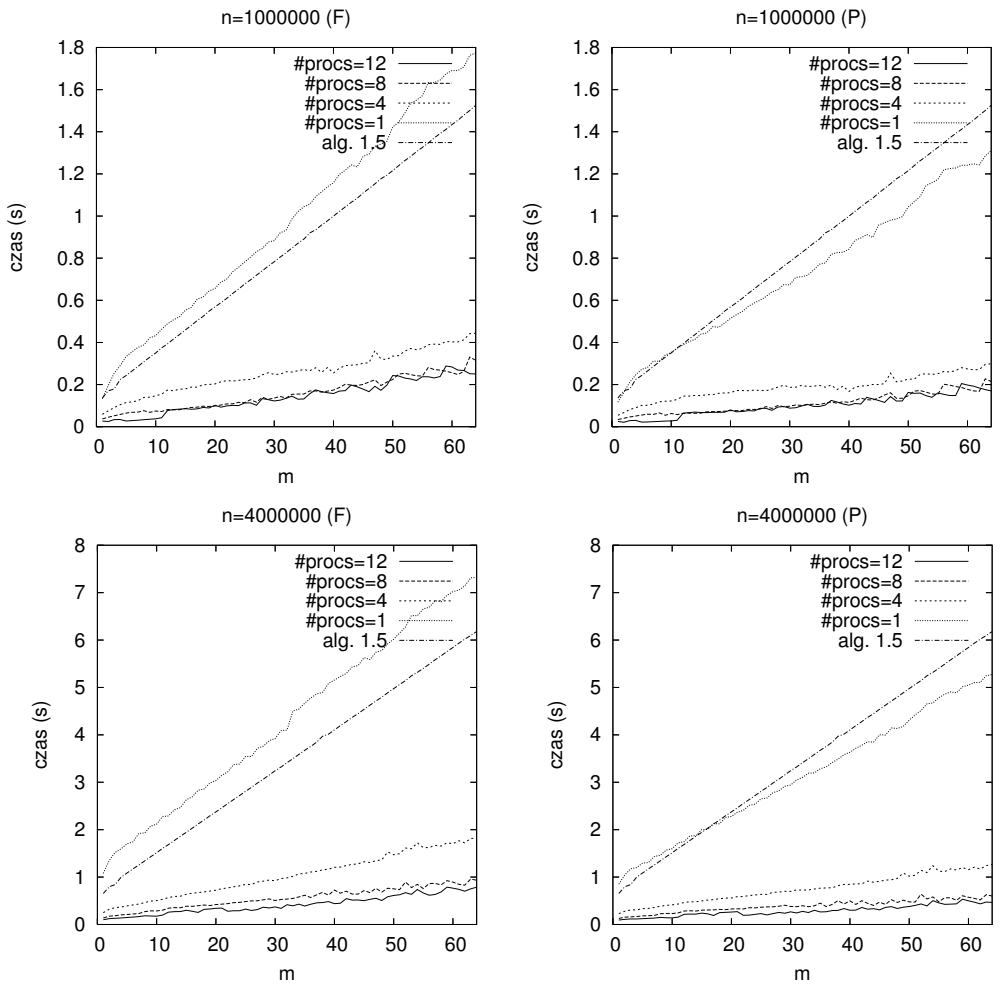
Rys. 3.4. Czas wykonania algorytmów 3.1 (#procs=1), 3.2 (#procs=2) i 1.5 na komputerze Intel Pentium III dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.4. Execution time of the algorithms 3.1, 3.2 and 1.5 for finding full (F) and partial (P) solution on a Pentium III



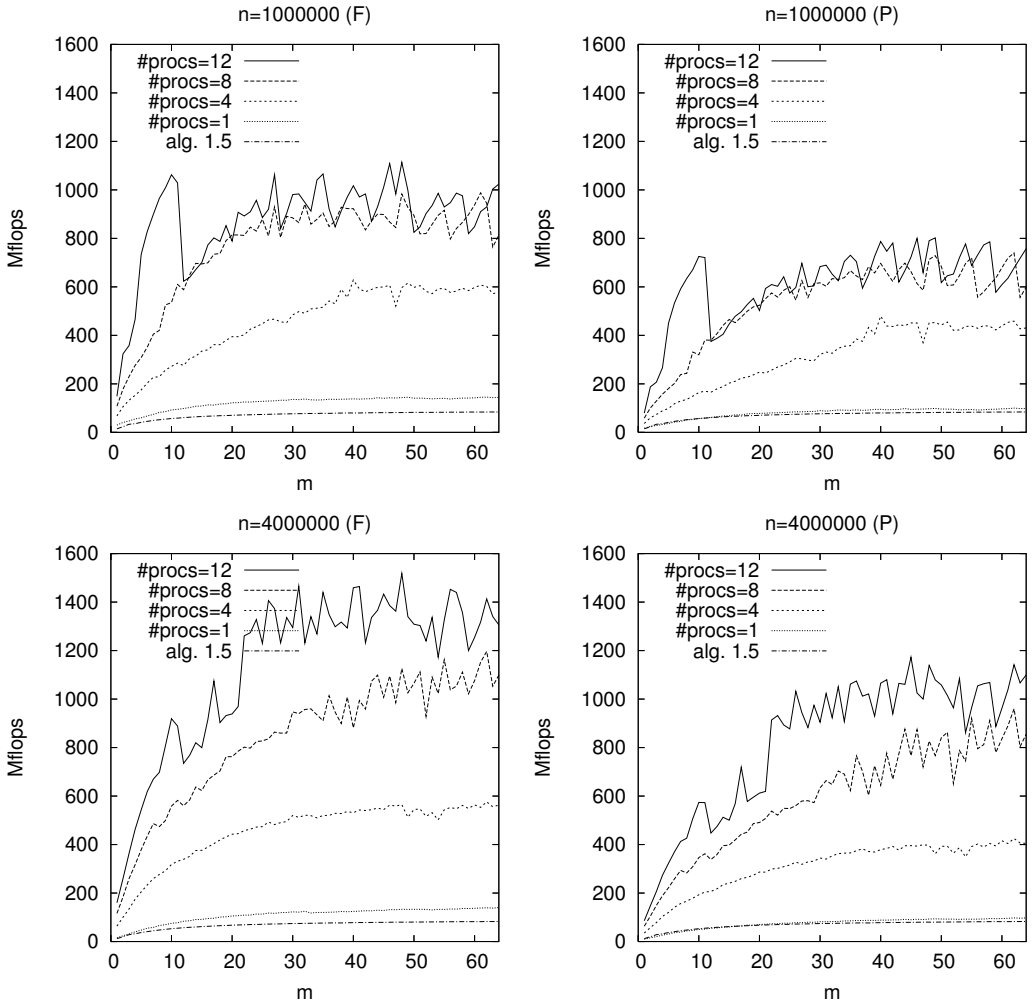
Rys. 3.5. Wydajność Pentium III dla algorytmów 3.1 (#procs=1), 3.2 (#procs=2) oraz 1.5 przy wyznaczeniu rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.5. Performance of the algorithms 3.1 (#procs=1), 3.2 (#procs=2) and 1.5 for finding full (F) and partial (P) solution on a Pentium III



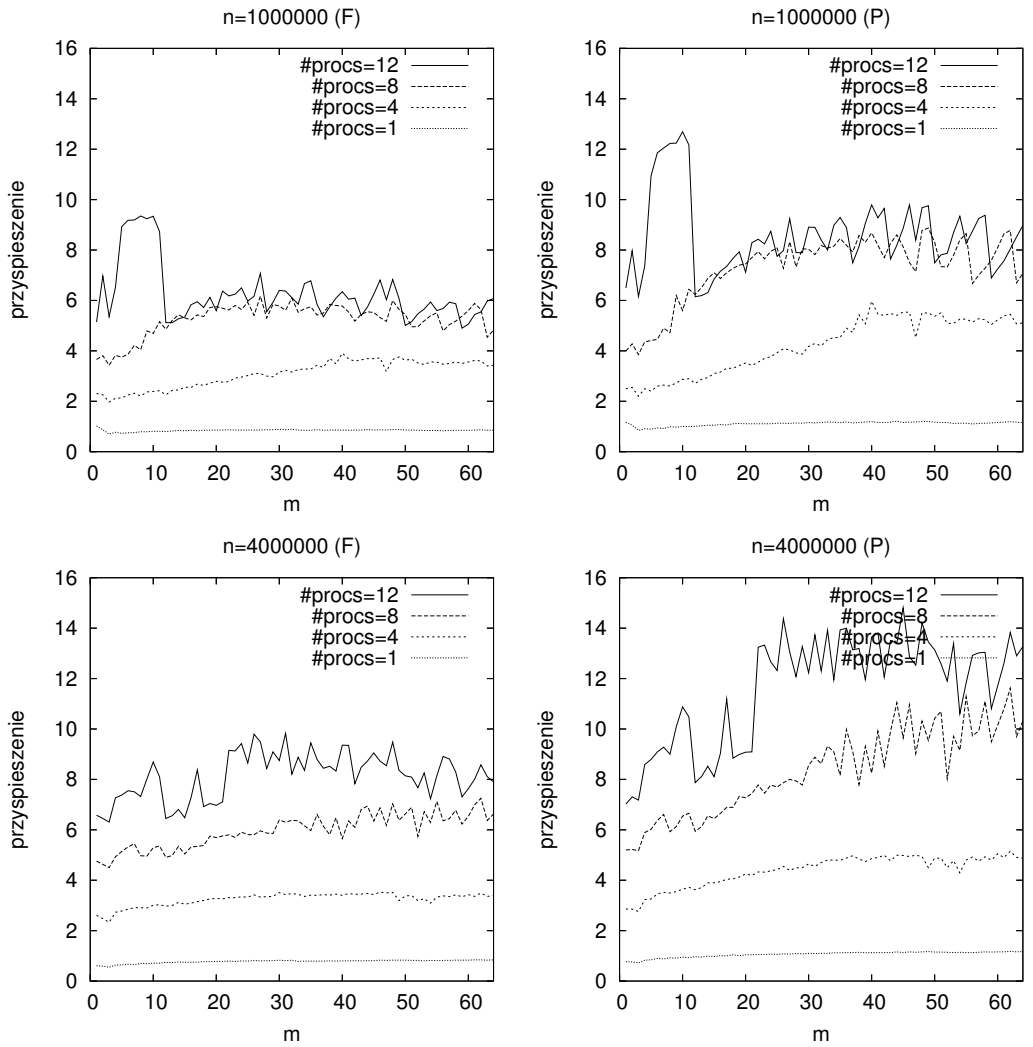
Rys. 3.6. Czas działania algorytmów 3.1 (#procs=1), 3.2 (#procs=4, 8, 12) i 1.5 na komputerze UltraSPARC II dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.6. Execution time of the algorithms 3.1 (#procs=1), 3.2 (#procs=4, 8, 12) and 1.5 for finding full (F) and partial (P) solution on an UltraSPARC II



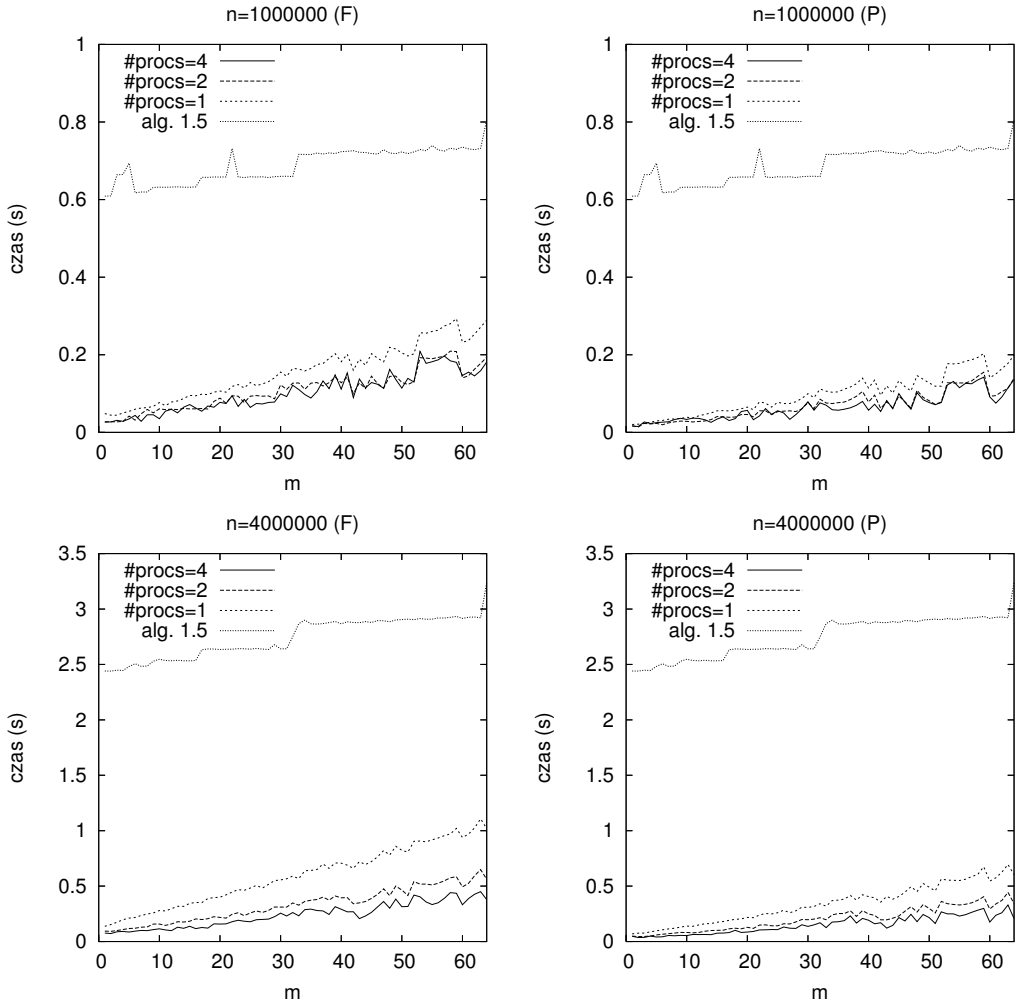
Rys. 3.7. Wydajność komputera UltraSPARC II dla algorytmów 3.1 (#procs= 1), 3.2 (#procs= 4, 8, 12) i 1.5 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.7. Performance of the algorithms 3.1 (#procs=1), 3.2 (#procs=4, 8, 12) and 1.5 for finding full (F) and partial (P) solution on an UltraSPARC II



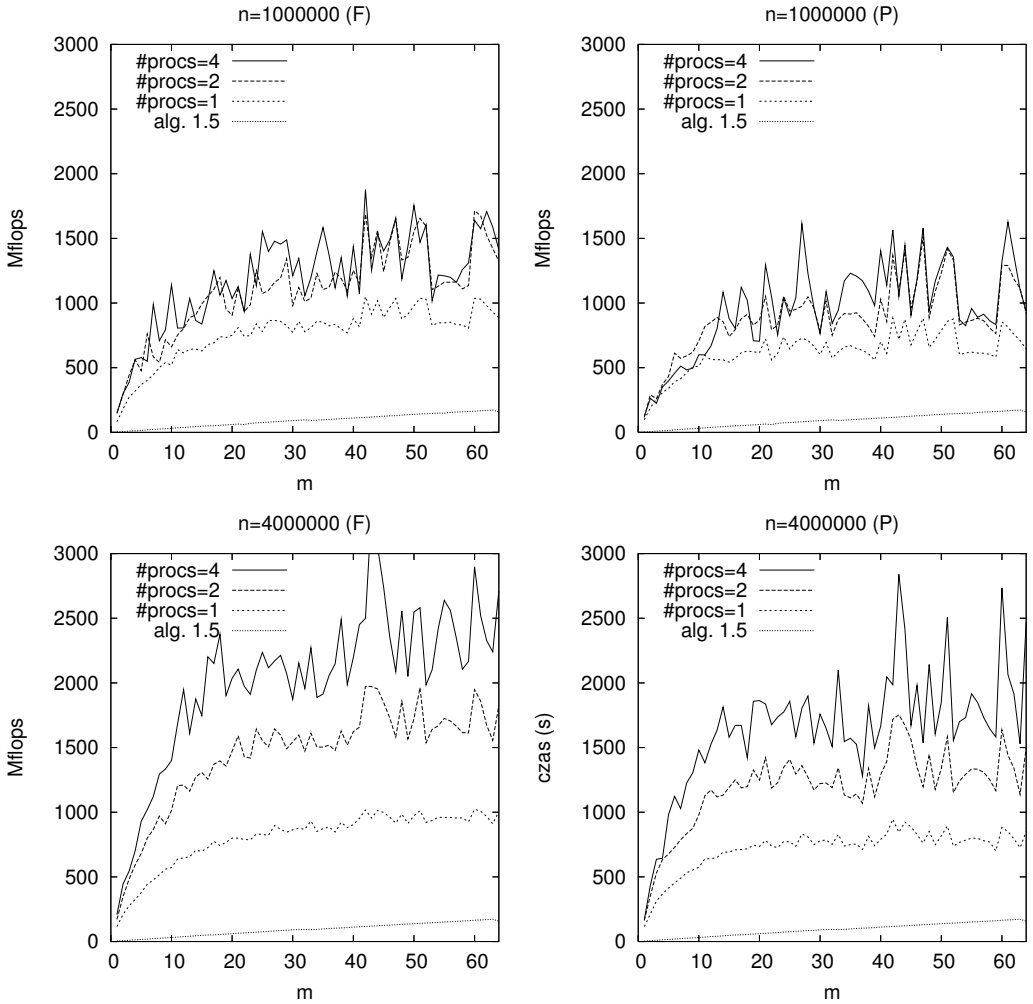
Rys. 3.8. Przyspieszenie algorytmów 3.1 (#procs=1) i 3.2 (#procs=4, 8, 12) względem algorytmu 1.5 na komputerze UltraSPARC II dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.8. Speedup of the algorithms 3.1 (#procs=1) and 3.2 (#procs=4, 8, 12) relative to Algorithm 1.5 for finding full (F) and partial (P) solution on an UltraSPARC II



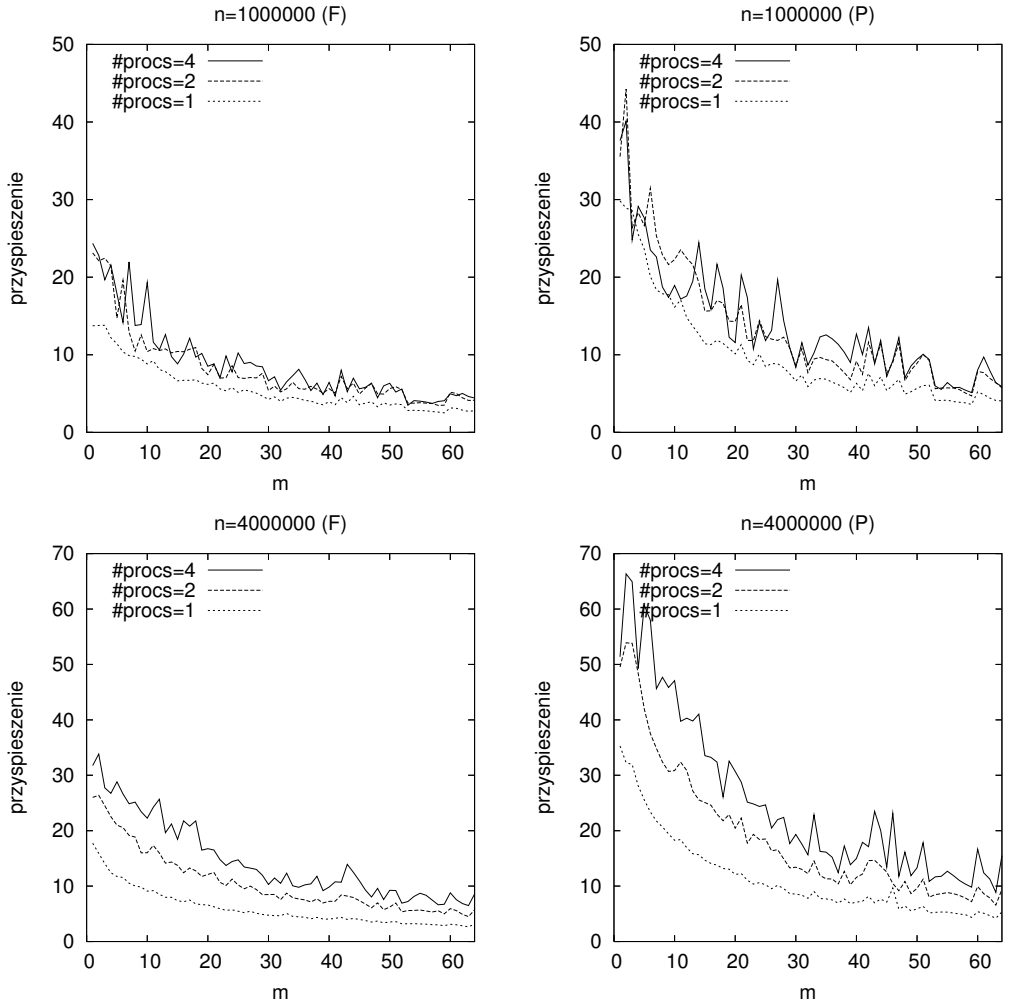
Rys. 3.9. Czas wykonania algorytmów 3.1 (#procs=1), 3.2 (#procs=2, 4) i 1.5 na komputerze Cray SV1 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.9. Execution time of the algorithms 3.1 (#procs=1), 3.2 (#procs=2, 4) and 1.5 for finding full (F) and partial (P) solution on a Cray SV1



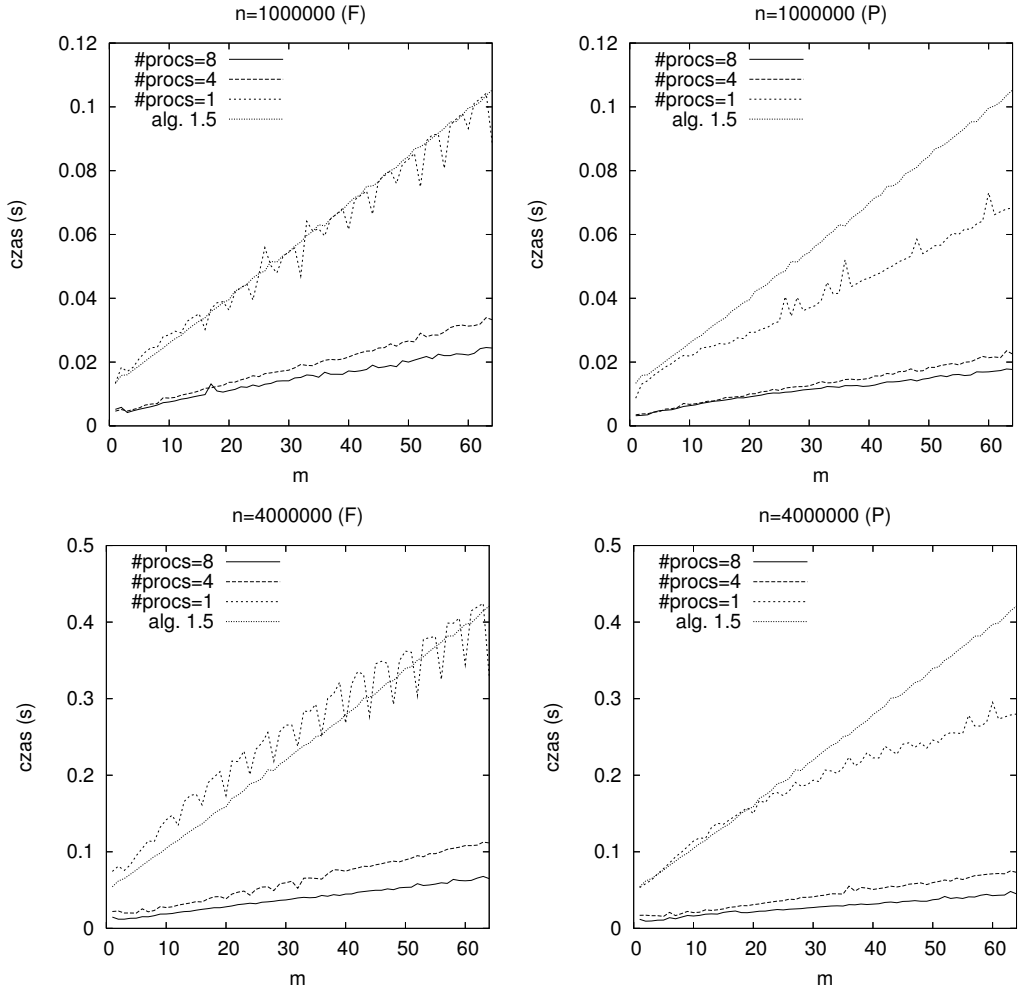
Rys. 3.10. Wydajność komputera Cray SV1 dla algorytmów 3.1 (#procs=1), 3.2 (#procs=2, 4) i 1.5 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.10. Performance of the algorithms 3.1 (#procs=1), 3.2 (#procs=2, 4) and 1.5 for finding full (F) and partial (P) solution on a Cray SV1



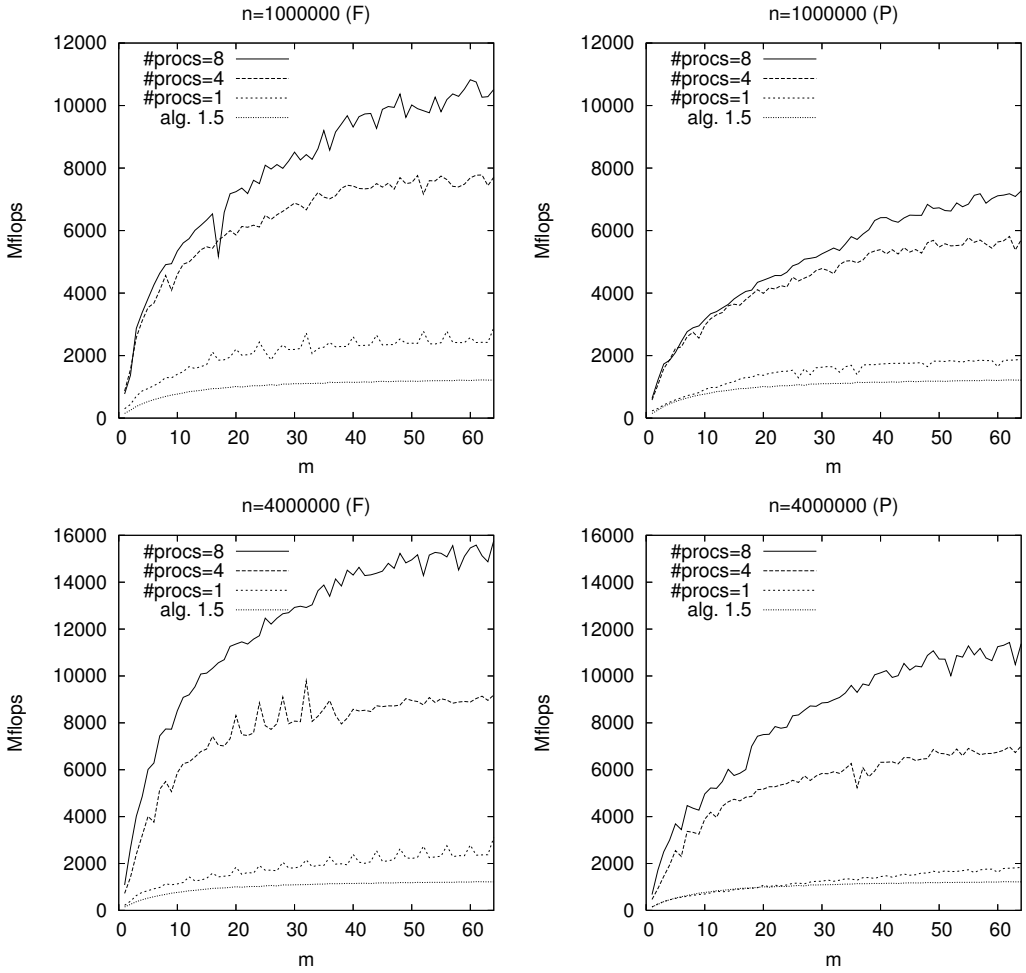
Rys. 3.11. Przyspieszenie algorytmów 3.1 (#procs=1) i 3.2 (#procs=2, 4) względem algorytmu 1.5 na komputerze Cray SV1 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.11. Speedup of the algorithms 3.1 (#procs=1) and 3.2 (#procs=2, 4) relative to Algorithm 1.5 for finding full (F) and partial (P) solution on a Cray SV1



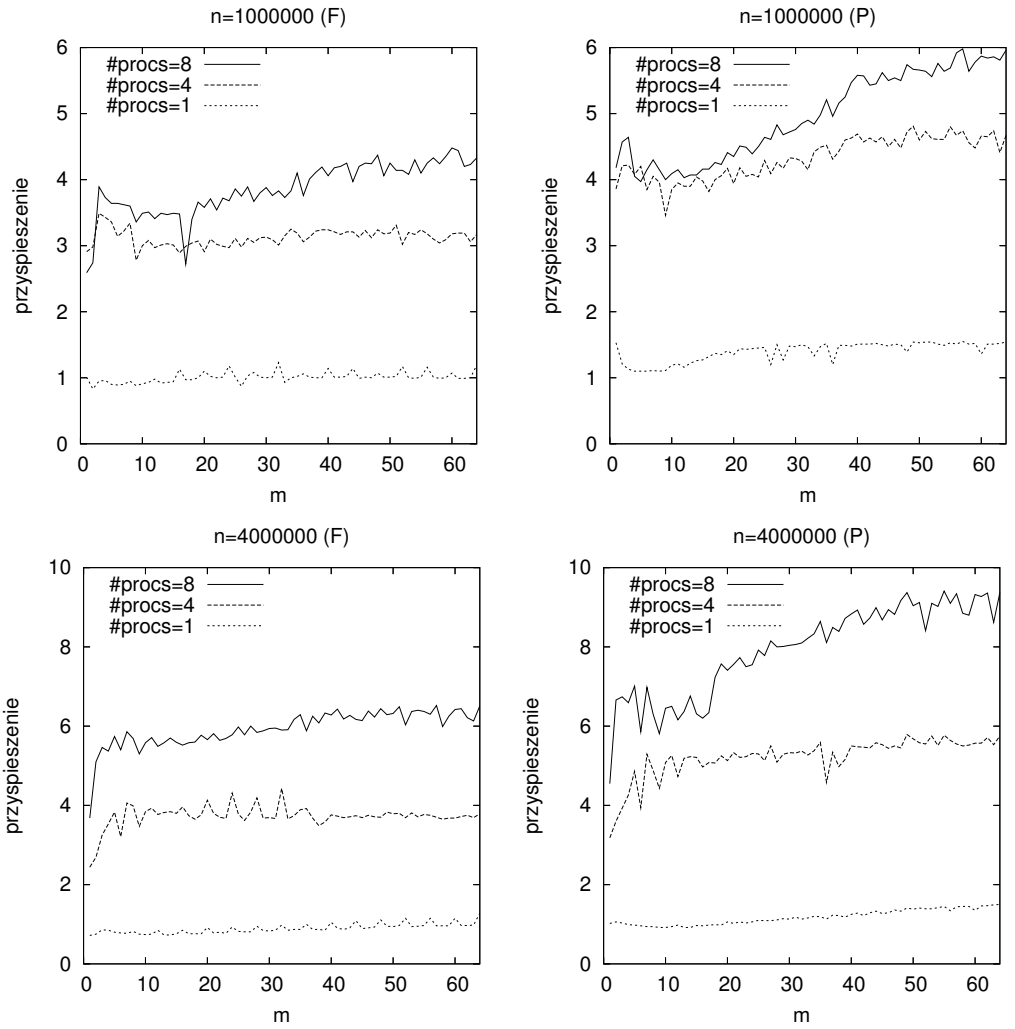
Rys. 3.12. Czas wykonania algorytmów 3.1 (#procs=1), 3.2 (#procs=4,8) i 1.5 na dwuprocesorowym komputerze Quad-Core Xeon dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.12. Execution time of the algorithms 3.1 (#procs=1) 3.2 (#procs=4,8) and 1.5 for finding full (F) and partial (P) solution on a dual processor Quad-Core Xeon



Rys. 3.13. Wydajność dwuprocesorowego komputera Xeon Quad-Core dla algorytmów 3.1 (#procs=1), 3.2 (#procs=4,8) i 1.5 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.13. Performance of the algorithms 3.1 (#procs=1) 3.2 (#procs=4,8) and 1.5 for finding full (F) and partial (P) solution on a dual processor Quad-Core Xeon



Rys. 3.14. Przyspieszenie algorytmów 3.1 (#procs=1) i 3.2 (#procs=4,8) względem algorytmu 1.5 na dwuprocesorowym komputerze Quad-Core Xeon dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 3.14. Speedup of the algorithms 3.1 (#procs=1) and 3.2 (#procs=4,8) relative to Algorithm 1.5 for finding full (F) and partial (P) solution on a dual processor Xeon Quad-Core

4. Obliczenia rekurencyjne na komputerach z pamięcią rozproszoną

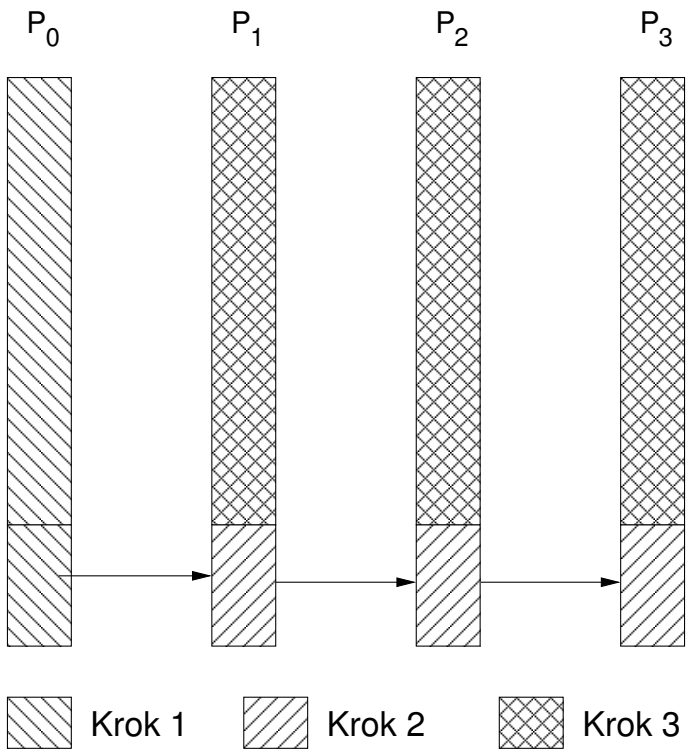
Algorytm 3.1 może być zaimplementowany na komputerach równoległych z pamięcią rozproszoną oraz klastrach. Oczywiście taka implementacja w środowisku rozproszonym będzie wymagała innych narzędzi programistycznych niż w przypadku komputerów z pamięcią wspólną, na przykład środowiska MPI [85], umożliwiającego programowanie w stylu SPMD oraz przede wszystkim opracowania metody rozmieszczenia danych w pamięciach lokalnych poszczególnych procesorów.

4.1. Rozproszone podejście *divide and conquer*

Opierając się na wzorach (2.5) i (2.6), można skonstruować algorytm *divide and conquer* działający w środowisku rozproszonym. Niech p oznacza liczbę dostępnych procesorów. Algorytm będzie działał jako grupa p procesów P_i, $i = 0, \ldots, p-1$, każdy wykonywany na oddzielnym procesorze. Każdy proces będzie wyznaczać jeden wektor $\mathbf{x}_j$. W tym celu przechowuje w pamięci jeden wektor $\mathbf{f}_j$ oraz wszystkie współczynniki a_k, $k = 1, \ldots, m$. Na początku (krok 1) każdy proces realizuje algorytm 1.5 dla wyznaczenia rozwiązania układu równań $L\mathbf{z}_j = \mathbf{f}_j$ oraz dodatkowo, aby uniknąć zbędnego przesyłania danych, każdy proces inny niż odpowiedzialny za wyznaczenie $\mathbf{x}_1$ realizuje algorytm 1.5 do wyznaczenia rozwiązania $L\mathbf{y}_1 = \mathbf{e}_1$. Po zakończeniu tego kroku, pierwszy proces ma już wyznaczony wektor $\mathbf{x}_1$ i wysyła jego m ostatnich składowych do procesu wyznaczającego $\mathbf{x}_2$. W drugim kroku każdy proces (inny niż pierwszy) odbiera m liczb, za ich pomocą wyznacza ze wzoru (2.6) m ostatnich składowych swojego wektora $\mathbf{x}_j$, i o ile nie jest to proces ostatni, wysyła te liczby do następnego procesu. W ostatnim (trzecim) kroku wszystkie procesy (z wyjątkiem pierwszego) stosują wzór (2.5) do wyznaczenia pozostałych składowych swojego wektora. Schemat komunikacji i kolejność wyznaczania poszczególnych elementów rozwiązania przedstawia rysunek 4.1.

Omówione wyżej podejście do wyznaczenia rozwiązania problemu (1.28) na komputerach z pamięcią rozproszoną ma kilka wad. Algorytm może wykorzystać operację `GEMV` z drugiego poziomu biblioteki BLAS jedynie w trzecim kroku przy wykorzystaniu wzoru (2.20). Krok drugi może być oparty na podprogramach BLAS-u poziomu pierwszego (jak w algorytmie 2.1) lub

drugiego (jak w algorytmie 3.1), co może mieć istotny wpływ na wzrost wydajności dla większych wartości m. Najmniej efektywny jest krok pierwszy, składający się wyłącznie z obliczeń skalarnych opartych na algorytmie 1.5. Dodatkowo, pierwszy proces wykonuje obliczenia tylko w ramach kroku pierwszego. Pozostałe procesy wykonują w kroku drugim dwukrotnie więcej obliczeń, zatem ma miejsce stosunkowo słabo zrównoważony podział zadań między procesy.



Rys. 4.1. Wyznaczanie kolumn rozwiązania w podejściu *divide and conquer*. Każdy proces wyznacza jeden wektor $\mathbf{x}_j$ (zaznaczony jako jedna kolumna)

Fig. 4.1. Finding columns of the solution using the *divide and conquer* approach. Each process finds one vector $\mathbf{x}_j$ (drawn as one column)

W pracy [108] zamieściliśmy wyniki eksperymentów, dotyczących między innymi efektywności opisanego wyżej podejścia *divide and conquer*, które pokazują jego stosunkowo niską efektywność. Algorytm charakteryzuje się małym przyspieszeniem oraz nawet spadkiem przyspieszenia dla $m > 2$. Dodatkowo, wymaga większej liczby działających równolegle procesorów (około 10), aby osiągnąć przyspieszenie większe niż 2. Jest zatem bardzo mało efektywny. Jego użycie jest zasadne jedynie w przypadku wyznaczania rozwiązania cząstkowego, co zostało pokazane w pracy [106].

4.2. Rozproszony algorytm wykorzystujący BLAS poziomów 2 i 3

W pracy [108] wskazaliśmy, że efektywność podejścia *divide and conquer* może być zwiększona przez przydział każdemu procesowi większej liczby (bloku) kolumn, podobnie jak miało to miejsce w algorytmie 3.2.

Niech p oznacza liczbę procesów. Każdy proces P_i, $i = 0, \ldots, p-1$, będzie wyznaczać liczby x_i, tworzące blok kolumn macierzy X. Oczywiście, na ogół liczba p nie będzie dzielnikiem liczby r, zatem zdefiniujmy liczby całkowite

$$t = \lfloor r/p \rfloor, \qquad t' = r - (p-1)t.$$

Każdy proces P_i, $i = 0, \ldots, p-2$, będzie wyznaczać macierz

$$X^{(i)} = (\mathbf{x}_{it+1}, \ldots, \mathbf{x}_{(i+1)t}) \in \mathbb{R}^{s \times t}, \tag{4.1}$$

a proces P_{p-1} macierz

$$X^{(p-1)} = (\mathbf{x}_{(p-1)t+1}, \ldots, \mathbf{x}_r) \in \mathbb{R}^{s \times t'}, \tag{4.2}$$

przy czym wektory $\mathbf{x}_j$ są zdefiniowane wzorem (2.8). Dodatkowo, zdefiniujmy macierze $Z^{(i)}$ oraz $F^{(i)}$ dla $i = 0, \ldots, p-2$

$$\begin{aligned} Z^{(i)} &= (\mathbf{z}_{it+1}, \ldots, \mathbf{z}_{(i+1)t}, \mathbf{y}_1), \\ F^{(i)} &= (\mathbf{f}_{it+1}, \ldots, \mathbf{f}_{(i+1)t}, \mathbf{e}_1) \in \mathbb{R}^{s \times (t+1)} \end{aligned} \tag{4.3}$$

oraz

$$\begin{aligned} Z^{(p-1)} &= (\mathbf{z}_{(p-1)t+1}, \ldots, \mathbf{z}_r, \mathbf{y}_1), \\ F^{(p-1)} &= (\mathbf{f}_{(p-1)t+1}, \ldots, \mathbf{f}_r, \mathbf{e}_1) \in \mathbb{R}^{s \times (t'+1)}, \end{aligned} \tag{4.4}$$

gdzie wektory $\mathbf{z}_j$ i $\mathbf{f}_j$ są zdefiniowane wzorami (2.8) i (2.10). Każdy proces P_i będzie posiadał wszystkie współczynniki a_j, $j = 1, \ldots, m$, oraz liczby f_j tworzące odpowiednie kolumny macierzy $F^{(i)}$. W pierwszym kroku algorytmu proces P_i będzie wyznaczał rozwiązanie układu równań liniowych

$$LZ^{(i)} = F^{(i)}$$

za pomocą sekwencji wywołań operacji `GEMV`, czyli przy założeniu, że początkowo zachodzi $Z^{(i)} = F^{(i)}$, dla $k = 2, \ldots, s$ będzie realizowana operacja

$$Z^{(i)}_{k,*} \leftarrow Z^{(i)}_{k,*} + C_{1,\max\{1,m-k+2\}:m} Z^{(i)}_{\max\{1,k-m\}:k-1,*}, \tag{4.5}$$

podobna do (3.2), ale ograniczona do lokalnych (posiadanych przez dany proces) kolumn macierzy X. Na zakończenie tego kroku każdy proces wykorzystuje ostatnią kolumnę wyznaczonej macierzy $Z^{(i)}$ i tworzy (posługując się operacją `COPY`) macierz Y zdefiniowaną wzorem (2.18).

Następnie (krok drugi) pierwszy proces wyznacza m ostatnich składowych swoich kolumn macierzy X oraz odpowiednie współczynniki α_j^k tworząc macierz

$$A^{(0)} = \begin{pmatrix} \alpha_2^1 & \cdots & \alpha_t^1 \\ \vdots & & \vdots \\ \alpha_2^m & \cdots & \alpha_t^m \end{pmatrix} \in \mathbb{R}^{m\times(t-1)}.$$

W tym celu wykonuje dla $j = 2, \ldots, t$ parę operacji

$$\begin{cases} A^{(1)}_{*,j-1} \leftarrow CX^{(1)}_{s-m+1:s,j-1} \\ X^{(1)}_{s-m+1:s,j} \leftarrow X^{(1)}_{s-m+1:s,j} + Y_{s-m+1:s,*}A^{(1)}_{*,j-1}. \end{cases} \tag{4.6}$$

Następnie P_0 wysyła m ostatnich składowych ostatniej kolumny macierzy $X^{(0)}$ (czyli liczby $X^{(0)}_{s-m+1:s,t}$) do P_1. Każdy inny proces P_i, $i = 1, \ldots, p-1$, odbiera m liczb od P_{i-1}, następnie wyznacza m ostatnich składowych macierzy $X^{(i)}$ oraz macierz współczynników α_j^k, to znaczy

$$A^{(i)} = \begin{pmatrix} \alpha^1_{it+1} & \cdots & \alpha^1_{(i+1)t} \\ \vdots & & \vdots \\ \alpha^m_{it+1} & \cdots & \alpha^m_{(i+1)t} \end{pmatrix} \in \mathbb{R}^{m\times t},$$

wykonując dla $j = 1, \ldots, t$ następującą parę operacji

$$\begin{cases} A^{(i)}_{*,j} \leftarrow CX^{(i)}_{s-m+1:s,j-1} \\ X^{(i)}_{s-m+1:s,j} \leftarrow X^{(i)}_{s-m+1:s,j} + Y_{s-m+1:s,*}A^{(i)}_{*,j} \end{cases} \tag{4.7}$$

gdzie $X^{(i)}_{s-m+1:s,0}$ oznacza liczby otrzymane od procesu P_{i-1}. Oczywiście, P_{p-1} wyznacza macierz

$$A^{(p-1)} = \begin{pmatrix} \alpha^1_{(p-1)t+1} & \cdots & \alpha^1_r \\ \vdots & & \vdots \\ \alpha^m_{(p-1)t+1} & \cdots & \alpha^m_r \end{pmatrix} \in \mathbb{R}^{m\times t'},$$

wykonując parę operacji (4.7) dla $j = 1, \ldots, t'$. Następnie każdy proces inny niż P_{p-1} wysyła m ostatnich składowych ostatniej kolumny macierzy $X^{(i)}$ do procesu P_{i+1}. W ostatnim (trzecim) kroku algorytmu każdy proces, wykorzystując macierze $A^{(i)}$ oraz Y, używa jednego wywołania operacji `GEMM` dla wyznaczenia $s-m$ pierwszych składowych kolumn swojej macierzy $X^{(i)}$. Zatem, proces P_0 wykonuje operację

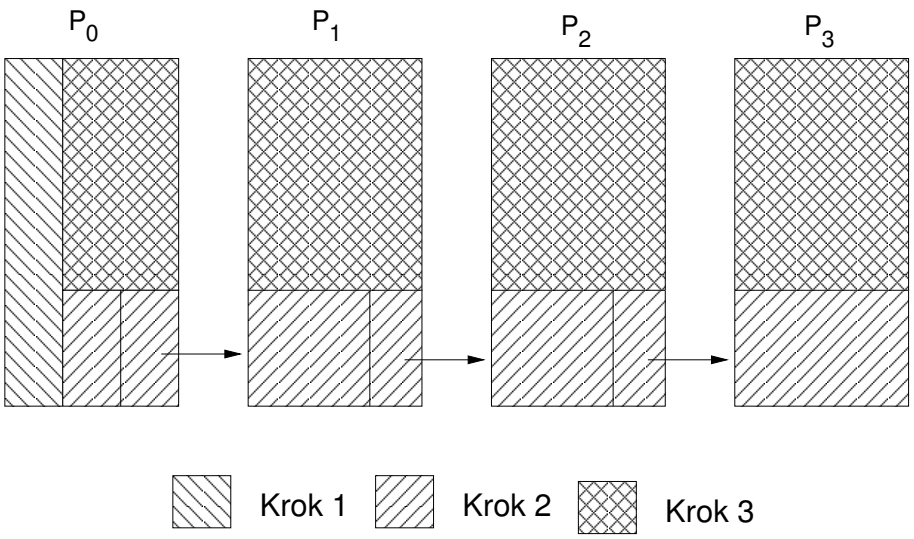
$$X^{(0)}_{1:s-m,2:t} \leftarrow X^{(0)}_{1:s-m,2:t} + Y_{1:s-m,*}A^{(0)}, \tag{4.8}$$

procesy P_i, $i = 1, \ldots, p-2$ operację

$$X^{(i)}_{1:s-m,1:t} \leftarrow X^{(i)}_{1:s-m,1:t} + Y_{1:s-m,*}A^{(i)}, \tag{4.9}$$

a proces P_{p-1} operację

$$X^{(p-1)}_{1:s-m,1:t'} \leftarrow X^{(p-1)}_{1:s-m,1:t'} + Y_{1:s-m,*}A^{(p-1)}. \tag{4.10}$$



Rys. 4.2. Wyznaczanie elementów rozwiązania w algorytmie 4.1. Każdy proces wyznacza blok kolumn macierzy X

Fig. 4.2. Finding elements of the solution using Algorithm 4.1. Each process finds a block of columns of X

Kolejność wyznaczania poszczególnych elementów rozwiązania oraz schemat komunikacji przestawiono na rysunku 4.2. Algorytm 4.1 stanowi wersję SPMD omówionego wyżej postępowania. Podobnie jak w przypadku algorytmów 3.1 i 3.2, zastosowanie 4.1 do rozwiązania problemu (1.28) dla dowolnych wartości n, przy dowolnych (ustalonych) wartościach r i s, wymaga przydzielenia współczynników $f_{rs+1}, \ldots, f_n$ ostatniemu procesowi, który po zakończeniu wykonywania czynności opisanych algorytmem 4.1 zastosuje algorytm 1.5 dla wyznaczenia pozostałych liczb $x_{rs+1}, \ldots, x_n$.

4.2.1. Analiza algorytmu

Przeprowadzimy teraz analizę czasu wykonania algorytmu 4.1 za pomocą modelu BSP, opisanego w punkcie 1.6.4. W tym celu dokonamy podziału operacji realizowanych w ramach algorytmu na superkroki.

Algorytm 4.1. Rozproszone wyznaczanie rozwiązania równania (1.28) przy $r > 1$, $s > 1$ takich, że $rs = n$ przez proces P_i, $i = 0, \ldots, p-1$.

Wejście: $a_1, \ldots, a_m$ oraz f_j tworzące macierz $F^{(i)}$ zdefiniowaną przez (4.3) albo (4.4).

Wyjście: macierz $X^{(i)}$ zdefiniowana przez (4.1) albo (4.2).

1: $X \leftarrow (F^{(i)}, \mathbf{e}_1)$; $C \leftarrow \begin{pmatrix} a_m & \cdots & a_1 \\ & \ddots & \vdots \\ 0 & & a_m \end{pmatrix}$; $t \leftarrow \lfloor r/p \rfloor$

2: **if** $i = p-1$ **then**

3: $\quad t \leftarrow r - (p-1)t$

4: **end if**

5: **for** $k = 2$ to s **do**

6: $\quad X^{(i)}_{k,*} \leftarrow X^{(i)}_{k,*} + C_{1,\max\{1,m-k+2\}:m} X^{(i)}_{\max\{1,k-m\}:k-1,*}$ {operacja `GEMV`}

7: **end for**

8: **if** $i = 0$ **then**

9: $\quad$ **for** $j = 2$ to t **do**

10: $\quad\quad A^{(0)}_{*,j-1} \leftarrow C X^{(0)}_{s-m+1:s,j-1}$ {operacja `TRMV`}

11: $\quad\quad X^{(0)}_{s-m+1:s,j} \leftarrow X^{(0)}_{s-m+1:s,j} + Y_{s-m+1:s,*} A^{(0)}_{*,j-1}$ {operacja `GEMV`}

12: $\quad$ **end for**

13: $\quad$ **send** $X^{(0)}_{s-m+1:s,t}$ **to** P_1

14: **else**

15: $\quad$ **receive** $X^{(i)}_{s-m+1:s,0}$ **from** P_{i-1}

16: $\quad$ **for** $j = 1$ to t **do**

17: $\quad\quad A^{(i)}_{*,j-1} \leftarrow C X^{(i)}_{s-m+1:s,j-1}$ {operacja `TRMV`}

18: $\quad\quad X^{(i)}_{s-m+1:s,j} \leftarrow X^{(i)}_{s-m+1:s,j} + Y_{s-m+1:s,*} A^{(i)}_{*,j-1}$ {operacja `GEMV`}

19: $\quad$ **end for**

20: $\quad$ **if** $i \neq p-1$ **then**

21: $\quad\quad$ **send** $X^{(i)}_{s-m+1:s,t}$ **to** P_{i+1}

22: $\quad$ **end if**

23: **end if**

24: $Y \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_m)$ {stosując (2.18)}

25: **if** $i = 0$ **then**

26: $\quad X^{(0)}_{1:s-m,2:t} \leftarrow X^{(0)}_{1:s-m,2:t} + Y_{1:s-m,*} A^{(0)}$ { `GEMM`}

27: **else**

28: $\quad X^{(i)}_{1:s-m,1:t} \leftarrow X^{(i)}_{1:s-m,1:t} + Y_{1:s-m,*} A^{(i)}$ { `GEMM`}

29: **end if**

Poniżej przedstawiamy łączny koszt algorytmu. Pierwszy superkrok stanowią instrukcje 5–13 (w przypadku pierwszego procesu) oraz instrukcje 5–7 dla pozostałych procesów. Łączny koszt opisują wzory (4.11) i (4.12). Następne $p-2$ superkroki to instrukcje 15–22 powtarzane przez procesy $P_1, \ldots, P_{p-2}$, łącznie (4.13). Ostatni superkrok, to instrukcje 16–19 i 25–29 wykonywane przez proces P_{p-1} oraz instrukcje 25–29 wykonywane przez pozostałe procesy, co daje koszt (4.14). Po każdym superkroku wykonywana jest operacja synchronizacji o koszcie l. Każda operacja **send** wysyła m liczb, stąd jej koszt wynosi mg. Liczby operacji arytmetycznych w poszczególnych superkrokach obliczamy podobnie jak w dowodzie twierdzenia 3.1.

$$C_{p,r,s}(n,m) = 2m\left(s - \frac{m+1}{2}\right)\left(\frac{r}{p} + 1\right) \tag{4.11}$$

$$+ \; 3m^2\frac{r}{p} + mg + l \tag{4.12}$$

$$+ \; (p-2)\left(3m^2\frac{r}{p} + mg + l\right) \tag{4.13}$$

$$+ \; 3m^2\frac{r}{p} + 2(s-m)m\left(\frac{r}{p}\right) + l. \tag{4.14}$$

Aby wyznaczyć optymalną wartość parametru s, zminimalizujemy koszt opisany łącznie wzorami (4.11), (4.12), (4.13) oraz (4.14). Stąd otrzymujemy optymalną wartość parametru

$$s^* = \left\lfloor \sqrt{\frac{n(3mp - 3m - 1)}{2p}} \right\rfloor, \tag{4.15}$$

a zatem prawdziwy jest następujący wniosek.

Wniosek 4.1. *Dla modelu BSP optymalny wybór wartości parametru s w algorytmie 4.1 zależy wyłącznie od rozmiaru problemu n, m oraz liczby procesorów p.*

Powyższy wniosek oznacza, że wybór teoretycznej optymalnej wartości parametru s nie zależy od wartości g i l, które charakteryzują konkretną maszynę, jest zatem uniwersalny. Zauważmy również, że podobnie jak w przypadku algorytmu blokowego 3.1, wartość parametru s dana przez (4.15) może być skorygowana za pomocą wzoru (3.14). Wybór wartości parametru r realizujemy wówczas przez (3.15).

4.2.2. Wyniki eksperymentów

Algorytm 4.1 został zaimplementowany przy użyciu standardu MPI oraz uruchomiony na trzech platformach sprzętowych: klastrze komputerów z procesorami Intel Pentium III, klastrze dwudziestu czterech procesorów Itanium 2 oraz komputerze Cray X1, dla różnych rozmiarów problemu n i m, różnych wartości parametru s oraz dla wyznaczenia rozwiązania pełnego (F) oraz cząstkowego (P).

Tabela 4.1
Optymalne wartości s dla $n = 1000000$

m	$p = 4$	$p = 8$	$p = 15$
64	8477	9161	9464
16	4227	4575	4729
4	2091	2277	2359

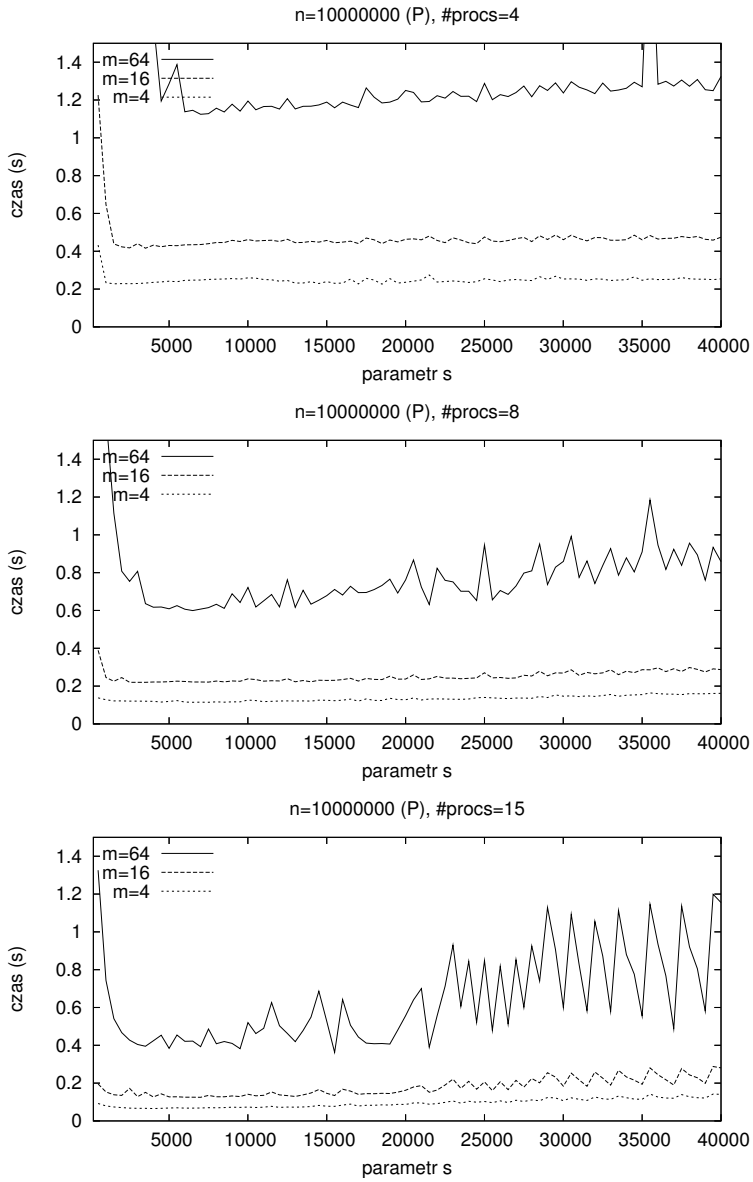
Klastry komputerów z procesorami Intela

Rysunek 4.3 pokazuje czas działania algorytmu 4.1 na klastrze Pentium III dla różnych wartości parametru s oraz różnej liczby procesorów. Optymalne, obliczone ze wzoru (4.15) wartości parametru s dla $n = 1000000$ i różnych m i p prezentuje tabela 4.1. W przypadku obu klastrów algorytm osiąga najlepszy czas dla wartości s równej w przybliżeniu $s^*/2$. Ten wybór pozostaje słuszny zarówno dla wyznaczania rozwiązania pełnego, jak i cząstkowego. Wynika to stąd, że szybkość działania algorytmu zależy od szybkości działania wywoływanych podprogramów z biblioteki BLAS, a to nie jest bezpośrednio uwzględniane w modelu BSP. W przypadku gdy $s = s^*$, macierze, na których operują podprogramy z biblioteki BLAS, są „zbyt wąskie", to znaczy charakteryzują się niewielką liczbą kolumn w stosunku do liczby wierszy. Powoduje to mniej efektywne wykorzystanie pamięci podręcznej i w konsekwencji wolniejsze działanie podprogramów. Podobna sytuacja ma miejsce również na platformie Itanium 2. Stąd ostatecznie w przypadku klastrów opartych na procesorach Intela, wybór parametru s określa następujący wzór

$$s = \begin{cases} \lfloor s^*/2 \rfloor - 1 & \text{dla } \lfloor s^*/2 \rfloor \text{ parzystych,} \\ \lfloor s^*/2 \rfloor & \text{w przeciwnym przypadku.} \end{cases} \tag{4.16}$$

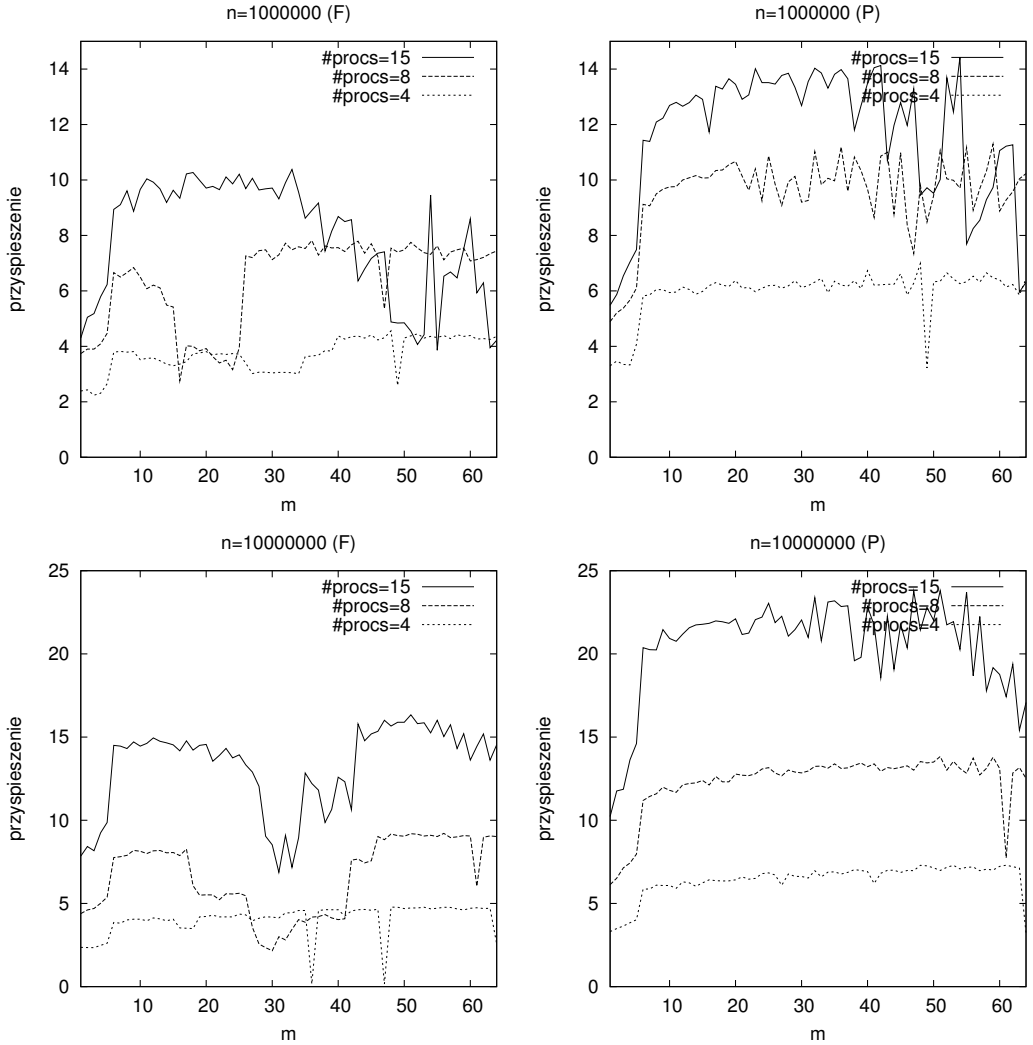
Należy jednak podkreślić, że zmiana wartości s określonej przez (4.15) o $\pm 10\%$ na ogół wpływa nieznacznie na czas działania algorytmu, stąd należy tę wartość traktować jako wstępny dobór i w wyniku eksperymentów można ją ewentualnie zmienić tak, aby uzyskać lepszą szybkość działania algorytmu.

Rysunki 4.4 oraz 4.5 pokazują odpowiednio przyspieszenie oraz wydajność dla algorytmu 4.1, dla różnych rozmiarów problemu n i m, przy wartości s określonej wzorem (4.16), uzyskane na klastrze Pentium III. Możemy zaobserwować, że wydajność wykonania algorytmu 4.1 rośnie dla większych wartości n i m. Jednak dla mniejszych wartości n wskazane jest użycie mniejszej liczby procesorów. Dla piętnastu procesorów przy wyznaczaniu rozwiązania pełnego osiągnięto wydajność około 4500 Mflops, a dla rozwiązania częściowego 3500 Mflops, co jest porównywalne z wydajnością czterech procesorów komputera Cray SV1 dla algorytmu 3.2.



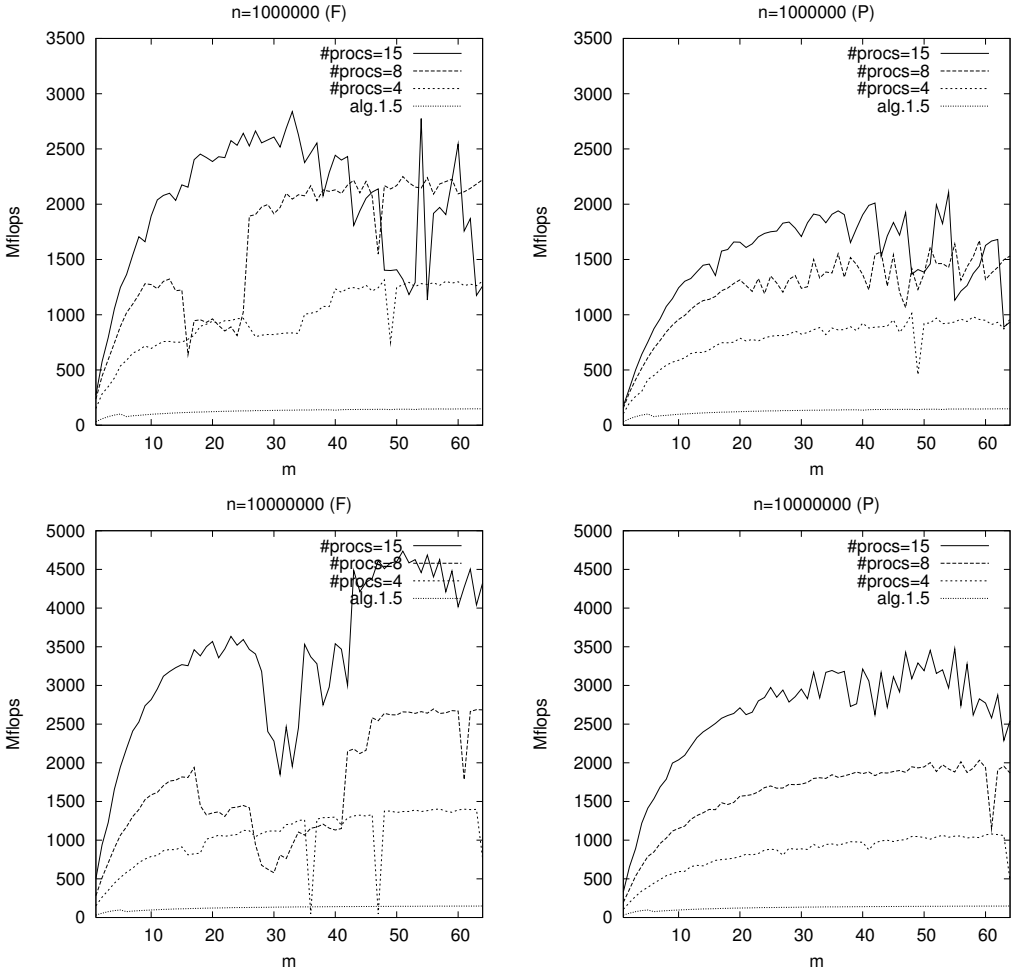
Rys. 4.3. Czas wykonania algorytmu 4.1 na klastrze Pentium III dla różnych wartości s

Fig. 4.3. Execution time of Algorithm 4.1 on a cluster of Pentium III for various values of s



Rys. 4.4. Przyspieszenie algorytmu 4.1 względem algorytmu 1.5 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P) na klastrze Pentium III

Fig. 4.4. Speedup of Algorithm 4.1 relative to Algorithm 1.5 for finding full (F) and partial (P) solution on a cluster of Pentium III



Rys. 4.5. Wydajność klastra Pentium III dla algorytmów 4.1 i 1.5 wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 4.5. Performance of the algorithms 4.1 and 1.5 for finding full (F) and partial (P) solution on a cluster of Pentium III

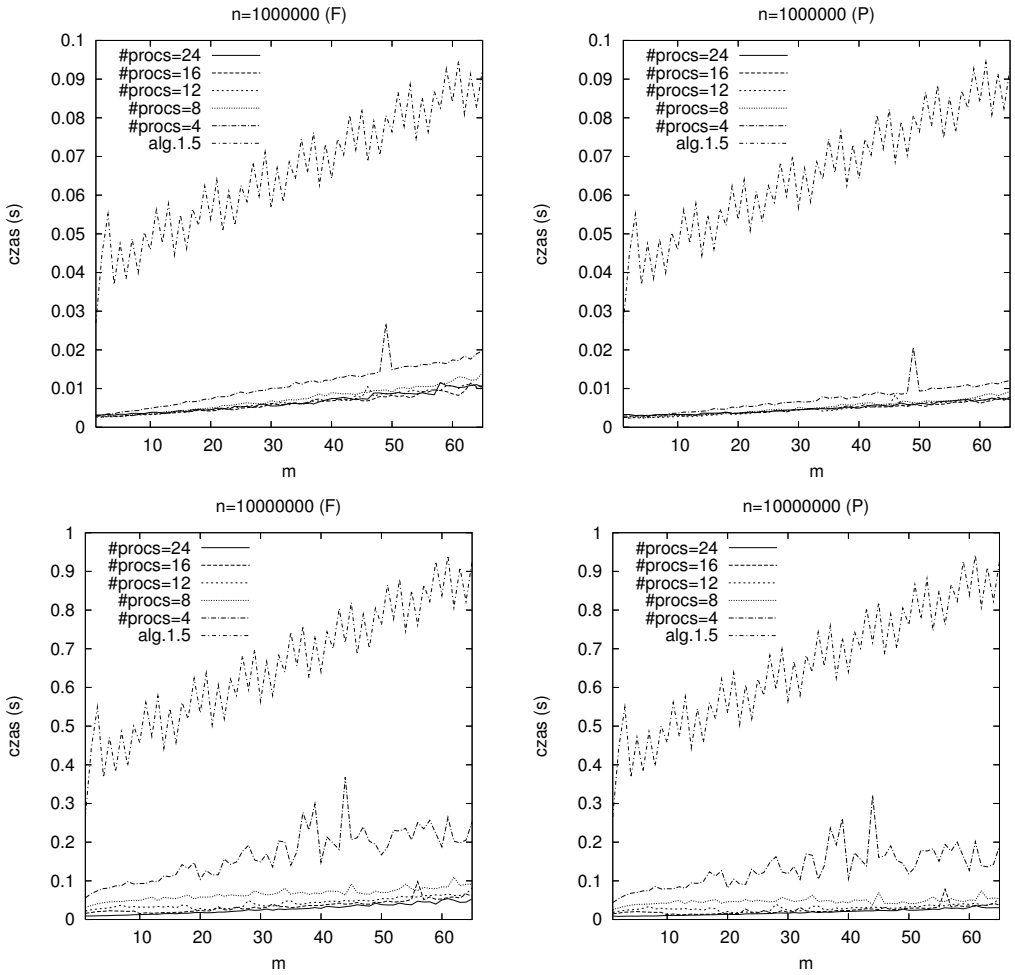
Podobną sytuację możemy zaobserwować na klastrze procesorów Itanium (rysunki 4.6, 4.7 oraz 4.8). Wydajność dla algorytmu 4.1 rośnie wraz ze wzrostem rozmiaru problemu. Jednocześnie przyspieszenie spada, gdyż podobnie jak w przypadku algorytmu 1.5 wykonywanego na komputerach wektorowych, wraz ze wzrostem wartości m rośnie wydajność jego wykonania. Dla odpowiednio dużych rozmiarów problemu, dla algorytmu 4.1 osiągana jest wydajność do 60 Gflops, co stanowi do 50% teoretycznej wydajności klastra. Różnica w wydajności dla algorytmu przy wyznaczaniu rozwiązania pełnego oraz cząstkowego wynika z tego, że w przypadku rozwiązania pełnego wykonywane jest dodatkowo mnożenie macierzy (operacja `GEMM` z trzeciego poziomu biblioteki BLAS), realizowane przy znacznie większej wydajności niż wcześniejsze dwa kroki algorytmu.

Cray X1

W przypadku tego komputera eksperymenty wykazały, że wybór optymalnej wartości parametru s nie zależy od wartości m oraz liczby procesorów, a największa szybkość jest osiągnięta dla wartości $s \approx \lfloor\sqrt{2n}\rfloor$ (rysunek 4.9). Zatem, za optymalny należy przyjąć wybór określony wzorem (2.26).

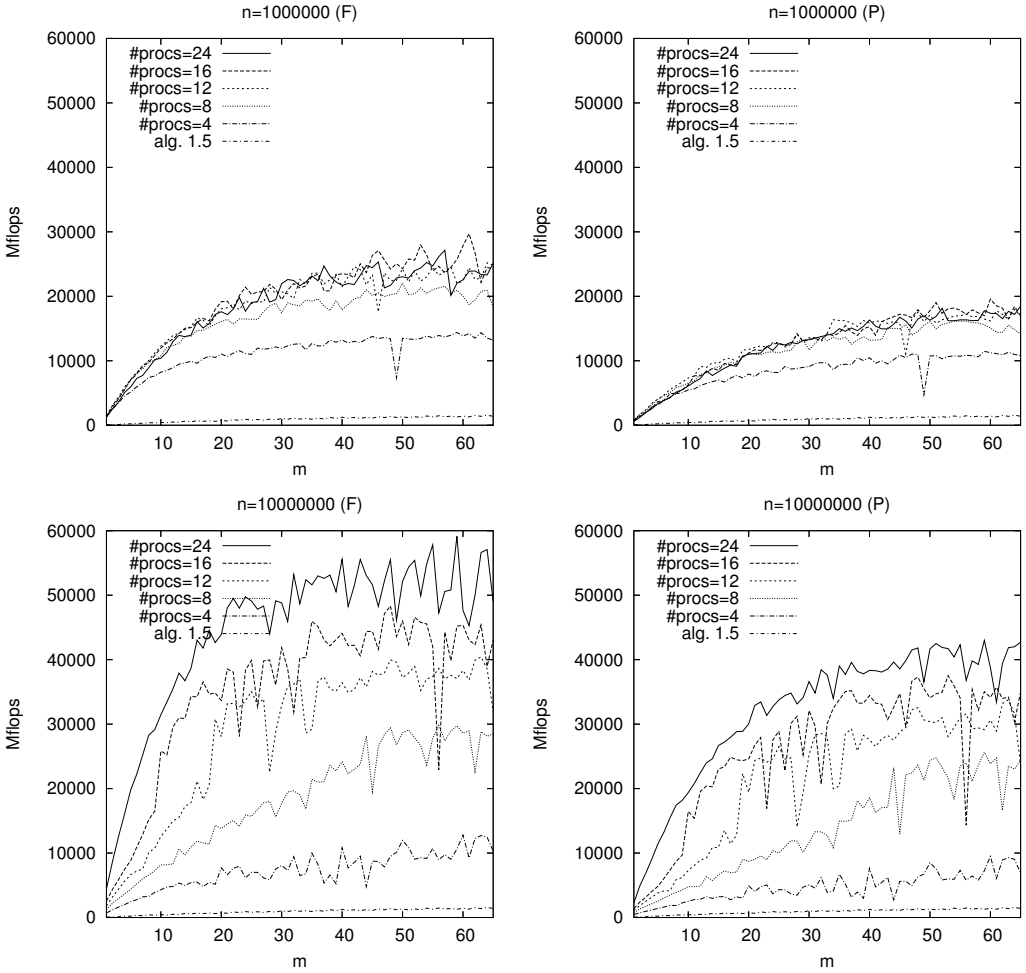
Rysunki 4.10, 4.11 i 4.12 pokazują odpowiednio czas działania, wydajność oraz przyspieszenie algorytmu 4.1 względem 1.5 na komputerze Cray X1 dla różnych wartości n i m przy wartości parametru s określonego wzorem (2.26). Sytuacja jest podobna do wyników działania algorytmu na klastrze Itanium 2, przy czym przyspieszenie algorytmu jest znacznie większe, co wynika z bardzo słabego wykorzystania architektury komputera przez algorytm 1.5. Rysunek 4.13 pokazuje przyspieszenie i wydajność dla bardzo dużej wartości n. Dla odpowiednio dużych wartości m algorytm osiągnięto wydajność 50 Gflops, co stanowi około 50% maksymalnej wydajności ośmiu procesorów MSP.

Zarówno w przypadku klastra komputerów z procesorami Itanium 2, jak i ośmiu procesorów MSP komputera Cray X1 osiągnięto wykorzystanie mocy obliczeniowej na poziomie 50%. Wyniki eksperymentów potwierdziły bardzo dobrą skalowalność metody. Dodatkowo w przypadku uruchomienia na jednym procesorze, algorytm 4.1 redukuje się do algorytmu 3.1 i pojedynczy procesor MSP komputera Cray X1 osiąga dla niego wydajność 10 Gflops, co stanowi około 75% mocy tego procesora. Trzeba również zaznaczyć, że klasyczna metoda Wanga dla zagadnienia (1.28) zaimplementowana i uruchomiona na klastrze dziesięciu procesorów daje tylko około trzykrotne przyspieszenie w stosunku do wykonywanego w środowisku sekwencyjnym algorytmu 1.5, wykorzystując zaledwie kilka procent mocy obliczeniowej klastra [108].



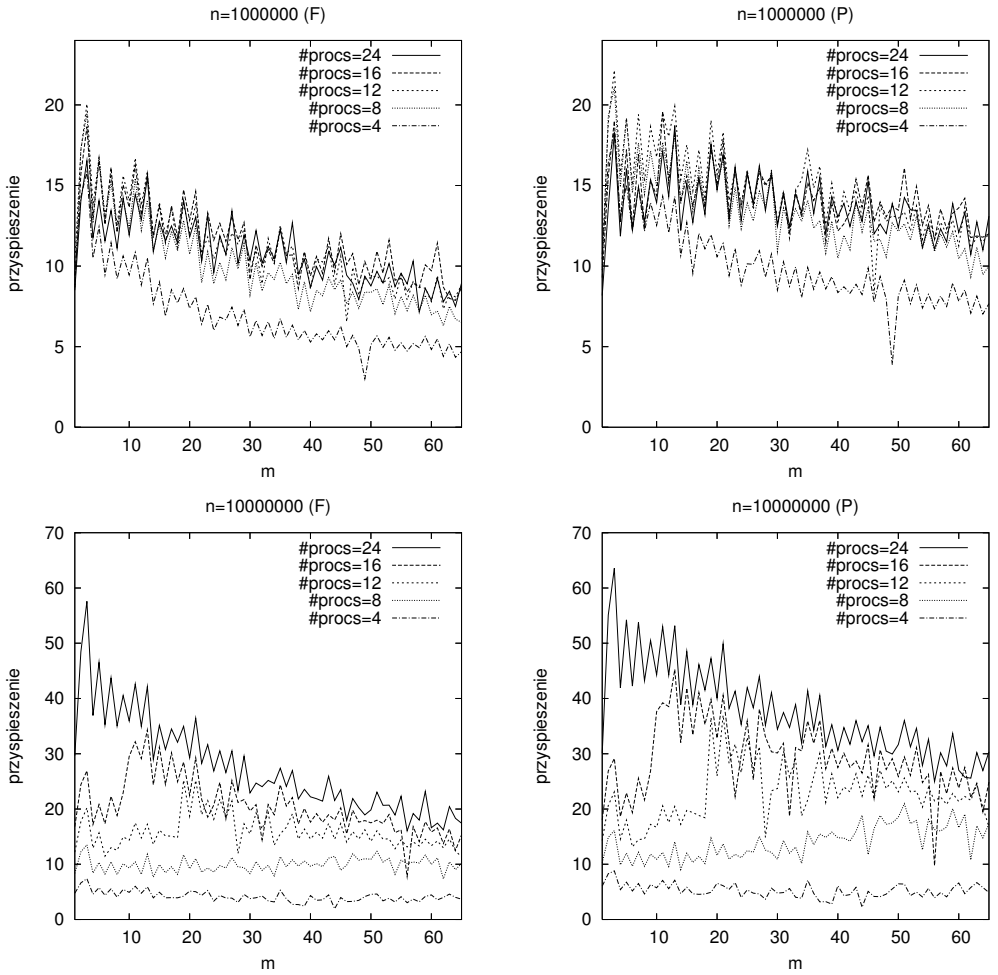
Rys. 4.6. Czas wykonania algorytmów 4.1 i 1.5 na klastrze Itanium 2 przy wyznaczeniu rozwiązania pełnego (F) i cząstkowego (P)

Fig. 4.6. Execution time of the algorithms 3.1 i 1.5 for finding full (F) and partial (P) solution on a cluster of Itanium 2



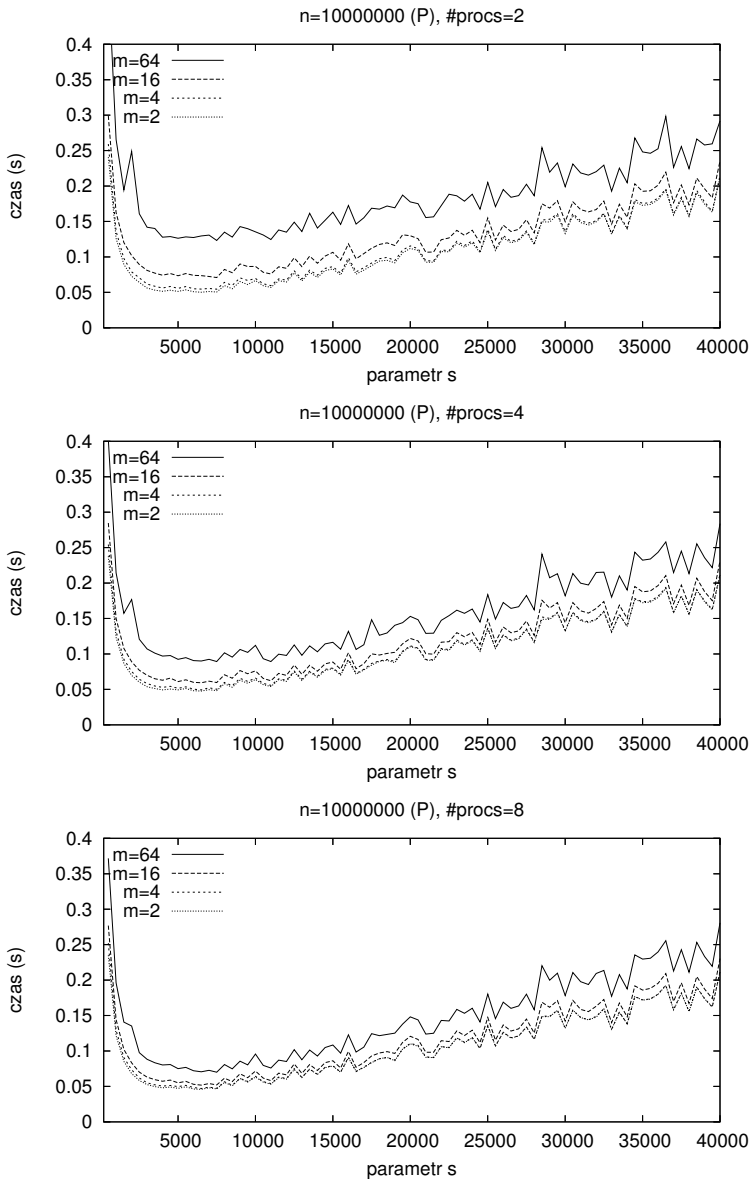
Rys. 4.7. Wydajność klastra Itanium 2 dla algorytmów 4.1 i 1.5 przy wyznaczeniu rozwiązania pełnego (F) i cząstkowego (P)

Fig. 4.7. Performance of the algorithms 4.1 and 1.5 for finding full (F) and partial (P) solution on a cluster of Itanium 2



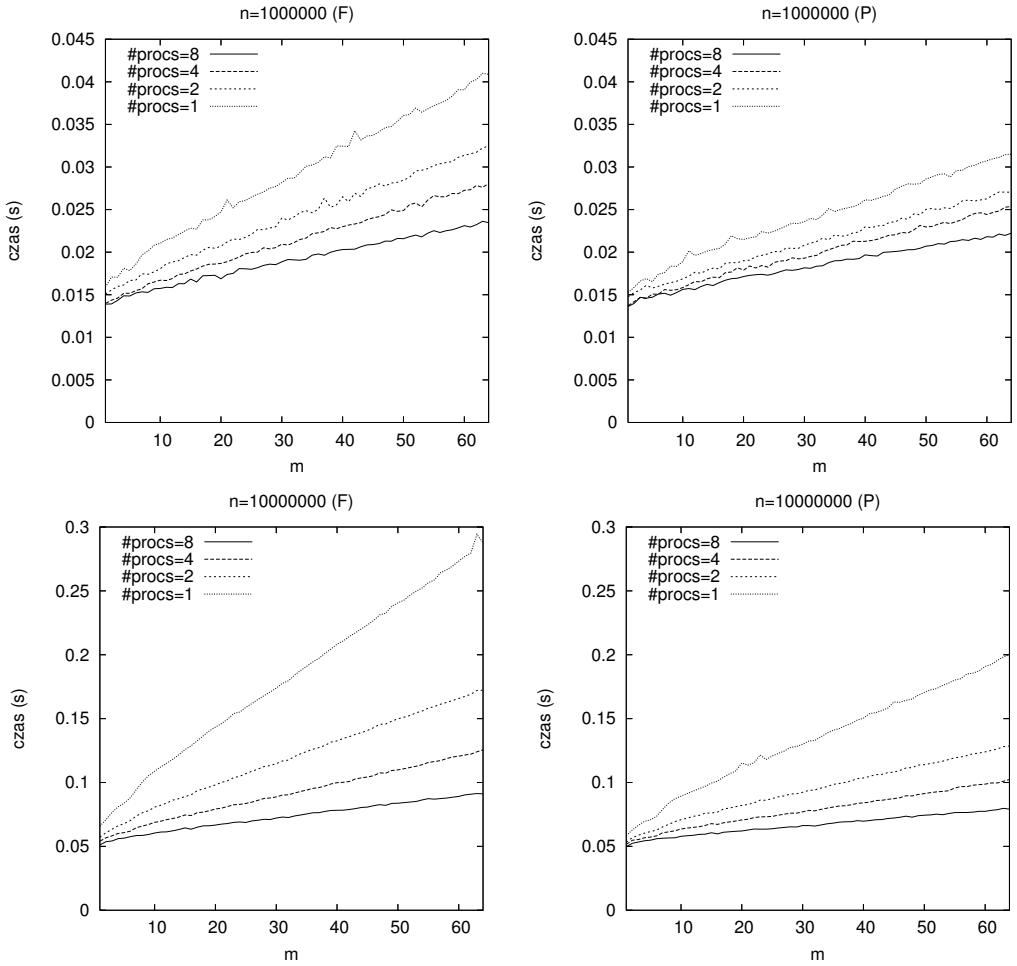
Rys. 4.8. Przyspieszenie algorytmu 4.1 względem algorytmu 1.5 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P) na klastrze Itanium 2

Fig. 4.8. Speedup of Algorithm 4.1 relative to Algorithm 1.5 for finding full (F) and partial (P) solution on a cluster of Itanium 2



Rys. 4.9. Czas wykonania algorytmu 4.1 na komputerze Cray X1 dla różnych wartości s. Optymalna wartość $s \approx 4471$

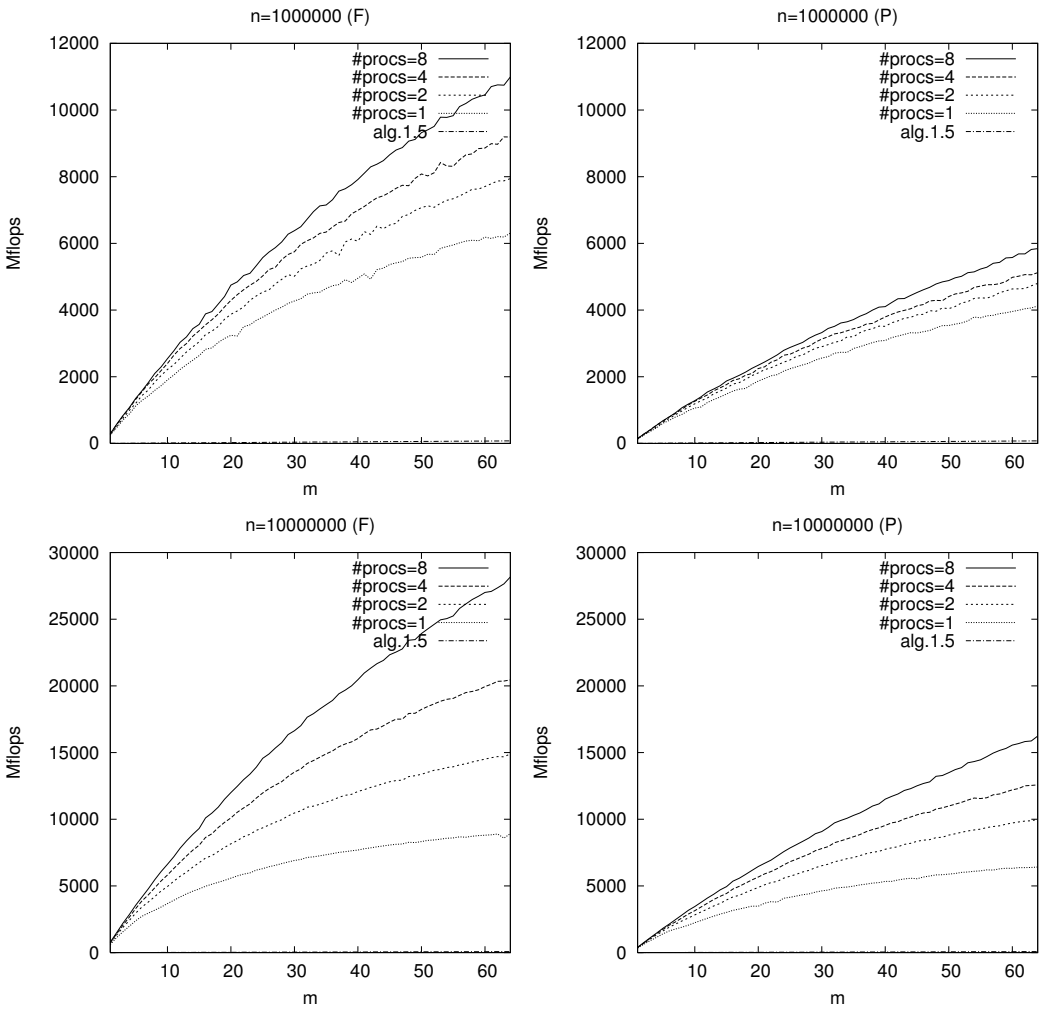
Fig. 4.9. Execution time of Algorithm 4.1 on a Cray X1 for various s. The optimal value is $s \approx 4471$



Rys. 4.10. Czas wykonania algorytmów 4.1 (#procs=2, 4, 8) i 3.1 (#procs=1) na komputerze Cray X1 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

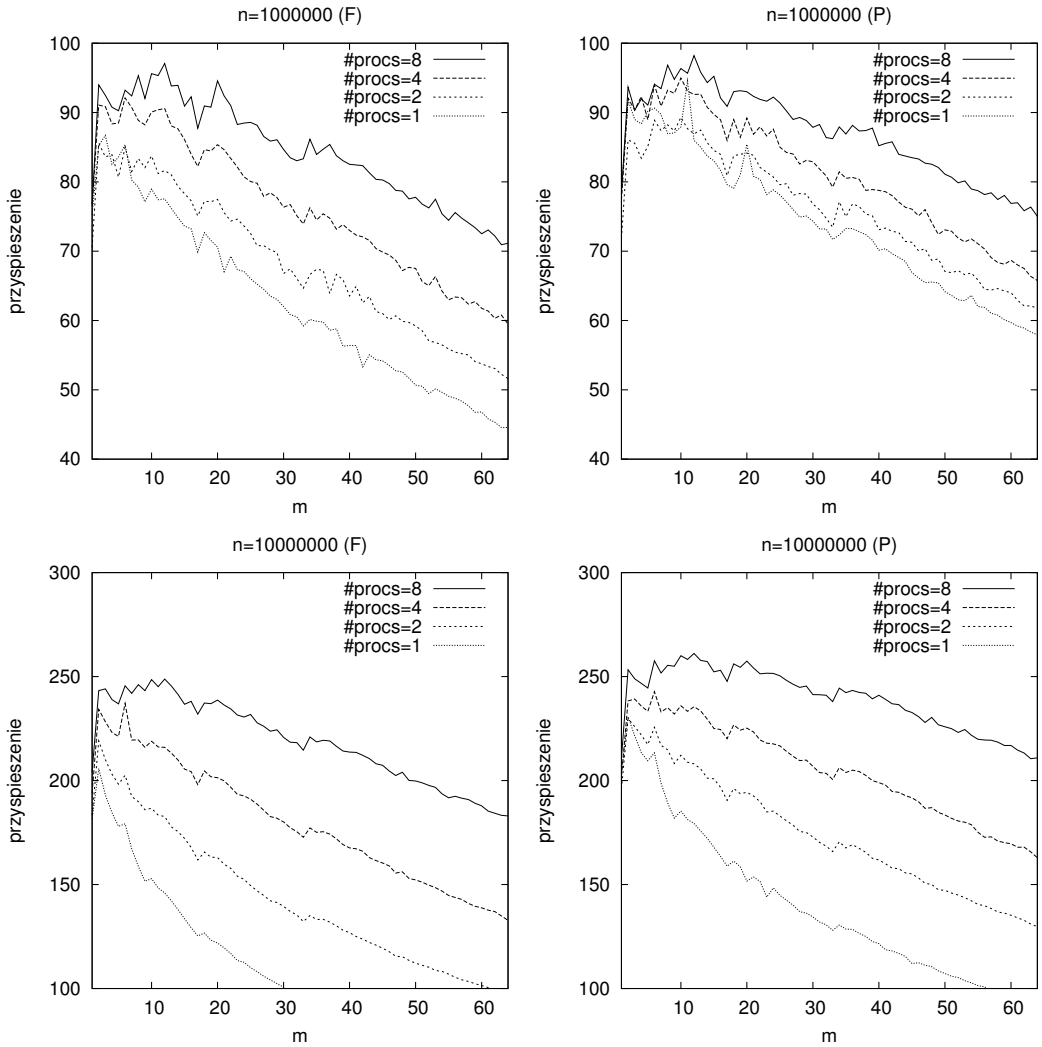
Fig. 4.10. Execution time of the algorithms 4.1 (#procs=2, 4, 8) and 3.1 (#procs=1) for finding full (F) and partial (P) solution on a Cray X1

Na zakończenie tego rozdziału podkreślmy, że dzięki wykorzystaniu operacji z wyższych poziomów biblioteki BLAS możliwe było sformułowanie bardzo efektywnych, skalowalnych algorytmów rozwiązywania problemu (1.28), charakteryzujących się znacznie lepszym przyspieszeniem niż inne opisane w literaturze [9]. Podobnie jak w przypadku algorytmów 2.1 i 3.2 można podać wstępne, niezależne od charakterystyki konkretnego komputera, przybliżenie optymalnych parametrów metody.



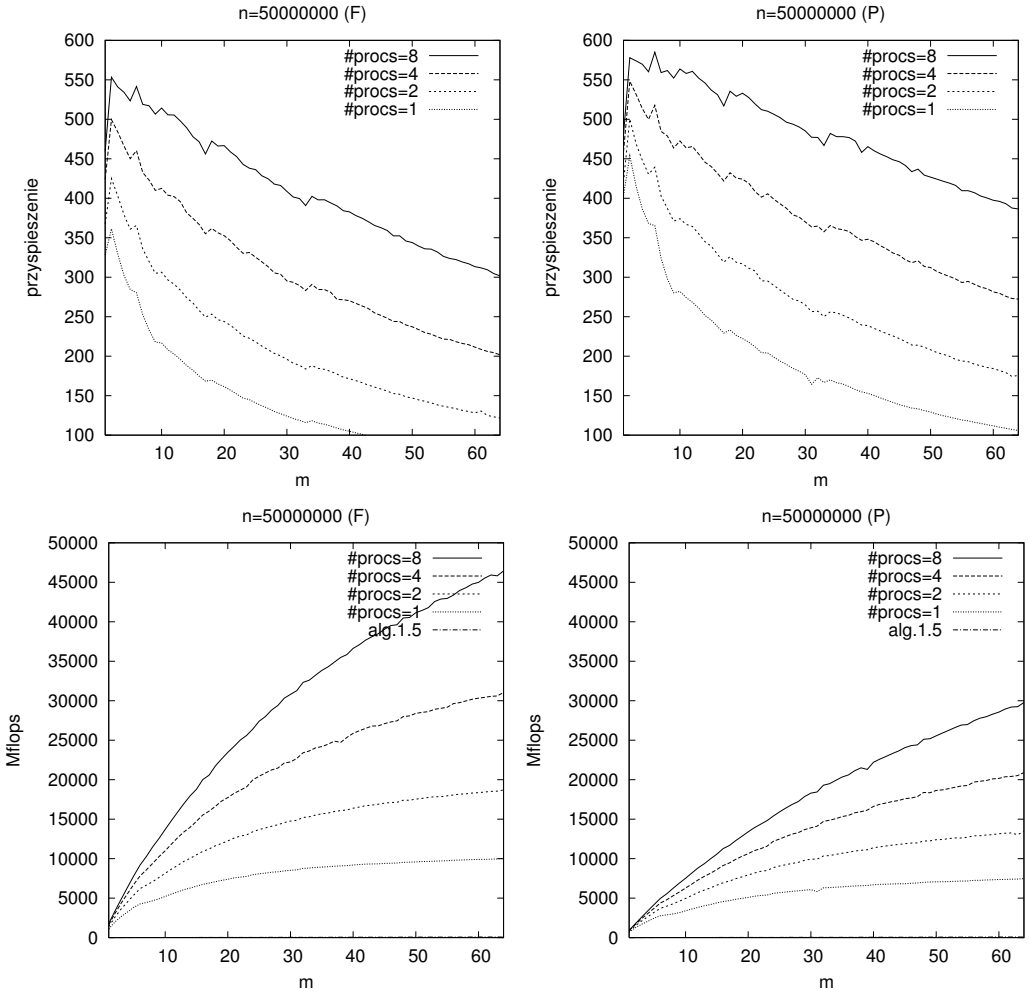
Rys. 4.11. Wydajność komputera Cray X1 dla algorytmów 4.1 (#procs=2, 4, 8), 3.1 (#procs=1) i 1.5 wyznaczenia rozwiązania pełnego (F) i cząstkowego (P)

Fig. 4.11. Performance of the algorithms 4.1 (#procs=2, 4, 8), 3.1 (#procs=1) and 1.5 for finding full (F) and partial (P) solution on a Cray X1



Rys. 4.12. Przyspieszenie algorytmów 4.1 (#procs=2, 4, 8) i 3.1 (#procs=1) względem algorytmu 1.5 dla wyznaczenia rozwiązania pełnego (F) i cząstkowego (P) na komputerze Cray X1

Fig. 4.12. Speedup of the algorithms 4.1 (#procs=2, 4, 8) and 3.1 (#procs=1) relative to Algorithm 1.5 for finding full (F) and partial (P) solution on a Cray X1



Rys. 4.13. Przyspieszenie algorytmów 4.1 (#procs=2, 4, 8) i 3.1 (#procs=1) względem algorytmu 1.5 i wydajność na komputerze Cray X1 dla bardzo dużych rozmiarów problemu

Fig. 4.13. Speedup of the algorithms 4.1 (#procs=2, 4, 8) and 3.1 (#procs=1) relative to Algorithm 1.5 for big problem sizes on a Cray X1

5. Liniowe filtry rekurencyjne

W niniejszym rozdziale zajmiemy się problemem wyznaczania wartości liniowego filtra rekurencyjnego, który ma duże zastosowanie w dziedzinie przetwarzania sygnałów [101]. Dla zadanego ciągu liczb rzeczywistych

$$x_1, x_2, \ldots, x_n,$$

zwanego ciągiem sygnałów wejściowych należy wyznaczyć liczby rzeczywiste

$$y_1, y_2, \ldots, y_n,$$

spełniające równość

$$y_k = \sum_{j=1}^{m} b_j y_{k-j} + \sum_{j=0}^{m} a_j x_{k-j}, \tag{5.1}$$

gdzie m jest niewielką, parzystą liczbą całkowitą, $x_k = 0$, $y_k = 0$ dla $k \leq 0$, a_j, b_j zaś są dane [101]. Znane są implementacje problemu (5.1) na dedykowanych procesorach potokowych [94] oraz wieloprocesorowych komputerach typu SIMD [97]. Naszym celem będzie sformułowanie algorytmu rozproszonego wyznaczania (5.1), wykorzystującego poziomy drugi i trzeci biblioteki BLAS. W tym celu zauważmy, że wzór (5.1) może być zapisany jako

$$y_k = f_k + \sum_{j=1}^{m} b_j y_{k-j}. \tag{5.2}$$

gdzie

$$f_k = \sum_{j=0}^{m} a_j x_{k-j}. \tag{5.3}$$

Zatem, algorytm może działać w dwóch etapach: najpierw obliczane są wielkości f_k, a następnie wyznaczane są liczby y_k według (5.2). Oczywiście, (5.2) jest o postaci (1.28), zatem jeśli znane są już wartości f_k, wówczas do wyznaczenia liczb y_k mogą posłużyć omówione w poprzednich rozdziałach algorytmy 2.1, 3.1, 3.2 oraz rozproszony 4.1. Aby w procesie wyznaczenia (5.1) była dobrze wykorzystana moc obliczeniowa, obliczenie wartości f_k powinno być realizowane w sposób równoległy przy wykorzystaniu wyższych poziomów biblioteki BLAS.

5.1. Algorytm mnożenia dolnotrójkątnej pasmowej macierzy Toeplitza przez wektor

Zauważmy, że wyznaczenie liczb f_k według wzoru (5.3) może być zapisane w postaci następującej operacji mnożenia macierzy przez wektor:

$$\begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix} \leftarrow \begin{pmatrix} a_0 & & & & & \\ a_1 & \ddots & & & & \\ \vdots & \ddots & \ddots & & & \\ a_m & \dots & a_1 & a_0 & & \\ & \ddots & & \ddots & \ddots & \\ & & a_m & \cdots & a_1 & a_0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}. \tag{5.4}$$

Macierz ma postać dolnotrójkątną, pasmową i jest typu Toeplitza, czyli jej elementy na poszczególnych przekątnych są stałe. Wybierzmy dwie dodatnie liczby całkowite r i s takie, że $rs \leq n$. Zastosujemy następujący blokowy algorytm dla wyznaczenia liczb $f_1, \dots, f_{rs}$ oraz (5.3) dla wyznaczenia brakujących $f_{rs+1}, \dots, f_n$. W tym celu zdefiniujmy wektory

$$\mathbf{x}_j = (x_{(j-1)s+1}, \dots, x_{js})^T, \mathbf{f}_j = (f_{(j-1)s+1}, \dots, f_{js})^T \in \mathbb{R}^s, \tag{5.5}$$

dla $j = 1, \dots, r$ oraz macierze

$$L = \begin{pmatrix} a_0 & & & & \\ \vdots & \ddots & & & \\ a_m & \dots & a_0 & & \\ & \ddots & & \ddots & \\ & & a_m & \cdots & a_0 \end{pmatrix} \in \mathbb{R}^{s \times s},$$

$$U = \begin{pmatrix} & a_m & \cdots & a_1 \\ & & \ddots & \vdots \\ & & & a_m \\ 0 & & & \end{pmatrix} \in \mathbb{R}^{s \times s}.$$

Wówczas zachodzi $\mathbf{f}_1 = L\mathbf{x}_1$, pozostałe zaś wektory $\mathbf{f}_j$, $j = 2, \dots, r$, spełniają równość

$$\mathbf{f}_j = L\mathbf{x}_j + U\mathbf{x}_{j-1}.$$

Jeśli teraz zdefiniujemy macierze

$$F = (\mathbf{f}_1, \dots, \mathbf{f}_r), \; X = (\mathbf{x}_1, \dots, \mathbf{x}_r) \in \mathbb{R}^{s \times r} \tag{5.6}$$

oraz

$$G = \begin{pmatrix} a_m & a_{m-1} & \cdots & a_1 \\ 0 & a_m & \cdots & a_2 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & a_m \end{pmatrix} \in \mathbb{R}^{m\times m},$$

to możemy wyznaczyć wektory $\mathbf{f}_j$ w następujący sposób. Po pierwsze, należy wykonać operację $F \leftarrow LX$, a następnie dodać do m pierwszych składowych każdego wektora $\mathbf{f}_{j+1}$ wektor $GX_{s-m+1:s,j}$, gdzie $j = 1, \ldots, r-1$. Istotnie, biorąc pod uwagę postać macierzy U, można zaobserwować, że wektor $U\mathbf{x}_{j-1}$ będzie miał tylko m pierwszych składowych różnych od zera oraz faktycznie na wynik będzie miało wpływ jedynie m ostatnich składowych wektora $\mathbf{x}_{j-1}$. Otrzymamy w ten sposób algorytm 5.1.

Algorytm 5.1. Wyznaczanie współczynników f_k, $k = 1, \ldots, rs$, zdefiniowanych przez (5.3) przy ustalonych wartościach całkowitych $r > 1$, $s > 1$.

Wejście: liczby $a_0, \ldots, a_m$ oraz $x_1, \ldots, x_{rs}$ tworzące wektory $\mathbf{x}_1, \ldots, \mathbf{x}_r$ zdefiniowane przez (5.5)

Wyjście: $F_{1:s,1:r} = (\mathbf{f}_1, \ldots \mathbf{f}_r)$.

1: $G \leftarrow \begin{pmatrix} a_m & \cdots & a_1 \\ & \ddots & \vdots \\ 0 & & a_m \end{pmatrix}$

2: **for** $k = 1$ to s **do**

3: $\quad F_{k,*} \leftarrow (a_{\min\{k-1,m\}}, \ldots, a_1, a_0) \cdot X_{\max\{1:k-m\}:k,*}$ {operacja `GEMV`}

4: **end for**

5: $X_{s-m+1:s,1:r-1} \leftarrow GX_{s-m+1:s,1:r-1}$ {operacja `TRMM`}

6: **for** $k = 2$ to r **do**

7: $\quad F^{(i)}_{1:m,k} \leftarrow F^{(i)}_{1:m,k} + X_{s-m+1:s,k-1}$ {operacja `AXPY`}

8: **end for**

Instrukcje 2–4 mogą być zaimplementowane jako ciąg s wywołań operacji `GEMV` o postaci

$$\mathbf{y} \leftarrow \beta\mathbf{y} + \alpha A^T\mathbf{x},$$

gdzie $\beta = 0$ i $\alpha = 1$. Liczba działań zmiennopozycyjnych w k-tym wywołaniu wynosi

$$2r \max\{k, m+1\} \tag{5.7}$$

Instrukcja 5, to operacja `TRMM` (mnożenie macierzy trójkątnej przez macierz pełną), która wymaga $(r-1)m^2$ działań plus $(r-1)m$ działań potrzebnych dla dodania $r-1$ wektorów o liczbie składowych m przy użyciu operacji `AXPY` (instrukcje 6-8).

Algorytm 5.1 może być zaimplementowany na komputerach z pamięcią rozproszoną i klastrach przy użyciu MPI. Analogicznie jak w przypadku algorytmu 4.1, każdy proces $P_0, \ldots, P_{p-2}$ wyznacza $t = \lfloor r/p \rfloor$ kolumn macierzy F, a P_{p-1} wyznacza $t' = r - t(p-1)$ kolumn. Zdefiniujmy macierze opisujące dane przetwarzane w pamięciach lokalnych, a zatem dla $i = 0, \ldots, p-2$

$$X^{(i)} = (\mathbf{x}_{it+1}, \ldots, \mathbf{x}_{(i+1)t}), \tag{5.8}$$

$$F^{(i)} = (\mathbf{f}_{it+1}, \ldots, \mathbf{f}_{(i+1)t}) \in \mathbb{R}^{s \times t} \tag{5.9}$$

oraz

$$X^{(p-1)} = (\mathbf{x}_{(p-1)t+1}, \ldots, \mathbf{x}_r), \tag{5.10}$$

$$F^{(p-1)} = (\mathbf{f}_{(p-1)t+1}, \ldots, \mathbf{f}_r) \in \mathbb{R}^{s \times t'}. \tag{5.11}$$

Zanim rozpoczną się obliczenia, ma miejsce komunikacja, a dokładniej każdy proces P_i z wyjątkiem P_{p-1} wysyła m ostatnich składowych swojej ostatniej kolumny macierzy X do P_{i+1}. Z kolei każdy proces P_i z wyjątkiem P_0 odbiera od P_{i-1} m liczb. Następnie wykonywane są instrukcje 2-4, a na koniec instrukcja 5. Otrzymujemy w ten sposób algorytm 5.2.

5.2. Algorytmy obliczania wartości filtrów rekurencyjnych

Jako podsumowanie powyższych rozważań odnotujmy, że do wyznaczenia ciągu poszczególnych wartości $y_1, y_2, \ldots, y_n$, zdefiniowanych wzorem (5.1) można posłużyć się prostym algorytmem 5.3. Z połączenia algorytmów 5.1 i 3.1 otrzymujemy algorytm 5.4, sekwencja zastosowania algorytmów 5.2 i 4.1 daje algorytm 5.5.

Wyznaczymy teraz koszt algorytmu wyznaczania ciągu sygnałów wyjściowych za pomocą sekwencji algorytmów 5.2 oraz 4.1 używając modelu BSP. Na początek zbadajmy koszt algorytmu mnożenia macierzy 5.2. Każdy proces operuje na bloku r/p kolumn. Komunikacja obejmująca instrukcje 5-10 może być zrealizowana w trakcie jednego superkroku, a zatem wykorzystując (5.7) oraz analizę dotyczącą liczby operacji potrzebnych dla realizacji poszczególnych kroków algorytmu otrzymujemy następujący koszt algorytmu

$$\begin{aligned} C^I_{p,r,s}(n,m) &= \underbrace{2mg + l}_{5\text{-}10} + \underbrace{2s(m+1)r/p - m(m+1)r/p}_{12\text{-}14} + \underbrace{m(m+1)r/p}_{15\text{-}22} + l \\ &= 2(mg + l) + 2s(m+1)r/p. \end{aligned} \tag{5.12}$$

Algorytm 5.2. Wyznaczanie współczynników f_k, $k = 1, \ldots, rs$, zdefiniowanych przez (5.3) przy ustalonych wartościach całkowitych $r > 1$, $s > 1$ przez proces P_i, $i = 0, \ldots, p-1$.

Wejście: liczby $a_0, \ldots, a_m$ oraz liczby tworzące macierz $X^{(i)}$ zdefiniowaną przez (5.8) albo (5.10).

Wyjście: $F^{(i)}$ zdefiniowana przez (5.9) albo (5.11).

1: $t \leftarrow \lfloor r/p \rfloor$
2: **if** $i = p - 1$ **then**
3: $\quad t \leftarrow r - (p-1)t$
4: **end if**
5: **if** $i \neq p - 1$ **then**
6: $\quad$ **send** $X^{(i)}_{s-m+1:s,t}$ **to** P_{i+1}
7: **end if**
8: **if** $i \neq 0$ **then**
9: $\quad$ **receive** $X^{(i)}_{s-m+1:s,0}$ **from** P_{i-1}
10: **end if**
11: $G \leftarrow \begin{pmatrix} a_m & \cdots & a_1 \\ & \ddots & \vdots \\ 0 & & a_m \end{pmatrix}$
12: **for** $k = 1$ to s **do**
13: $\quad F^{(i)}_{k,*} \leftarrow (a_{\min\{k-1,m\}}, \ldots, a_1, a_0) \cdot X^{(i)}_{\max\{1:k-m\}:k,*}$ {operacja `GEMV`}
14: **end for**
15: **if** $i \neq 0$ **then**
16: $\quad X^{(i)}_{s-m+1:s,0} \leftarrow GX^{(i)}_{s-m+1:s,0}$ {operacja `TRMV`}
17: $\quad F_{1:m,1} \leftarrow F_{1:m,1} + X^{(i)}_{s-m+1:s,0}$ {operacja `AXPY`}
18: **end if**
19: $X_{s-m+1:s,1:r-1} \leftarrow GX_{s-m+1:s,1:r-1}$ {operacja `TRMM`}
20: **for** $k = 2$ to r **do**
21: $\quad F^{(i)}_{1:m,k} \leftarrow F^{(i)}_{1:m,k} + X^{(i)}_{s-m+1:s,k-1}$ {operacja `AXPY`}
22: **end for**

Algorytm 5.3. Sekwencyjne (skalarne) obliczanie wartości filtra (5.1).

Wejście: liczby $x_1, \ldots, x_n$ oraz $a_0, \ldots, a_m$, $b_1, \ldots, b_m$

Wyjście: $y_1, \ldots, y_n$ spełniające (5.1)

```
for k = 1 to n do
    y_k ← a_0 x_k
    for j = 1 to min{m, k − 1} do
        y_k ← y_k + a_j x_{k−j} + b_j y_{k−j}
    end for
end for
```

Algorytm 5.4. Blokowe obliczanie wartości filtra (5.1).

Wejście: liczby $x_1, \ldots, x_n$ oraz $a_0, \ldots, a_m$, $b_1, \ldots, b_m$

Wyjście: $y_1, \ldots, y_n$ spełniające (5.1)

1: zastosuj algorytm 5.1 dla wyznaczenia $f_1, \ldots, f_n$ spełniających (5.3)

2: zastosuj algorytm 3.1 dla wyznaczenia $y_1, \ldots, y_n$ spełniających (5.2)

Algorytm 5.5. Blokowe obliczanie wartości filtra y_k, $k = 1, \ldots, rs$, zdefiniowanych przez (5.1) przy ustalonych wartościach całkowitych $r > 1$, $s > 1$ dla procesu P_i, $i = 0, \ldots, p - 1$.

Wejście: liczby $a_0, \ldots, a_m$, $b_1, \ldots, b_m$ oraz liczby tworzące macierz $X^{(i)}$ zdefiniowaną przez (5.8) albo (5.10)

Wyjście: macierz $Y^{(i)}$ liczb y_k spełniających (5.1)

1: zastosuj algorytm 5.2 dla wyznaczenia $F^{(i)}$

2: zastosuj algorytm 4.1 dla wyznaczenia $Y^{(i)}$

Koszt drugiej fazy, to znaczy obliczenia (5.2), otrzymujemy na podstawie wzorów (4.11)-(4.14):

$$\begin{aligned} C^{II}_{p,r,s}(n,m) &= 2m\,(s-(m+1)/2)\,(r/p+1)+l \\ &+ 3m^2r/p+mg+(p-2)\left(3m^2r/p+mg+l\right)+3m^2r/p+2l \\ &+ 2(s-m)m\,(r/p)+l, \end{aligned} \tag{5.13}$$

Stąd, aby znaleźć wartość optymalną s, minimalizujemy

$$C_{p,r,s}(n,m) = C^{I}_{p,r,s}(n,m) + C^{II}_{p,r,s}(n,m)$$

przy warunku $rs = n$ i uwzględniając to, że s musi być liczbą całkowitą, otrzymujemy

$$s^* = \left\lfloor \sqrt{\frac{n(3mp-3m-1)}{2p}} \right\rfloor \tag{5.14}$$

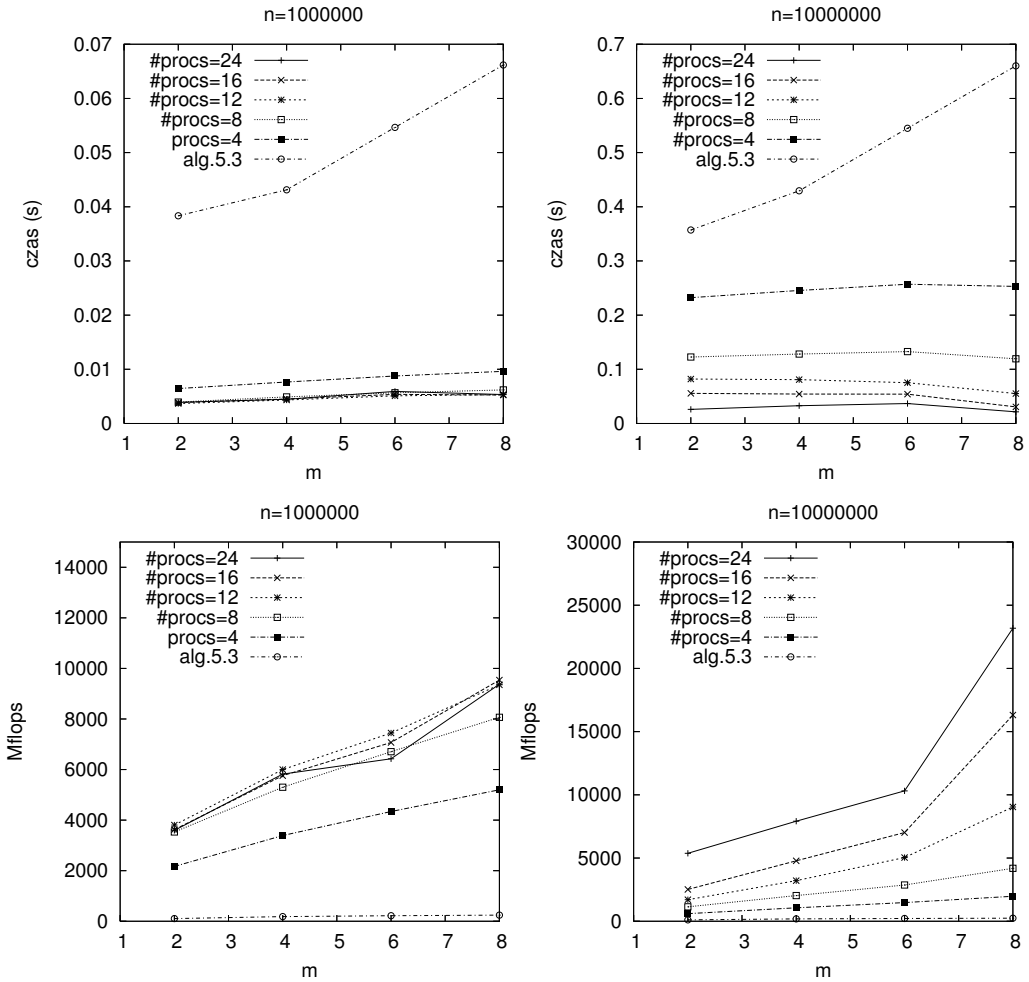
jako teoretyczną optymalną wartość parametru s. Stąd dostajemy również następujący wniosek.

Wniosek 5.1. *Dla modelu BSP optymalny wybór wartości parametru s w algorytmie 5.5 wyznaczania ciągu sygnałów wyjściowych (5.1) zależy wyłącznie od rozmiaru problemu n, m oraz liczby procesorów p.*

5.3. Wyniki eksperymentów

Algorytm 5.5 wyznaczania wartości (5.1) został zaimplementowany za pomocą standardu MPI oraz uruchomiony na klastrze komputerów z procesorami Itanium 2 (wyniki przedstawione na rysunku 5.1) oraz komputerze Cray X1, (rysunek 5.2), dla różnych rozmiarów problemu n i m, różnych wartości parametru s oraz różnej liczby procesorów. Mierzono czas wykonania algorytmu oraz wydajność.

Własności algorytmu 5.5 są podobne do omówionych w poprzednim rozdziale własności algorytmu 4.1. W przypadku klastra procesorów Itanium 2 algorytm wykorzystuje do około 30% jego teoretycznej wydajności obliczeniowej, podczas gdy prosty algorytm tylko około 4% wydajności pojedynczego procesora. Na rysunku 5.1 można zaobserwować, że dla dostatecznie dużych rozmiarów problemu ($n = 10000000$, $m = 8$) osiągnięto znaczny wzrost wydajności. Na komputerze Cray X1 dla algorytmu 5.5 osiągnięto wydajność działania nieco mniejszą niż 6000 Mflops, podczas gdy wydajność dla algorytmu opartego na (5.1) to zaledwie 40 Mflops.

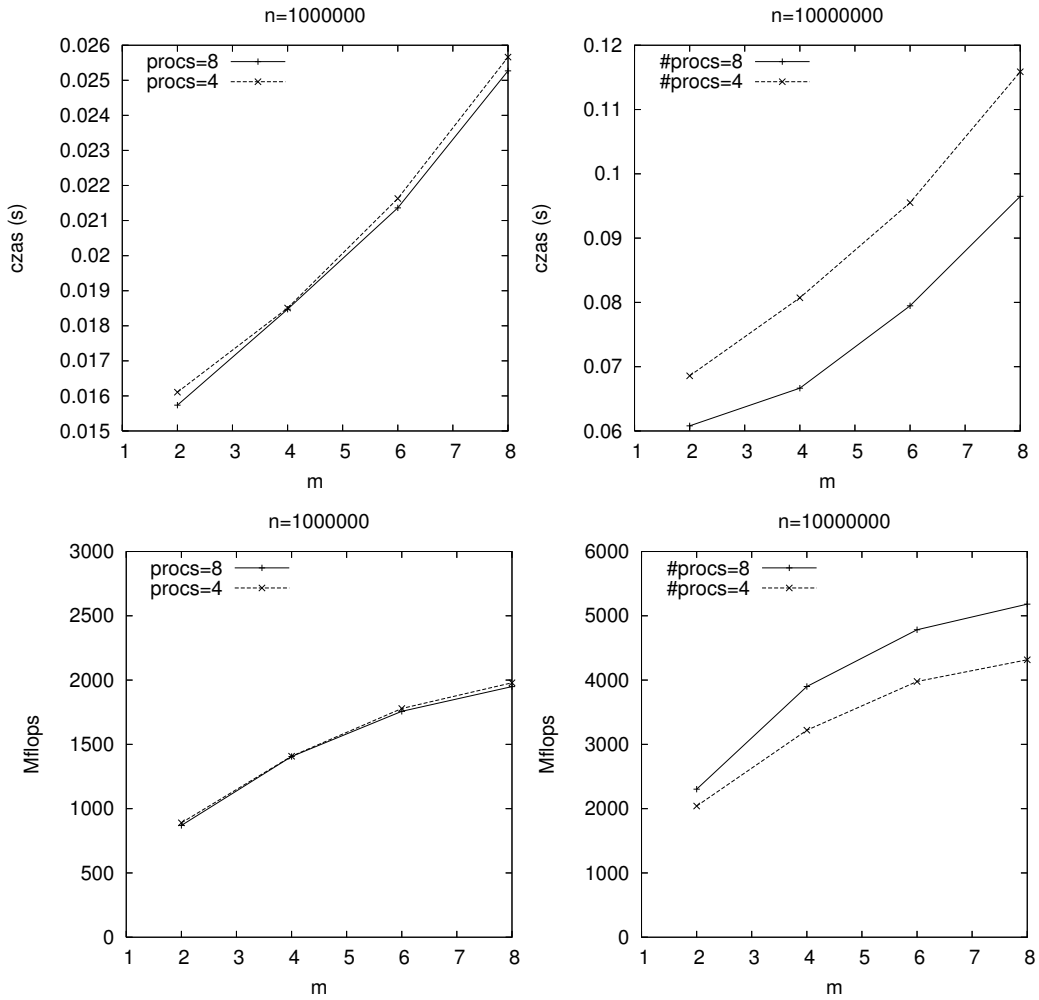


Rys. 5.1. Czas wykonania algorytmów 5.5 i 5.3 oraz wydajność dla klastra Itanium 2. Wartość s^* określona wzorem (5.14)

Fig. 5.1. Execution time of the algorithms 5.5 and 5.3 and their performance on a cluster of Itanium 2. The value of s^* is defined by (5.14)

5.4. Wykorzystanie nowego sposobu reprezentacji macierzy

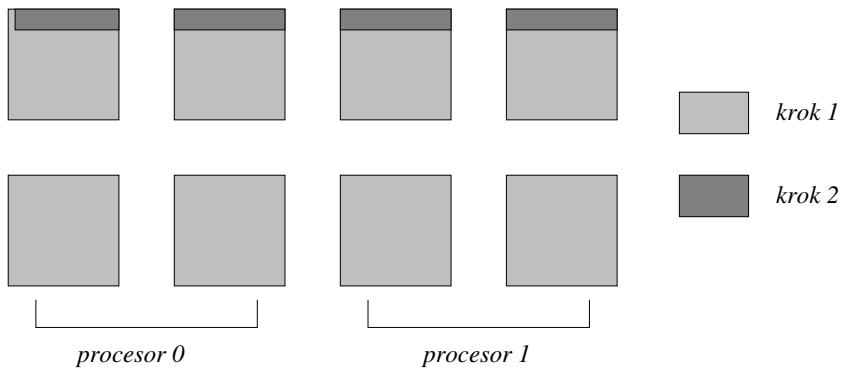
Omówimy teraz wykorzystanie omówionego w podrozdziale 1.2.3 nowego sposobu reprezentacji macierzy do implementacji algorytmu wyznaczania wartości filtra rekurencyjnego na bazie algorytmu 5.4. Dla uproszczenia przyjmijmy, że $s = n_b m_g$. Wówczas macierz X zdefiniowana wzorem (5.6) może być podzielona na bloki według (1.3) i wtedy wszystkie bloki (być może z wyjątkiem $X_{1n_g}, \ldots, X_{m_g n_g}$) będą kwadratowe.



Rys. 5.2. Czas wykonania algorytmu 5.5 oraz wydajność dla komputera Cray X1 (za optymalną wartość parametru s przyjęto $s \approx \sqrt{2n}$). Czas wykonania algorytmu 5.3 około 10 sekund

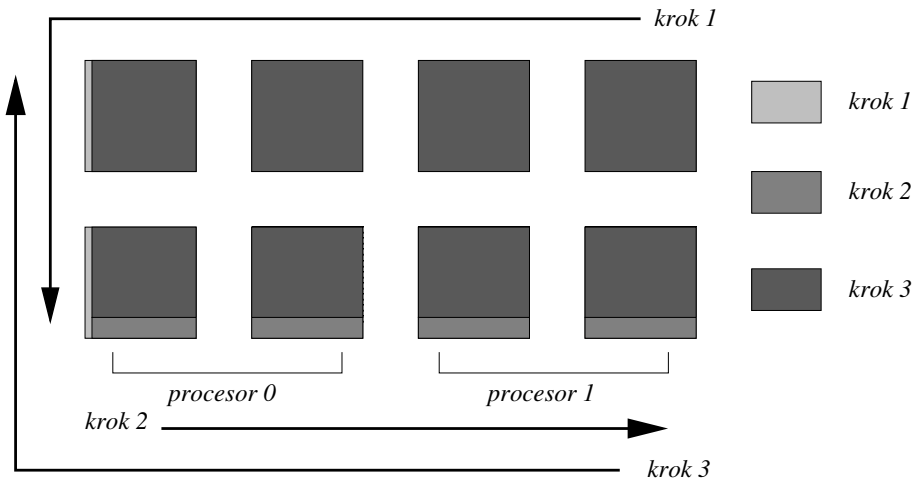
Fig. 5.2. Execution time of Algorithm 5.5 and its performance on a Cray X1 (the optimal value of s is $s \approx \sqrt{2n}$). Execution time of Algorithm 5.3 is about 10 s

Mnożenie macierzy przez wektor $F \leftarrow LX$ i aktualizacja pierwszych m składowych każdego wektora $\mathbf{f}_j$ $j = 1, \ldots, r$, czyli pierwszy etap algorytmu, może być wykonany lokalnie na każdej kolumnie bloków. Dzięki temu możliwe jest łatwe zrównoleglenie tego etapu, co zostało pokazane na rysunku 5.3. Każdy procesor odpowiada za wyznaczenie zbioru kolumn bloków. Podobnie realizujemy drugi etap algorytmu (rysunek 5.4). Zauważmy, że ze względu na większy stopień komplikacji drugiego etapu, jego przebieg jest nieco bardziej złożony: pierwszy i trzeci krok mogą być wykonywane równolegle, krok drugi zaś sekwencyjnie. Dodatkowo, aby zredukować zjawisko *cache miss*, bloki powinny być przetwarzane we właściwym porządku.



Rys. 5.3. Kroki algorytmu 5.2 używającego nowego sposobu reprezentacji macierzy (kolumny bloków mogą być wyznaczane w dowolnym porządku)

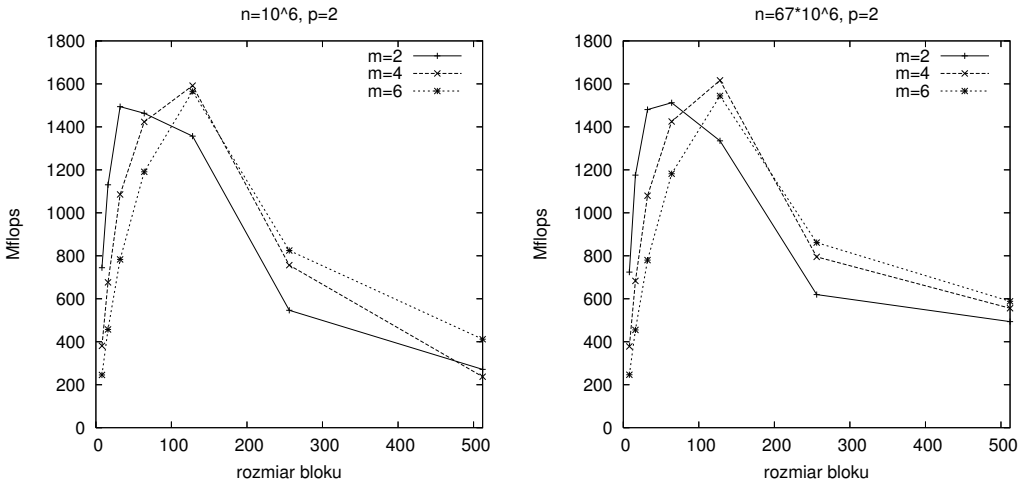
Fig. 5.3. Steps of Algorithm 5.2 using the square blocked data format (columns of blocks can be evaluated in any order)



Rys. 5.4. Kroki algorytmu 3.2 używającego nowego sposobu reprezentacji macierzy (strzałki wskazują kolejność wyznaczania bloków)

Fig. 5.4. Steps of Algorithm 3.2 using the square blocked data format (arrows indicate the order of the evaluation of the blocks)

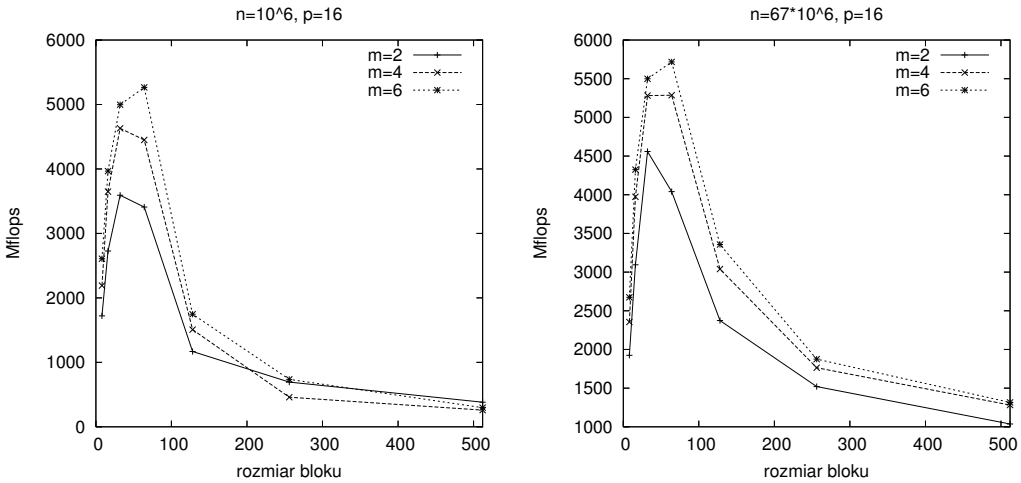
W czasie realizacji pierwszego kroku poszczególne kolumny powinny być przetwarzane od strony prawej do lewej, bloki zaś w ramach kolumny od góry do dołu. Następnie w drugim kroku przetwarzany jest dolny wiersz bloków od strony lewej do prawej. Ostatecznie, w trzecim kroku kolumny są przetwarzane ponownie od prawej do lewej, bloki zaś w ramach kolumny z dołu do góry. Dzięki temu zminimalizowana zostanie liczba pobrań poszczególnych bloków z pamięci, gdyż każdy następny krok będzie się rozpoczynał od przetwarzania bloku, na którym zakończył się krok poprzedni.



Rys. 5.5. Wydajność dwuprocesorowego komputera Itanium 2 dla równoległej wersji algorytmu 5.4 i nowego sposobu reprezentacji macierzy przy różnych n, m, n_b

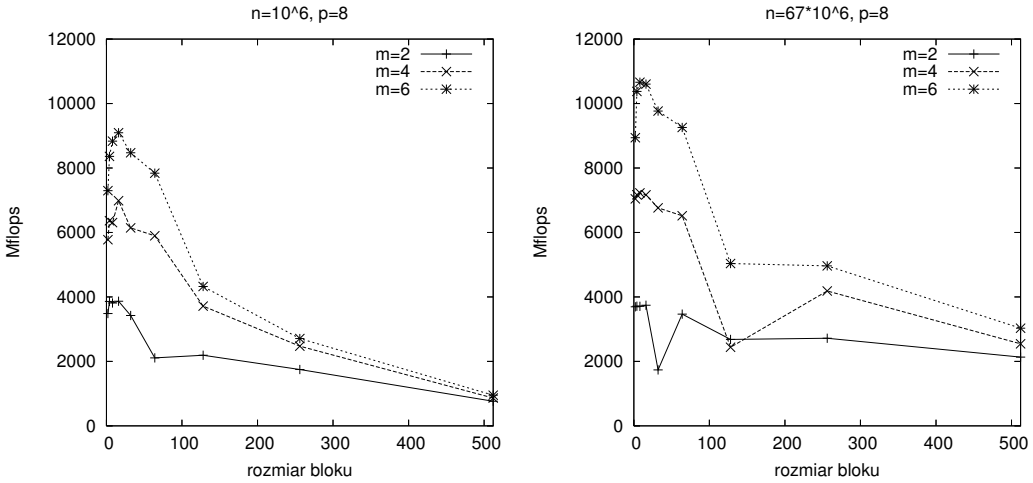
Fig. 5.5. Performance of the parallel version of Algorithm 5.4 using the square blocked data format for various n, m, n_b on a dual processor Itanium 2

Rysunki 5.5 – 5.10 pokazują przykładowe wyniki eksperymentów przeprowadzonych na dwuprocesorowym komputerze Itanium 2 (rysunki 5.5, 5.8), komputerze Cray X1 działającym w trybie SSP (rysunki 5.6, 5.9) oraz dwuprocesorowym komputerze Quad-Core Xeon (rysunki 5.7, 5.10). Rysunki 5.5, 5.6 i 5.7 pokazują wydajność poszczególnych komputerów przy wykorzystaniu maksymalnej dostępnej liczby procesorów i zmieniającym się rozmiarze bloku n_b. Z uwagi na to, że liczba operacji zmiennopozycyjnych w algorytmie 5.4 nie zależy od rozmiaru bloku, uzyskane wartości Mflops mogą posłużyć do porównania wydajności poszczególnych architektur w zależności od konkretnej wartości n_b. Wydajność (oraz oczywiście czas wykonania algorytmu) zależy bardzo wyraźnie od rozmiaru bloku.



Rys. 5.6. Wydajność komputera Cray X1 dla równoległej wersji algorytmu 5.4 i nowego sposobu reprezentacji macierzy przy różnych n, m, n_b

Fig. 5.6. Performance of the parallel version of Algorithm 5.4 using the square blocked data format for various n, m, n_b on a Cray X1



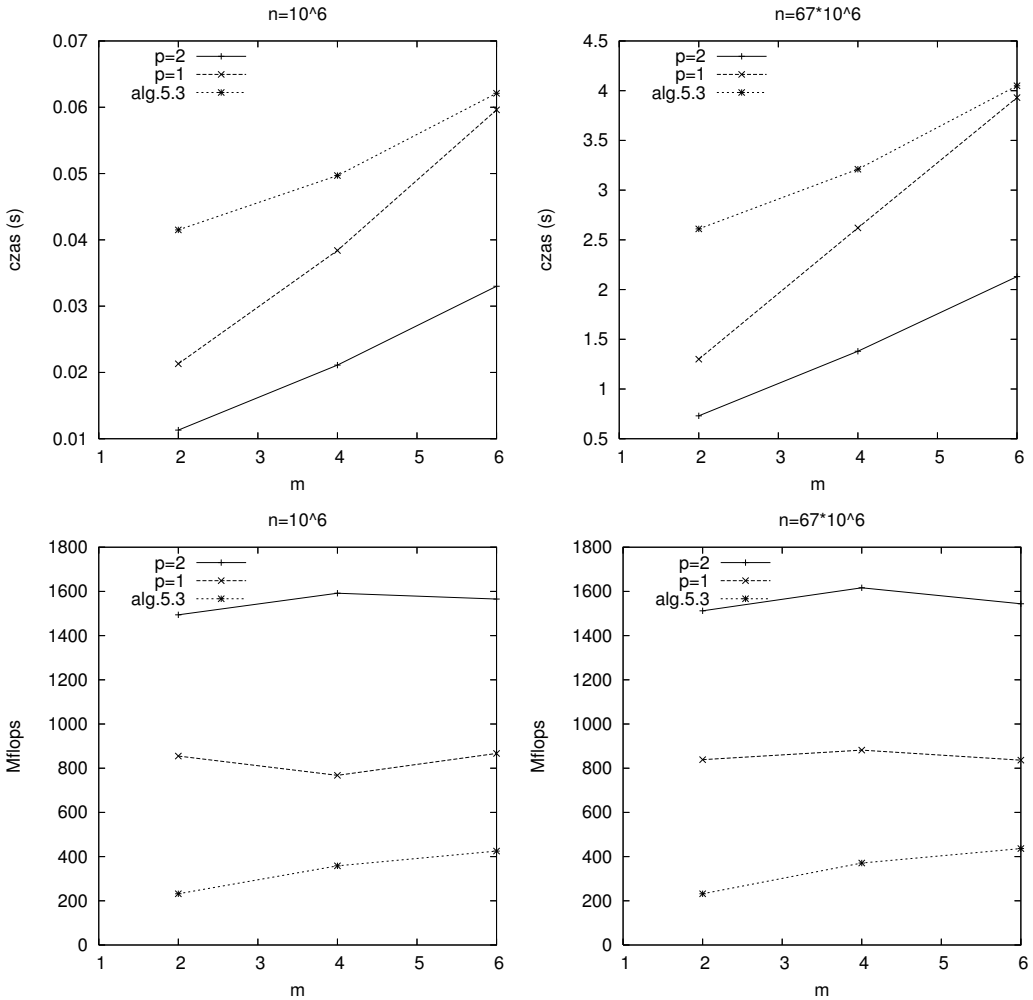
Rys. 5.7. Wydajność dwuprocesorowego komputera Quad-Core Xeon dla równoległej wersji algorytmu 5.4 i nowego sposobu reprezentacji macierzy przy różnych n, m, n_b

Fig. 5.7. Performance of the parallel version of Algorithm 5.4 using the square blocked data format for various n, m, n_b on a dual-processor Quad-Core Xeon

Na komputerze Itanium optymalna wartość n_b wynosi od 32 dla $m = 2$ do 128 dla większych wartości m. Na komputerze Cray X1 optymalny rozmiar bloku jest zawarty w przedziale wartości od 32 do 64.

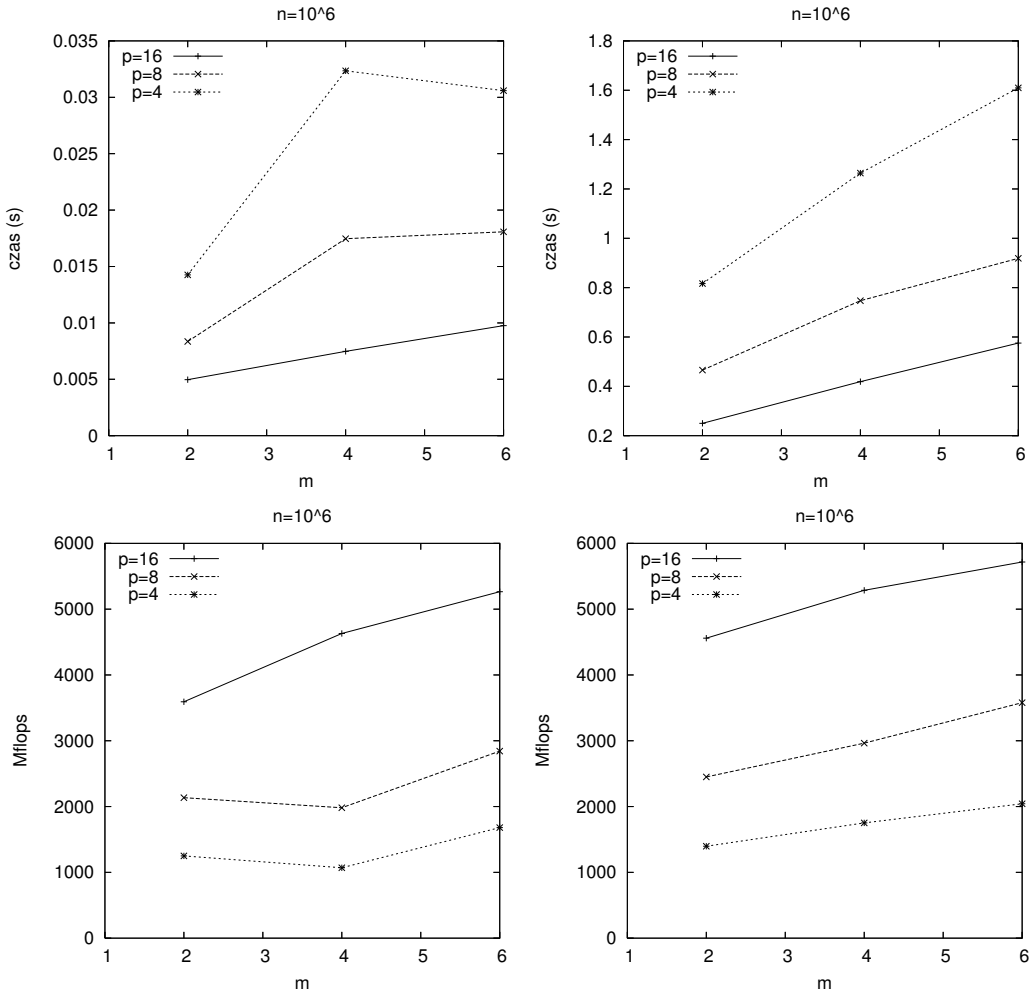
Na komputerze Xeon najlepiej jest przyjąć $n_b = 16$ dla każdej wartości m. W każdym przypadku przyjęcie zbyt dużego rozmiaru bloku powoduje znaczny spadek wydajności (wzrost czasu działania algorytmu) gdyż ma miejsce zjawisko *cache miss*. Algorytm jest znacznie mniej wrażliwy na właściwy wybór wartości parametrów r i s. W praktyce dobre wyniki uzyskuje się przy $r = s$ (kwadratowa siatka bloków).

Rysunki 5.8 – 5.10 pokazują, że algorytm wykorzystujący nowy sposób reprezentacji macierzy daje się dobrze zrównoleglić i w przypadku każdej architektury użycie większej liczby procesorów daje skrócenie czasu działania algorytmu. W każdym przypadku czas wykonania algorytmu jest krótszy niż prostego algorytmu 5.3, a to dzięki możliwości zastosowania wektoryzacji lub mechanizmów SSE. Można również zauważyć, że wraz ze wzrostem rozmiaru problemu (wartość n) bardzo nieznacznie rośnie wydajność każdej z architektur. Jednocześnie, porównując uzyskane wyniki z podstawowymi wersjami algorytmów 5.4 i 5.5 (podrozdział 5.3), można zauważyć, że użycie nowego sposobu reprezentacji macierzy istotnie skraca czas wyznaczenia rozwiązania problemu. Dodatkowo, znacznie łatwiej jest dobrać właściwy rozmiar bloku (parametr n_b) przy $r = s$, niż określać właściwy dobór wartości s w podstawowych wersjach algorytmów 5.4 i 5.5.



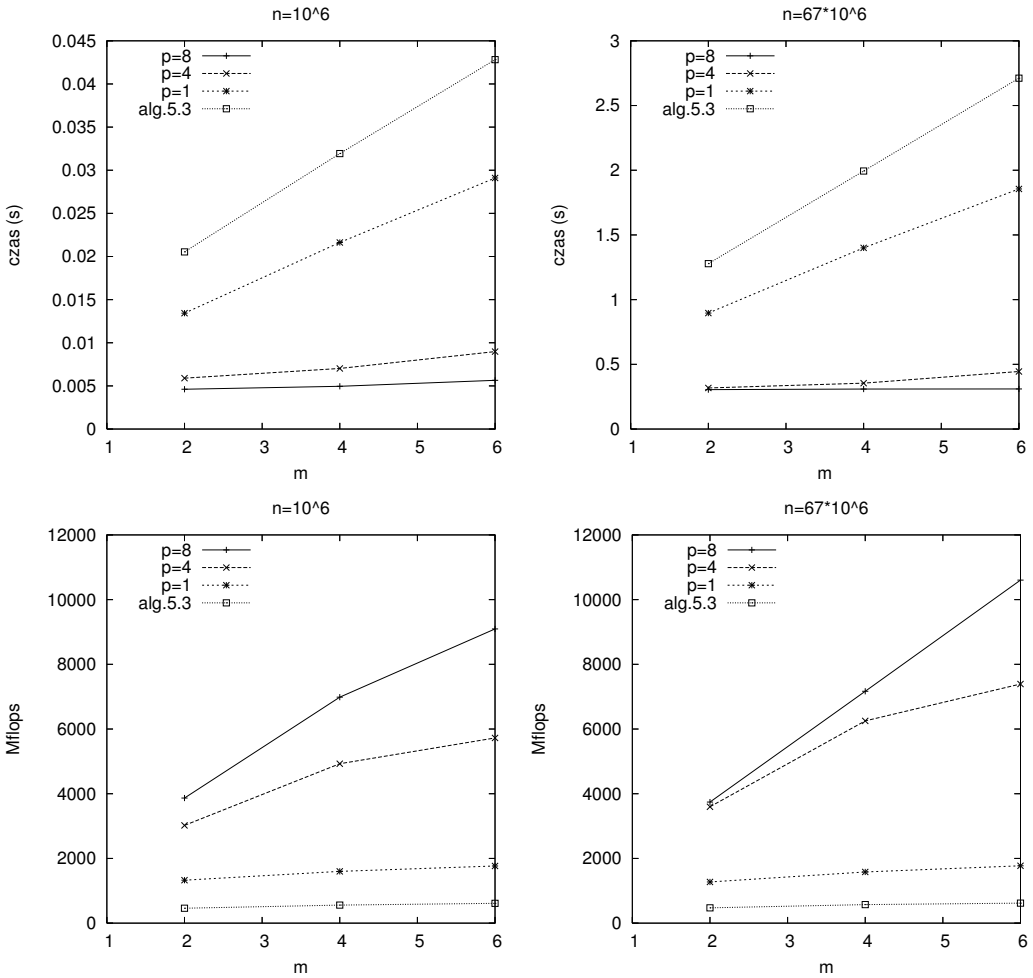
Rys. 5.8. Czas wykonania algorytmów 5.4 i 5.3 oraz wydajność komputera z procesorami Itanium 2 przy optymalnym rozmiarze bloku (nowa blokowa reprezentacja macierzy)

Fig. 5.8. Execution time of the algorithms 5.4 and 5.3 and their performance on a dual-processor Itanium 2 (with the square blocked data format)



Rys. 5.9. Czas wykonania algorytmu 5.4 oraz wydajność komputera Cray X1 przy optymalnym rozmiarze bloku (nowa blokowa reprezentacja macierzy)

Fig. 5.9. Execution time of Algorithm 5.4 and its performance on a Cray X1 (with the square blocked data format)



Rys. 5.10. Czas wykonania algorytmów 5.4 i 5.3 oraz wydajność komputera z procesorami Quad-Core Xeon przy optymalnym rozmiarze bloku (nowa blokowa reprezentacja macierzy)

Fig. 5.10. Execution time of the algorithms 5.4 and 5.3 and their performance on a dual-processor Quad-Core Xeon (with the square blocked data format)

6. Obliczanie sum trygonometrycznych i ich zastosowania

Zajmiemy się teraz problemem wyznaczania sum trygonometrycznych o postaci

$$C(x) = \sum_{k=0}^{n} b_k \cos kx \quad \text{oraz} \quad S(x) = \sum_{k=1}^{n} b_k \sin kx, \tag{6.1}$$

które mają zastosowanie między innymi w zagadnieniach związanych z interpolacją trygonometryczną [103] oraz numerycznym wyznaczaniem odwrotności transformanty Laplace'a, która z kolei ma duże znaczenie w obliczeniach naukowych i technicznych [1].

Najprostszym sposobem wyznaczenia wartości $C(x)$, $S(x)$ jest algorytm Goertzla [103], który sprowadza się do wyznaczenia rozwiązania liniowego równania rekurencyjnego o postaci (1.28), gdzie $m = 2$. Nie jest on jednak numerycznie stabilny dla małych co do wartości bezwzględnej x. Powszechnie stosowany jest następujący algorytm Reinscha [103]. Niech $S_{n+2} = D_{n+1} = 0$. Gdy $\cos x > 0$, wówczas wyznacza się rozwiązanie układu liniowych równań rekurencyjnych

$$\begin{cases} S_{k+1} = D_{k+1} + S_{k+2} \\ D_k = b_k + uS_{k+1} + D_{k+1} \end{cases} \tag{6.2}$$

dla $k = n, n-1, \ldots, 0$, gdzie $u = -4\sin^2 \frac{x}{2}$. Gdy $\cos x \leq 0$, wówczas wyznacza się rozwiązanie układu

$$\begin{cases} S_{k+1} = D_{k+1} - S_{k+2} \\ D_k = b_k + uS_{k+1} - D_{k+1} \end{cases} \tag{6.3}$$

gdzie $u = 4\cos^2 \frac{x}{2}$. Ostatecznie, w obu przypadkach szukane wartości wyznaczamy ze wzorów

$$S(x) = S_1 \sin x, \qquad C(x) = D_0 - \frac{u}{2} S_1. \tag{6.4}$$

Wzory (6.2) i (6.3) można łatwo zaprogramować, jednak taki algorytm będzie sekwencyjny, nie będzie można go w bezpośredni sposób zrównoleglić, nie będzie też możliwa wektoryzacja i wykorzystanie podprogramów biblioteki BLAS. W pracy [27] podano równoległą wersję algorytmu Reinscha, ale bez możliwości wektoryzacji oraz użycia biblioteki BLAS. W pracach [118, 91] podaliśmy inną równoległą wersję algorytmu, o podobnych własnościach jak

algorytm z pracy [27], ale umożliwiającą zastosowanie operacji `AXPY`. Pokażemy, jak zredukować liczbę operacji arytmetycznych w równoległej wersji algorytmu oraz podać jego wersję umożliwiającą efektywną wektoryzację przez redukcję liczby kontaktów z pamięcią, co również opublikowaliśmy w pracy [115].

6.1. Wektorowa-równoległa wersja algorytmu Reinscha

Dla uproszczenia przyjmijmy na początek, że liczba całkowita n we wzorze (6.1) może być zapisana jako iloczyn dwóch liczb całkowitych $pq = n$. Przyjmijmy dalej, że

$$x_k = \begin{cases} S_{n-\lfloor k/2 \rfloor} & \text{dla } k = 1, 3, \ldots, 2n-1 \\ D_{n-k/2} & \text{dla } k = 2, 4, \ldots, 2n \end{cases}, \tag{6.5}$$

$$f_k = \begin{cases} b_n & \text{dla } k = 1 \\ b_{n-1} - \delta b_n & \text{dla } k = 2 \\ 0 & \text{dla } k = 3, 5, \ldots, 2n-1 \\ b_{n-k/2} & \text{dla } k = 4, 6, \ldots, 2n \end{cases} \tag{6.6}$$

oraz zdefiniujmy macierz

$$L = \begin{pmatrix} 1 & & & & & & \\ -u & 1 & & & & & \\ \delta & -1 & 1 & & & & \\ & \delta & -u & 1 & & & \\ & & \delta & -1 & 1 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \delta & -u & 1 \end{pmatrix} \in \mathbb{R}^{2n \times 2n}, \tag{6.7}$$

gdzie

$$\delta = \begin{cases} -1 & \text{dla } \cos x > 0 \\ 1 & \text{dla } \cos x \leq 0. \end{cases} \tag{6.8}$$

Zauważmy, że aby wyznaczyć wielkości S_1, D_0, występujące we wzorze (6.4), należy wyznaczyć wszystkie liczby ze wzorów (6.2) albo (6.3). Wzory te są równoważne obliczeniu dwóch ostatnich składowych rozwiązania układu

$$L\mathbf{x} = \mathbf{f}, \tag{6.9}$$

gdzie

$$\mathbf{x} = (x_1, \ldots, x_{2n})^T, \quad \mathbf{f} = (f_1, \ldots, f_{2n})^T \in \mathbb{R}^{2n}. \tag{6.10}$$

Wyznaczenie rozwiązania układu (6.9) można przeprowadzić za pomocą algorytmu opartego na postępowaniu opisanym w podrozdziale 3.1. Niech zatem $L^{(q)} \in \mathbb{R}^{2q\times 2q}$ oznacza macierz utworzoną z pierwszych $2q$ wierszy i kolumn macierzy L danej wzorem (6.7). Niech dalej

$$U^{(q)} = \begin{pmatrix} & \delta & -1 \\ & 0 & \delta \\ & & \\ 0 & & \end{pmatrix} \in \mathbb{R}^{2q\times 2q} \tag{6.11}$$

oraz

$$\mathbf{f}_j = (f_{2(j-1)q+1}, \ldots, f_{2jq})^T, \qquad \mathbf{z}_j = (z_{2(j-1)q+1}, \ldots, z_{2jq})^T.$$

Wówczas dla $m = 2$, wzór (2.5) przyjmie postać

$$\begin{cases} \mathbf{x}_1 = (L^{(q)})^{-1}\mathbf{f}_1 \\ \mathbf{x}_j = (L^{(q)})^{-1}\mathbf{f}_j - (L^{(q)})^{-1}U^{(q)}\mathbf{x}_{j-1} \quad \text{dla } j = 2, \ldots, p. \end{cases} \tag{6.12}$$

Zauważmy dalej, że iloczyn macierzy $(L^{(q)})^{-1}U^{(q)}$ jest następującej postaci

$$(L^{(q)})^{-1}U^{(q)} = (\underbrace{\mathbf{0}, \ldots, \mathbf{0}}_{2q-2}, \mathbf{y}_1, \mathbf{y}_2) \in \mathbb{R}^{2q\times 2q},$$

gdzie wektory $\mathbf{y}_1$, $\mathbf{y}_2$ są rozwiązaniami następujących układów równań liniowych

$$L^{(q)}\mathbf{y}_1 = (\delta, 0, \ldots, 0)^T, \;\; L^{(q)}\mathbf{y}_2 = (-1, \delta, 0, \ldots, 0)^T. \tag{6.13}$$

Prawe strony układów (6.13) stanowią odpowiednio dwie ostatnie kolumny macierzy $U^{(q)}$, a pierwsze $2q-2$ kolumn macierzy $U^{(q)}$ zawiera same zera. Stąd otrzymujemy

$$-(L^{(q)})^{-1}U^{(q)}\mathbf{x}_{j-1} = -x_{2(j-1)q-1}\mathbf{y}_1 - x_{2(j-1)q}\mathbf{y}_2. \tag{6.14}$$

Oczywiście, naszym celem jest wyznaczenie jedynie wielkości S_1, D_0, zatem zastosowanie wzoru (6.14) może być ograniczone do dwóch ostatnich składowych wektora wynikowego (czyli problemu wyznaczenia cząstkowego rozwiązania liniowego równania rekurencyjnego). Niech

$$\mathbf{y}_1 = (y_1^{(1)}, \ldots, y_{2q}^{(1)})^T, \tag{6.15}$$

$$\mathbf{y}_2 = (y_1^{(2)}, \ldots, y_{2q}^{(2)})^T, \tag{6.16}$$

oraz zdefiniujmy macierz

$$M = \begin{pmatrix} y_{2q-1}^{(1)} & y_{2q-1}^{(2)} \\ y_{2q}^{(1)} & y_{2q}^{(2)} \end{pmatrix} \in \mathbb{R}^{2\times 2}. \tag{6.17}$$

Na podstawie wzoru (6.12) mamy

$$\begin{pmatrix} S_1 \\ D_0 \end{pmatrix} = \begin{pmatrix} x_{2n-1} \\ x_{2n} \end{pmatrix} = \begin{pmatrix} z_{2n-1} \\ z_{2n} \end{pmatrix} - \sum_{j=1}^{p-1} M^{p-j} \begin{pmatrix} z_{jq-1} \\ z_{jq} \end{pmatrix}. \tag{6.18}$$

Aby wyznaczyć macierz M, nie musimy obliczać rozwiązań układów równań (6.13). Liczby $y_{2q-1}^{(1)}$, $y_{2q}^{(1)}$, $y_{2q-1}^{(2)}$, $y_{2q}^{(2)}$ mogą być wyznaczone bezpośrednio dzięki zastosowaniu następujących twierdzeń.

Twierdzenie 6.1. *Dla $x \neq k\pi$, $k \in \mathbb{Z}$, dwie ostatnie składowe wektora $\mathbf{y}_1 = (y_1^{(1)}, \ldots, y_{2q}^{(1)})^T$ zdefiniowanego przez (6.13) spełniają*

$$y_{2q-1}^{(1)} = (\delta \sin qx + \sin(q-1)x)/\sin x \tag{6.19}$$

$$y_{2q}^{(1)} = \delta \cos qx + \cos(q-1)x + \frac{\beta}{2} y_{2q-1}^{(1)} \tag{6.20}$$

gdzie δ zdefiniowano przez (6.8) oraz $u = -4\sin^2 \frac{x}{2}$ dla $\cos x > 0$ i $u = 4\cos^2 \frac{x}{2}$ dla $\cos x \leq 0$.

Dowód. Wyznaczenie dwóch ostatnich składowych wektora $\mathbf{y}_1$ może być zrealizowane za pomocą algorytmu Reinscha, gdzie $n = q$ oraz współczynniki b_k, $k = 1, \ldots, q$, są dane wzorem

$$b_k = \begin{cases} \delta & \text{dla } k = q \\ 1 & \text{dla } k = q-1 \\ 0 & \text{dla } k = 1, \ldots, q-2. \end{cases}$$

Istotnie, stosując (6.6), otrzymujemy $f_1 = \delta$ oraz

$$f_2 = b_{q-1} - \delta b_q = 1 - \delta^2 = 0.$$

Stąd na mocy (6.4) mamy

$$S(x) = S_1 \sin x = \sin(q-1)x + \delta \sin qx.$$

Podobnie, korzystając z (6.4), otrzymujemy

$$C(x) = D_0 - \frac{u}{2} S_1 = \cos(q-1)x + \delta \cos qx.$$

Przyjmując $y_{2q-1}^{(1)} = S_1$ oraz $y_{2q}^{(1)} = D_0$ otrzymujemy odpowiednio wzory (6.19) i (6.20). ■

Twierdzenie 6.2. *Dla $x \neq k\pi$, $k \in \mathbb{Z}$, dwie ostatnie składowe wektora $\mathbf{y}_2 = (y_1^{(2)}, \ldots, y_q^{(2)})^T$ zdefiniowanego przez (6.13) spełniają*

$$y_{2q-1}^{(2)} = -\sin qx / \sin x \tag{6.21}$$

$$y_{2q}^{(2)} = -\cos qx + \frac{\beta}{2} y_{2q-1}^{(2)} \tag{6.22}$$

gdzie δ zdefiniowano przez (6.8) oraz $u = -4\sin^2 \frac{x}{2}$ dla $\cos x > 0$ i $u = 4\cos^2 \frac{x}{2}$ dla $\cos x \leq 0$.

Dowód. Wyznaczenie dwóch ostatnich składowych wektora $\mathbf{y}_2$ może być zrealizowane za pomocą algorytmu Reinscha, gdzie współczynniki b_k, $k = 1, \ldots, q$, są dane przez

$$b_k = \begin{cases} -1 & \text{dla } k = q \\ 0 & \text{dla } k = 1, \ldots, q-1. \end{cases}$$

Istotnie, $f_1 = -1$ oraz

$$f_2 = b_{q-1} - \delta b_q = 0 - (-1)\delta = \delta,$$

zatem $S_1 = -\sin qx / \sin x$ oraz $D_0 = \frac{u}{2} S_1 - \cos qx$, co daje wzory (6.19) i (6.20). ■

Przyjęte na początku założenie $pq = n$ może być pominięte. Przedstawiony wyżej algorytm może być zastosowany do wyznaczenia dwóch ostatnich składowych rozwiązania układu równań (6.9), a następnie pozostałe liczby $x_{2pq+1}, \ldots, x_n$ obliczone ze wzorów (6.2)–(6.3).

Kolejnym problemem jest dobór odpowiednich wartości parametrów p i q. Jeśli przyjmiemy, że p jest równe liczbie dostępnych procesorów, to otrzymamy w ten sposób prosty algorytm typu *divide and conquer*, w którym nie będzie możliwa wektoryzacja obliczeń. W pracy [106] podaliśmy implementację takiego właśnie przypadku w języku High Performance Fortran, charakteryzującą się dość dobrą efektywnością zrównoleglenia, jednak z powodu braku możliwości wektoryzacji oraz użycia wyższych poziomów biblioteki BLAS, wykorzystującą wydajność obliczeniową procesora w sposób dość ograniczony.

Innym rozwiązaniem jest wybór parametrów, tak by zminimalizować liczbę operacji w algorytmie. Niestety, funkcja liczby operacji zmiennopozycyjnych w algorytmie, w zależności od zmiennych p i q, przy warunku $pq = n$, nie posiada minimum lokalnego, a najmniejsza wartość jest osiągnięta dla $p = 1$, co oczywiście nie daje możliwości wykorzystania wektoryzacji obliczeń. Wykorzystując fakt, że opisany algorytm jest szczególnym przypadkiem algorytmu 3.1, w pracach [107, 115] przyjęliśmy $q = \lfloor \sqrt{3n/2} \rfloor$ oraz $p = \lfloor n/q \rfloor$ jako wartość minimalizującą liczbę operacji w algorytmie 3.1 przy $m = 2$. Jednocześnie podkreślmy, że wybór dowolnej wartości q rzędu $\sqrt{n}$ daje na ogół podobne czasy działania algorytmu.

Zajmiemy się teraz efektywnymi sposobami wyznaczania rozwiązania układu równań liniowych

$$L^{(q)}(\mathbf{z}_1, \ldots, \mathbf{z}_p) = (\mathbf{f}_1, \ldots, \mathbf{f}_p). \tag{6.23}$$

Korzystając z (6.6), można zaobserwować, że macierz $F = (\mathbf{f}_1, \ldots, \mathbf{f}_p)$ jest następującej postaci

$$F = \begin{pmatrix} * & 0 & 0 & \cdots & 0 & 0 & 0 \\ * & * & * & \cdots & * & * & * \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ * & * & * & \cdots & * & * & * \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ * & * & * & \cdots & * & * & * \end{pmatrix} \in \mathbb{R}^{2q \times p}. \tag{6.24}$$

Każdy wiersz o numerze nieparzystym (z wyjątkiem pierwszego) składa się z samych zer, a każdy wiersz o numerze parzystym składa się z (na ogół) różnych od zera liczb

$$f_k, f_{k+2q}, f_{k+4q}, \ldots, f_{k+2(p-2)q}, f_{k+2(p-1)q}.$$

Wykorzystując (6.6), możemy zaobserwować, że są to następujące współczynniki

$$b_{n-k/2}, b_{n-k/2-q}, b_{n-k/2-2q}, \ldots, b_{n-k/2-(p-2)q}, b_{n-k/2-(p-1)q} \tag{6.25}$$

występujące we wzorze (6.1). Zauważmy, że ostatni współczynnik w k-tym wierszu to $b_{q-k/2}$ oraz pierwszy element drugiego wiersza to $b_{n-1} - \delta b_n$. Biorąc pod uwagę postać macierzy F określoną wzorem (6.24), przy wyznaczaniu rozwiązania układu (6.23) „nadpisujemy” rozwiązanie na macierz F. Najpierw modyfikujemy element w pierwszej kolumnie i drugim wierszu wykonując operację

$$F_{2,1} \leftarrow F_{2,1} + uF_{1,1}.$$

Dalej dla każdego wiersza o numerze nieparzystym $k \geq 3$ wykonujemy następującą operację wektorową

$$F_{k,*} \leftarrow F_{k-1,*} - \delta F_{k-2,*}. \tag{6.26}$$

Następnie aktualizujemy następny (parzysty) wiersz, wykorzystując dwa poprzednie wiersze

$$F_{k+1,*} \leftarrow F_{k+1,*} + uF_{k,*} - \delta F_{k-1,*}. \tag{6.27}$$

Podkreślmy, że nie wykonujemy mnożenia przez δ, a tylko w zależności od znaku δ dodajemy lub odejmujemy wektor $F_{k-2,*}$ we wzorze (6.26) oraz $F_{k-1,*}$ w (6.27). Następująca pętla w języku Fortran stanowi główny fragment implementacji rozwiązywania układu (6.23), opierając się na wzorach (6.26) oraz (6.27) dla przypadku $\delta = 1$.

```
do k=3,2*q,2
  call dcopy(p,f(k-1,1),2*q,f(k,1),2*q)
  call daxpy(p,dble(-1),f(k-2,1),2*q,f(k,1),2*q)
  call daxpy(p,u,f(k,1),2*q,f(k+1,1),2*q)
  call daxpy(p,dble(-1),f(k-1,1),2*q,f(k+1,1),2*q)
end do
```

Zauważmy, że każda iteracja pętli wymaga jedenastu odczytów i zapisów wektorów o długości p (operacja `COPY` wymaga odczytu i zapisu wektora, operacja `AXPY` dwóch odczytów i jednego zapisu). Wyniki eksperymentów zamieszczone w pracy [107] pokazują, że algorytm charakteryzuje się niską efektywnością w przypadku niewielkich rozmiarów problemu. Może być jednak ulepszony dzięki redukcji liczby odwołań do pamięci. Rozważmy następujący fragment kodu realizujący identyczną operację jak powyżej.

```
do k=3,2*q,2
  do j=1,p
    f(k,j)=f(k-1,j)-f(k-2,j)
    f(k+1,j)=f(k+1,j)+u*f(k,j)-f(k-1,j)
  end do
end do
```

Na każdą iterację pętli wewnętrznej o łącznej liczbie iteracji p przypada siedem odwołań do pamięci. Testy pokazują, że w przypadku procesorów Itanium i Pentium tak zmodyfikowany algorytm działa około 15% szybciej niż wersja wykorzystująca sekwencję operacji `AXPY`. Oczywiście, taka pętla może być też zwektoryzowana. Zauważmy również, że nie ma potrzeby pamiętania kolejnych wierszy wyznaczanego rozwiązania, a zatem zamiast dwuwymiarowej tablicy `f` można posłużyć się dwoma tablicami jednowymiarowymi `fo` i `fe` odpowiednio dla obliczanych wierszy o numerach nieparzystych i parzystych, a tablica `ff` dla przechowywania wyznaczanych kolejno według (6.25) wierszy parzystych macierzy F. Podany wyżej fragment kodu przyjmie wówczas następującą postać:

```
do k=3,2*q,2
  do j=1,p
    ff(j)= ......  <- obliczenie wartości
  end do
  do j=1,p
    fo(j)=fe(j)-fo(j)
    fe(j)=ff(j)+u*fo(j)-fe(j)
  end do
end do
```

Rozbicie pętli wewnętrznej na dwie pętle ma umożliwić efektywną wektoryzację wyznaczania rozwiązania układu (6.23) przez zastosowanie opisanego w podrozdziale 1.1 mechanizmu łańcuchowania: obliczane kolejno w sposób potokowy składowe tablicy `fo` mogą być na bieżąco wykorzystywane przy potokowym wyznaczaniu składowych tablicy `fe`. Na koniec zauważmy, że obliczanie rozwiązania układu (6.23) może być zrównoleglone w sposób analogiczny do zastosowanego w algorytmie 3.2.

6.2. Optymalizacja metody Talbota

Jako przykład zastosowania przedstawionego w poprzednim punkcie algorytmu obliczania sum trygonometrycznych (6.1) rozważmy następujący problem numerycznego wyznaczenia odwrotności transformanty Laplace'a, to znaczy znalezienia funkcji

$$f(t) : (0, +\infty) \longrightarrow \mathbb{R}$$

spełniającej równanie

$$F(s) = \int_0^\infty e^{-st} f(t)dt, \qquad \operatorname{Re} s > \sigma_0. \tag{6.28}$$

Metoda Talbota [121] polega na zastąpieniu wzoru Riemanna

$$f(t) = \frac{1}{2\pi i} \int_B e^{st} F(s)ds,$$

gdzie B oznacza kontur Bromwhicha, następującym wzorem całkowym

$$f(t) = \frac{\lambda e^{\sigma t}}{2\pi i} \int_{-\pi}^{\pi} e^{\lambda t s_\nu(\theta)} F(\sigma + \lambda s_\nu(\theta)) s'_\nu(\theta) d\theta, \tag{6.29}$$

gdzie λ, σ, ν oznaczają parametry metody oraz $s_\nu(\theta) = \theta \operatorname{ctg} \theta + i\nu\theta$. Stosując wzór trapezów [65, rozdział 7], można przybliżyć funkcję $f(t)$ przez

$$\tilde{f}(t) = \frac{\lambda e^{\sigma t}}{n} \left(\frac{\nu}{2} e^{\lambda t} F(\sigma + \lambda) + S_n(t) \right), \tag{6.30}$$

przy czym

$$S_n(t) = \sum_{j=1}^{n-1} c_j \cos \phi_j - \sum_{j=1}^{n-1} s_j \sin \phi_j, \tag{6.31}$$

$$c_j = e^{\rho_j}(\alpha_j \gamma_j - \beta_j \delta_j), \quad s_j = e^{\rho_j}(\beta_j \gamma_j + \alpha_j \delta_j) \tag{6.32}$$

oraz

$$\theta_j = j\frac{\pi}{n}, \quad \rho_j = \lambda t j \frac{\pi}{n} \operatorname{ctg}(j\frac{\pi}{n}), \quad \phi_j = \lambda t \nu j \frac{\pi}{n},$$

$$F(\sigma + \lambda s_\nu(\theta_j)) = \alpha_j + i\beta_j, \quad \frac{1}{i} s'_\nu(\theta_j) = \gamma_j + i\delta_j.$$

Wielkość zdefiniowana wzorem (6.31), stanowiącym centralną część metody, może być wyznaczona za pomocą algorytmu Reinscha [82, 27], czyli również metodą opisaną w poprzednim podrozdziale. Obliczenie $S_n(t)$ wymaga dwukrotnego zastosowania algorytmu: pierwszy raz do wyznaczenia sumy

$$C = \sum_{j=1}^{n-1} c_j \cos \phi_j, \tag{6.33}$$

a następnie

$$S = \sum_{j=1}^{n-1} s_j \sin \phi_j. \tag{6.34}$$

Oryginalna implementacja metody Talbota [82] składa się z dwóch podprogramów: `TAPAR` znajdującego dla zadanej dokładności ε, wartości parametrów λ, σ, ν oraz wielkość n występującą we wzorze (6.31). Podprogram `TSUM` oblicza (6.31), a następnie (6.30). W naszej implementacji [115] podprogram `TSUM` został zastąpiony podprogramem `PXTSUM`, który w sposób równoległy, przy wykorzystaniu OpenMP, wyznacza kolejno współczynniki c_j, s_j, stanowiące wiersz (6.25) przy zastosowaniu (6.32), umieszcza je w dwóch tablicach jednowymiarowych, a następnie wyznacza kolejne wiersze parzyste i nieparzyste, stanowiące rozwiązanie układu o postaci (6.23), oddzielnie dla obliczenia sumy (6.33) oraz drugiej sumy (6.34). Wyznaczenie współczynników c_j, s_j jest przeprowadzane w sposób umożliwiający wektoryzację. Potrzebne do tego są tymczasowe tablice jednowymiarowe dla składowania potrzebnych do wyznaczenia wektora (6.25) wektorów poszczególnych wielkości α_j, β_j, γ_j, δ_j, γ_j, ρ_j. Ostatnią czynnością podprogramu `PXTSUM` jest obliczenie przybliżenia wartości $f(t)$ przy użyciu (6.30).

Algorytm został przetestowany dla funkcji pochodzących ze zbioru funkcji testowych zaproponowanego w pracy [82], dla różnych wartości argumentu t oraz różnych wartości parametru ε. Wykorzystano komputer z dwoma procesorami Itanium 2, komputer z procesorem Pentium 4, oraz komputer Cray X1 działający w trybie SSP. Porównano szybkość działania i dokładność z oryginalną implementacją metody Talbota [82]. Tabele 6.1 – 6.5 zawierają przykładowe wyniki dla funkcji $F(s) = 1/s^2$ oraz $F(s) = \operatorname{arc\,tg}(1/s)$, przy czym n oznacza liczbę występującą we wzorze (6.31), wielkości t_1, e_1 czas działania oraz błąd względny rozwiązania dla algorytmu Talbota, a wielkości t_2, e_2 czas działania oraz błąd względny rozwiązania dla naszej metody. Dla każdej tabeli podano dokładną funkcję f, spełniającą (6.28) oraz argument t, dla którego szukamy $f(t)$.

Tabela 6.1
Czas działania i błąd względny wyniku dla oryginalnego algorytmu Talbota (t_1, e_1) i algorytmu wektorowo-równoległego (t_2, e_2) dla $F(s) = 1/s^2$, $f(t) = t$, $t = 100.0$ i różnych wartości n na komputerze Itanium 2

n	t_1	t_2	e_1	e_2
15195	0.3422E-02	0.2136E-02	0.51715E-10	0.52615E-10
16281	0.3666E-02	0.2222E-02	0.48268E-10	0.48801E-10
16801	0.3783E-02	0.2316E-02	0.46775E-10	0.47975E-10
17177	0.3868E-02	0.2380E-02	0.45754E-10	0.47768E-10
17711	0.3987E-02	0.2479E-02	0.44367E-10	0.45774E-10
17941	0.4039E-02	0.2480E-02	0.43797E-10	0.44845E-10
17865	0.4022E-02	0.2449E-02	0.43992E-10	0.42524E-10

W przypadku komputera z procesorami Itanium 2 można zaobserwować przyspieszenie około 1.6 względem metody Talbota, nawet dla niewielkich wartości n. Na komputerze z procesorem Pentium 4 nasza metoda jest do 20% szybsza niż oryginalna implementacja metody Talbota. Rysunek 6.1 pokazuje przyspieszenie uzyskane na komputerze Cray X1 dla różnych wartości n. Możemy zaobserwować, że przyspieszenie na ogół rośnie wraz ze wzrostem wartości n. Przeprowadzone testy pokazują, że omówiona metoda charakteryzuje się podobnymi własnościami numerycznymi jak metoda Talbota (tabele 6.1 – 6.5).

Pokazaliśmy zatem, że wprowadzone we wcześniejszych rozdziałach metody wyznaczania cząstkowych rozwiązań równań (1.28) mogą być z powodzeniem zastosowane do rozwiązywania pewnych szczególnych postaci równań (1.27). Dzięki temu otrzymaliśmy algorytm wyznaczania sum trygonometrycznych, który może być poddany wektoryzacji oraz zrównolegleniu. Dodatkową optymalizację można przeprowadzić redukując liczbę odwołań do pamięci. Tak opracowana metoda może być z powodzeniem zastosowana do rozwiązania bardziej złożonego problemu, jakim jest numeryczne odwracanie transformanty Laplace'a.

Tabela 6.2
Czas działania i błąd względny wyniku dla oryginalnego algorytmu Talbota (t_1, e_1) i algorytmu wektorowo-równoległego (t_2, e_2) dla $F(s) = \operatorname{arc\,tg}(1/s)$, $f(t) = \sin(t)/t$, $t = 1000.0$ i różnych wartości n na komputerze Itanium 2

n	t_1	t_2	e_1	e_2
1565	0.6540E-03	0.4010E-03	0.23919E-06	0.23918E-06
1681	0.6990E-03	0.4220E-03	0.22268E-06	0.22269E-06
1729	0.7210E-03	0.4182E-03	0.21651E-06	0.21650E-06
1801	0.7510E-03	0.4539E-03	0.20786E-06	0.20787E-06
1851	0.7720E-03	0.4790E-03	0.20223E-06	0.20220E-06
1977	0.8230E-03	0.4809E-03	0.18935E-06	0.18935E-06
2029	0.8459E-03	0.5062E-03	0.18449E-06	0.18450E-06
2107	0.8800E-03	0.5372E-03	0.17767E-06	0.17769E-06
2161	0.9010E-03	0.5391E-03	0.17323E-06	0.17323E-06
2215	0.9232E-03	0.5641E-03	0.16901E-06	0.16901E-06

Tabela 6.3
Czas działania i błąd względny wyniku dla oryginalnego algorytmu Talbota (t_1, e_1) i algorytmu wektorowo-równoległego (t_2, e_2) dla $F(s) = \operatorname{arc\,tg}(1/s)$, $f(t) = \sin(t)/t$, $t = 10000.0$ i różnych wartości n na komputerze Itanium 2

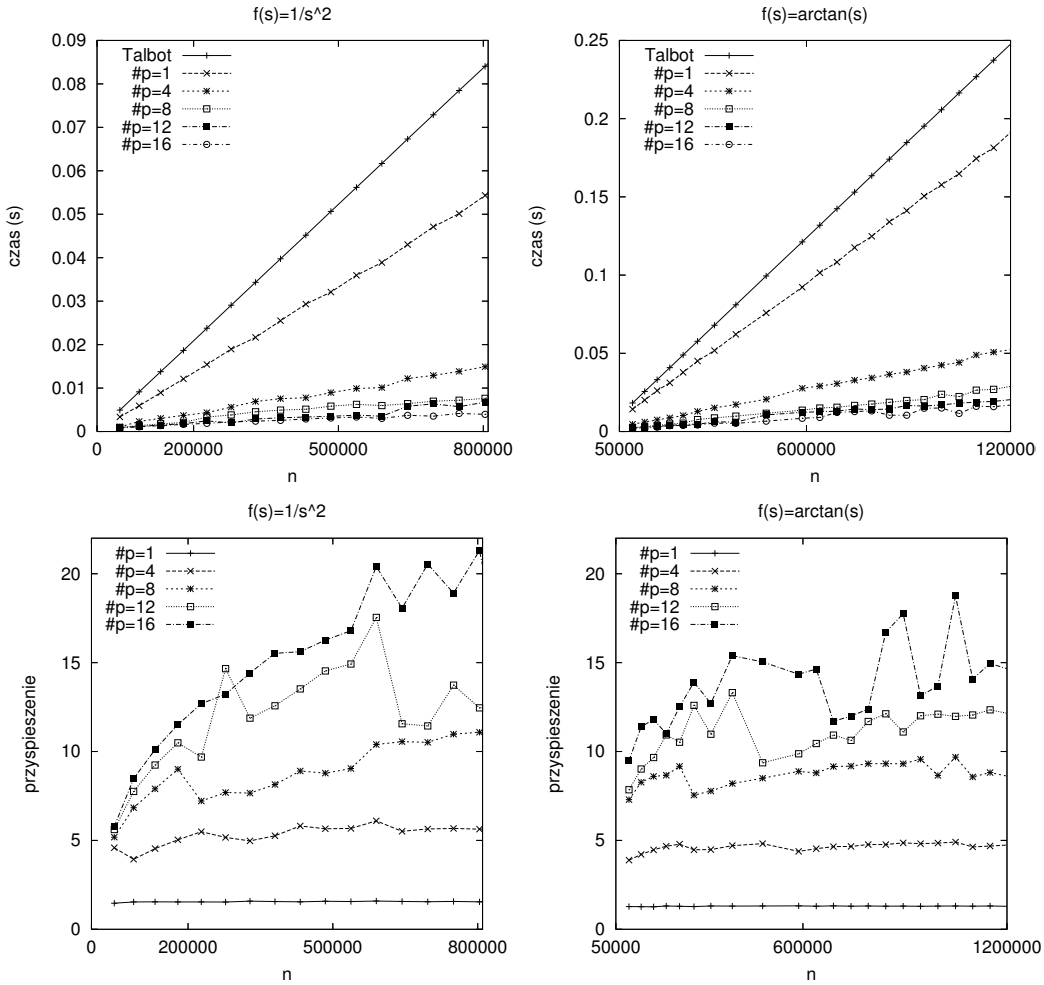
n	t_1	t_2	e_1	e_2
588305	0.2448E+00	0.1504E+00	0.93228E-09	0.11602E-08
639409	0.2660E+00	0.1607E+00	0.74041E-09	0.99562E-09
690241	0.2870E+00	0.1778E+00	0.77835E-09	0.70032E-09
792589	0.3296E+00	0.2042E+00	0.64559E-09	0.81140E-09
844291	0.3511E+00	0.2183E+00	0.62070E-09	0.71595E-09
946409	0.3935E+00	0.2390E+00	0.50365E-09	0.72116E-09
998211	0.4151E+00	0.2580E+00	0.43647E-09	0.88774E-09
1048435	0.4359E+00	0.2714E+00	0.36442E-09	0.37339E-09
1100497	0.4577E+00	0.2761E+00	0.44648E-09	0.28331E-09

Tabela 6.4
Czas działania i błąd względny wyniku dla oryginalnego algorytmu Talbota (t_1, e_1) i algorytmu wektorowo-równoległego (t_2, e_2) dla $F(s) = \operatorname{arc\,tg}(1/s)$, $f(t) = \sin(t)/t$, $t = 1000.0$ i różnych wartości n na komputerze Pentium 4

n	t_1	t_2	e_1	e_2
1565	0.6928E-03	0.5510E-03	0.23917E-06	0.23916E-06
1681	0.7119E-03	0.5870E-03	0.22268E-06	0.22270E-06
1729	0.7231E-03	0.6299E-03	0.20786E-06	0.20788E-06
1851	0.7741E-03	0.7241E-03	0.20224E-06	0.20221E-06
1977	0.8249E-03	0.6840E-03	0.18935E-06	0.18936E-06
2029	0.8581E-03	0.7038E-03	0.18449E-06	0.18450E-06
2107	0.9139E-03	0.7319E-03	0.17768E-06	0.17770E-06
2161	0.8988E-03	0.7489E-03	0.17323E-06	0.17322E-06
2215	0.9251E-03	0.7679E-03	0.16901E-06	0.16902E-06

Tabela 6.5
Czas działania i błąd względny wyniku dla oryginalnego algorytmu Talbota (t_1, e_1) i algorytmu wektorowo-równoległego (t_2, e_2) dla $F(s) = \operatorname{arc\,tg}(1/s)$, $f(t) = \sin(t)/t$, $t = 10000.0$ i różnych wartości n na komputerze Pentium 4

n	t_1	t_2	e_1	e_2
588305	0.2524E+00	0.2166E+00	0.97931E-09	0.98922E-09
639409	0.2733E+00	0.2392E+00	0.81797E-09	0.87523E-09
690241	0.2948E+00	0.2566E+00	0.93892E-09	0.85472E-09
742021	0.3171E+00	0.2764E+00	0.66004E-09	0.57862E-09
792589	0.3383E+00	0.2943E+00	0.71165E-09	0.83397E-09
844291	0.3614E+00	0.3155E+00	0.62383E-09	0.12623E-08
895987	0.3836E+00	0.3357E+00	0.65587E-09	0.34261E-08
946409	0.4048E+00	0.3540E+00	0.25500E-09	0.53398E-09
998211	0.4302E+00	0.3739E+00	0.52169E-09	0.52899E-09
1048435	0.4493E+00	0.3926E+00	0.50442E-09	0.46100E-09
1100497	0.4708E+00	0.4075E+00	0.46933E-09	0.49754E-09



Rys. 6.1. Czas działania oraz przyspieszenie wektorowo-równoległej wersji algorytmu Talbota względem oryginalnego algorytmu ($s = \lfloor\sqrt{2n}\rfloor$)

Fig. 6.1. Computation time and speedup of the vectorized parallel version of the Talbot algorithm relative to the original method ($s = \lfloor\sqrt{2n}\rfloor$)

7. Obliczanie wartości wielomianów

Rozważmy problem wyznaczania wartości wielomianu. Dla zadanej liczby rzeczywistej x należy wyznaczyć wartość $P(x)$, gdzie

$$P(x) = a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n. \tag{7.1}$$

Najprostszą metodą obliczenia wartości $P(x)$ jest zastosowanie schematu Hornera [103], który polega na wyznaczeniu ciągu wartości

$$p_k = \begin{cases} a_0 & \text{dla } k = 0 \\ a_k + x p_{k-1} & \text{dla } k = 1, \ldots, n, \end{cases} \tag{7.2}$$

i wtedy $P(x) = p_n$. Liczby $p_0, \ldots, p_n$ spełniają też równanie

$$a_0 y^n + a_1 y^{n-1} + \ldots + a_n = (p_0 y^{n-1} + p_1 y^{n-2} + \ldots + p_{n-1})(y - x) + p_n, \tag{7.3}$$

a zatem są współczynnikami ilorazu wielomianu $P(x)$ przez dwumian $y - x$.

Czasami zachodzi potrzeba wyznaczenia wartości wielomianów stosunkowo dużych stopni [9], stąd można znaleźć prace poświęcone równoległemu [10, 77, 81] bądź też wektorowemu [5, 123] wyznaczaniu wartości wielomianów. Jednakże algorytmy opisywane w cytowanych pracach charakteryzują się dużą liczbą operacji arytmetycznych, co czyni je mniej przydatnymi przy ograniczonej liczbie procesorów, bądź w przypadku procesorów, których mechanizmy wektorowe (przykładowo SSE) nie powodują bardzo dużego skrócenia czasu wykonania programu. W tym rozdziale pokażemy, że wyznaczenie wartości wielomianu może być zrealizowane algorytmem stanowiącym szczególny przypadek algorytmu 3.1 z rozdziału 3, mającym podobne własności numeryczne jak schemat Hornera.

7.1. Algorytm wykorzystujący operacje `AXPY` i `GER`

Wzór (7.2) stanowi szczególny przypadek (1.28), gdzie $m = 1$, $a_1 = x$, $f_k = a_{k-1}$ oraz $v_k = p_{k-1}$ dla $k = 1, \ldots, n + 1$. Zatem, po wyborze odpowiednich wartości r oraz s, gdzie

$r, s > 2$, można wyznaczyć szybkim algorytmem liczby $v_1, \ldots, v_{rs}$. Następnie, o ile $rs \neq n+1$, zastosować (7.2) dla wyznaczenia $v_{rs+1}, \ldots, v_{n+1}$. Niech zatem

$$\mathbf{v}_j = \left(v_{(j-1)s+1}, \ldots, v_{js}\right)^T, \ \mathbf{f}_j = \left(f_{(j-1)s+1}, \ldots, f_{js}\right)^T \in \mathbb{R}^s$$

oraz

$$L = \begin{pmatrix} 1 & & & \mathbf{0} \\ -x & \ddots & & \\ & \ddots & \ddots & \\ \mathbf{0} & & -x & 1 \end{pmatrix}, \ U = \begin{pmatrix} & & -x \\ & & \\ \mathbf{0} & & \end{pmatrix} \in \mathbb{R}^{s \times s}. \tag{7.4}$$

Wzór (2.2) przyjmie wówczas postać

$$\begin{cases} \mathbf{v}_1 = L^{-1}\mathbf{f}_1 \\ \mathbf{v}_j = L^{-1}\mathbf{f}_j - L^{-1}U\mathbf{v}_{j-1} \quad \text{dla } j = 2, \ldots, r. \end{cases} \tag{7.5}$$

Wykorzystując postać macierzy $U = -x\,\mathbf{e}_1\mathbf{e}_s^T$ zdefiniowanej wzorem (7.4) oraz kładąc $\mathbf{z}_j = L^{-1}\mathbf{f}_j$, otrzymujemy

$$\mathbf{v}_j = \mathbf{z}_j + L^{-1}(x\,\mathbf{e}_1\mathbf{e}_s^T)\mathbf{v}_{j-1} = \mathbf{z}_j + xv_{(j-1)s}L^{-1}\mathbf{e}_1.$$

Wzór (7.5) przyjmie wówczas postać

$$\begin{cases} \mathbf{v}_1 = \mathbf{z}_1 \\ \mathbf{v}_j = \mathbf{z}_j + \alpha_j\mathbf{y} \quad \text{dla } j = 2, \ldots, r \end{cases} \tag{7.6}$$

gdzie $\mathbf{y} = L^{-1}\mathbf{e}_1 = \left(1, x, x^2, \ldots, x^{s-1}\right)^T$ oraz dla $j = 2, \ldots, r$

$$\alpha_j = xv_{(j-1)s}. \tag{7.7}$$

Postępując podobnie jak w przypadku algorytmu 3.1, wyznaczymy rozwiązanie blokowego układu równań $LZ = F$, gdzie L zdefiniowano przez (7.4) oraz

$$Z = (\mathbf{z}_1, \ldots, \mathbf{z}_r, \mathbf{y}), \ F = (\mathbf{f}_1, \ldots, \mathbf{f}_r, \mathbf{e}_1) \in \mathbb{R}^{s \times (r+1)}, \tag{7.8}$$

stosując następujący wzór

$$Z_{k,*} = \begin{cases} F_{1,*} & \text{dla } k = 1 \\ F_{k,*} + xZ_{k-1,*} & \text{dla } k = 2, \ldots, s \end{cases} \tag{7.9}$$

Niech teraz $t = x^s$ oraz niech w_j, $j = 1, \ldots, r$, oznacza ostatnią składową odpowiedniego wektora $\mathbf{v}_j$. Wówczas, wykorzystując (7.6) oraz (7.7), otrzymujemy

$$\begin{cases} w_1 = z_{s,1} \\ w_j = z_{s,j} + t w_{j-1} \quad \text{dla } j = 2, \ldots, r. \end{cases} \tag{7.10}$$

Zauważmy, że gdy zachodzi potrzeba wyznaczenia wszystkich liczb v_i, wówczas można utworzyć następujący wektor

$$\mathbf{u} = (\alpha_2, \ldots, \alpha_r)^T \in \mathbb{R}^{r-1},$$

a następnie obliczyć $s - 1$ brakujących pierwszych składowych każdego wektora $\mathbf{v}_j$, używając pojedynczego odwołania do operacji `GER` z BLAS-u poziomu drugiego, mianowicie wykonując operację

$$V_{1:s-1,2:r} \longleftarrow Z_{1:s-1,2:r} + \left(1, x, \ldots, x^{s-2}\right)^T \mathbf{u}^T. \tag{7.11}$$

7.2. Analiza własności numerycznych algorytmu

Zajmiemy się teraz analizą własności numerycznych algorytmu przedstawionego w punkcie 7.1. Zakładamy, że algorytm jest realizowany w arytmetyce zmiennopozycyjnej [52]. Niech u oznacza precyzję arytmetyki, a symbol $\square$ operację arytmetyczną, przy czym $\square \in \{+, -, *, /\}$. Obliczona wartość wyrażenia $x \square y$ wynosi

$$fl(x \square y) = (x \square y)(1 + \varepsilon), \qquad fl(x \square y) = \frac{(x \square y)}{(1 + \delta)},$$

gdzie błędy zaokrągleń spełniają ograniczenia

$$|\varepsilon| \leq u, \qquad |\delta| \leq u.$$

Dla uproszczenia notacji wykorzystamy pojęcie licznika błędu względnego (ang. *relative error counter*) wprowadzone przez Stewarta [102]. Niech symbol $<k>$ oznacza

$$<k> = \prod_{i=1}^{k} (1 + \delta_i)^{\rho_i}, \tag{7.12}$$

przy czym $\rho_i = \pm 1$ oraz $|\delta_i| \leq u$. Można łatwo wykazać [52], że prawdziwe są następujące własności licznika błędu względnego

$$<j><k> = <j + k> \tag{7.13}$$

oraz

$$\frac{<j>}{<k>} =<j+k> . \tag{7.14}$$

Prostą konsekwencją (7.13) jest

$$<j>^k=<kj> . \tag{7.15}$$

Dodatkowo [52, Lemat 3.1], jeśli założymy, że dla pewnej ustalonej wartości całkowitej k zachodzi $ku < 1$, wówczas również

$$<k>= 1 + \theta_k, \qquad |\theta_k| \leq \frac{ku}{1-ku} =: \gamma_k. \tag{7.16}$$

W przypadku schematu Hornera (7.2) można wykazać (na przykład książka [52]), że dla wielomianów stopnia $n \geq 2$, obliczona wartość wielomianu (7.1) w punkcie x spełnia następującą równość

$$\widehat{P}(x) = \widehat{a}_0 x^n + \widehat{a}_1 x^{n-1} + \ldots + \widehat{a}_{n-1} x + \widehat{a}_n, \tag{7.17}$$

gdzie

$$\widehat{a}_k = \begin{cases} a_0 <2n> & \text{dla } k = 0 \\ a_k <2(n-k)+1> & \text{dla } k = 1, \ldots, n. \end{cases} \tag{7.18}$$

Zatem, obliczona wartość jest dokładną wartością wielomianu o nieco zaburzonych współczynnikach, w punkcie x. Zaburzenie względne każdego współczynnika wynosi co najwyżej γ_{2n}. Oznacza to [52, podrozdział 5.1], że błąd względny obliczonej wartości wielomianu w punkcie x spełnia oszacowanie

$$\frac{|P(x) - \widehat{P}(x)|}{|P(x)|} \leq \gamma_{2n} \frac{|\widetilde{P}(|x|)|}{|P(x)|}, \tag{7.19}$$

gdzie wielomian $\widetilde{P}(x)$ zdefiniowany jest następującym wzorem

$$\widetilde{P}(x) = \sum_{i=0}^{n} |a_{n-i}| x^i.$$

Opisany w punkcie 7.1 algorytm wyznaczania wartości wielomianu oraz znajdowania współczynników $p_0, \ldots, p_{n-1}$ zdefiniowanych wzorem (7.3) można scharakteryzować następującym twierdzeniem.

Twierdzenie 7.1. *Niech stopień n wielomianu $P(x)$ danego wzorem (7.1) spełnia równość $rs = n + 1$, gdzie $r, s \in \mathbb{Z}$ oraz $r, s \geq 2$. Wartość wielomianu obliczona za pomocą wzorów (7.9) – (7.10) jest dokładną wartością wielomianu $P(x)$ o nieco zaburzonych współczynnikach, przy czym zaburzenie każdego współczynnika jest nie większe niż*

$$\frac{(n + r + s - 2)u}{1 - (n + r + s - 2)u}. \tag{7.20}$$

Dowód. Zauważmy, że wyznaczenie ostatnich składowych wektorów $\mathbf{z}_j$, $j = 1, \ldots, r$, oraz $\mathbf{y}$, a dokładnie liczb $z_{s,j}$ oraz x^{s-1} może być potraktowane jako obliczenie wartości wielomianów

$$\sum_{k=0}^{s-1} a_{(j-1)s+k}\, x^{s-1-k}.$$

Zatem, stosując (7.17) i (7.18), otrzymujemy

$$fl(z_{s,j}) = \widehat{z}_{s,j} = \widehat{a}_{(j-1)s} x^{s-1} + \widehat{a}_{(j-1)s+1} x^{s-2} + \ldots + \widehat{a}_{js-2} x + \widehat{a}_{js-1}, \tag{7.21}$$

przy czym

$$\widehat{a}_{(j-1)s+k} = \begin{cases} a_{(j-1)s} <2(s-1)> & \text{dla } k = 0 \\ a_{(j-1)s+k} <2(s-k)-1> & \text{dla } k = 1, \ldots, s-1, \end{cases} \tag{7.22}$$

a w przypadku ostatniej składowej wektora $\mathbf{y}$ przemnożonej przez x zachodzi równość

$$\widehat{t} = x^s <s-1> . \tag{7.23}$$

Zastosowanie wzoru (7.10) dla wyznaczenia liczby w_r równej $P(x)$ jest równoważne użyciu schematu Hornera dla obliczenia wartości następującego wielomianu

$$Q(t) = \sum_{j=0}^{r-1} z_{s,r-j}\, t^j. \tag{7.24}$$

Oczywiście, wyznaczone wcześniej współczynniki $z_{s,r-j}$ oraz argument t są obarczone błędem, a zatem zamiast wartości wielomianu $Q(t)$ obliczamy wartość

$$\widehat{Q}(t) = \sum_{j=0}^{r-1} \widehat{z}_{s,r-j}\, \widehat{t}^j, \tag{7.25}$$

gdzie współczynniki $\widehat{z}_{s,r-j}$ oraz argument $\widehat{t}$ spełniają (7.21) oraz (7.23). Stąd, wykorzystując ponownie wzory (7.17) i (7.18), otrzymujemy

$$\begin{aligned} \widehat{P}(x) = fl(\widehat{Q}(t)) &= \widehat{z}_{s,1} <2(r-1)> \widehat{t}^{r-1} + \sum_{j=0}^{r-2} \widehat{z}_{s,r-j} <2j+1> \widehat{t}^j \\ &= \sum_{j=0}^{r-1} \widetilde{z}_{s,r-j}\, t^j. \end{aligned} \tag{7.26}$$

Uwzględniając (7.26) oraz (7.23) i własność (7.15), otrzymujemy

$$\widetilde{z}_{s,j} = \begin{cases} \widehat{z}_{s,1} <2(r-1)+(r-1)(s-1)> & \text{dla } j=1 \\ \widehat{z}_{s,j} <2(r-j)+1+(r-j)(s-1)> & \text{dla } j=2,\ldots,r-1 \\ \widehat{z}_{s,r} <1> & \text{dla } j=r \end{cases}$$

czyli ostatecznie

$$\widetilde{z}_{s,j} = \begin{cases} \widehat{z}_{s,1} <(r-1)(s+1)> & \text{dla } j=1 \\ \widehat{z}_{s,j} <(r-j)(s+1)+1> & \text{dla } j=2,\ldots,r-1 \\ \widehat{z}_{s,r} <1> & \text{dla } j=r. \end{cases} \tag{7.27}$$

Zatem, wstawiając dane wzorem (7.21) obliczone wartości $\widehat{z}_{s,j}$ do (7.27), przenosimy zaburzenia na współczynniki a_k. Stąd wykorzystując wzór (7.26) wnosimy, że obliczona wartość $\widehat{P}(x)$ jest dokładną wartością wielomianu o zaburzonych współczynnikach, to znaczy

$$\widehat{P}(x) = \widetilde{a}_0 x^n + \widetilde{a}_1 x^{n-1} + \ldots + \widetilde{a}_{n-1} x + \widetilde{a}_n.$$

Największa wartość zaburzenia będzie występować dla współczynnika a_0. Korzystając ze wzoru (7.27) dla $j=1$ oraz wzoru (7.21) dla $j=1$ i $k=0$, otrzymujemy

$$\widetilde{a}_0 = a_0 <2(s-1)+(r-1)(s+1)>= a_0 <n+r+s-2> . \tag{7.28}$$

Ostatecznie, wykorzystując daną wzorem (7.16) definicję wielkości γ_k, wnioskujemy, że maksymalne zaburzenie współczynników wielomianu (7.28) jest nie większe niż określone wzorem (7.20). ■

Zauważmy, że założenie $rs = n+1$ przyjęte w powyższym twierdzeniu może być w dalszym ciągu pominięte. Po wyznaczeniu w_r spełniona jest równość

$$\widehat{w}_r = \widetilde{a}_0 x^{rs-1} + \widetilde{a}_1 x^{rs-1} + \ldots + \widetilde{a}_{rs-2} x + \widetilde{a}_{rs-1}.$$

Gdy następnie zastosujemy wzór (7.2) dla wyznaczenia liczb $p_{rs}, \ldots, p_n$, wówczas zgodnie ze wzorem (7.18) zaburzenia współczynników a_i wielomianu (7.1) wzrosną o $2(n-rs+1)$. Co więcej, gdy przyjmiemy, że $r = O(\sqrt{n})$ i $s = O(\sqrt{n})$, wówczas na mocy (7.19) błąd względny obliczonej algorytmem (7.9) – (7.10) wartości wielomianu spełnia oszacowanie

$$\frac{|P(x)-\widehat{P}(x)|}{|P(x)|} \le \gamma_{n+O(\sqrt{n})} \frac{|\widetilde{P}(|x|)|}{|P(x)|}. \tag{7.29}$$

Zatem, algorytm oparty na wzorach (7.9) – (7.10) charakteryzuje się podobnym oszacowaniem błędu względnego jak w przypadku schematu Hornera.

Nie będziemy tutaj prezentować wyników dotyczących czasu wykonania algorytmu na różnych architekturach komputerowych, gdyż jest on szczególnym przypadkiem algorytmów 2.1 i 3.1. Zauważmy jednak, że opisany algorytm obliczania wartości wielomianów ma kilka ważnych zalet. Po pierwsze, liczba operacji zmiennopozycyjnych nieznacznie przekracza liczbę operacji w schemacie Hornera, która jest optymalna. Algorytm charakteryzuje się podobnymi jak schemat Hornera własnościami numerycznymi. Przy jego implementacji mogą być użyte mechanizmy wektorowe procesorów. Dodatkowo, z uwagi na jego macierzowy charakter, można zastosować opisany w podrozdziale 1.2.3 nowy sposób reprezentacji macierzy. Możliwe jest również jego łatwe zrównoleglenie (w sposób podobny jak przy przejściu od algorytmu 3.1 do 3.2).

8. Nowy sposób rozmieszczenia macierzy trójkątnych i symetrycznych

W niniejszym rozdziale zajmiemy się problemem rozproszonego rozwiązywania układów równań liniowych o macierzach trójkątnych dla wielu prawych stron, który należy do podstawowych zadań obliczeniowych [18, 51, 76, 98]. Podamy nowy sposób rozmieszczenia danych zadania w pamięciach poszczególnych procesów. Pokażemy, że jest on oszczędniejszy (wykorzystuje mniej pamięci operacyjnej) niż stosowane powszechnie rozwiązanie blokowo-cykliczne oraz w wielu przepadkach umożliwia konstrukcję szybszych algorytmów rozwiązywania problemu. Sposób opiera się na metodzie zastosowanej przez nas do reprezentacji macierzy symetrycznych na komputerach z pamięcią ortogonalną [105, 8].

8.1. Blokowe algorytmy rozwiązywania trójkątnych układów równań liniowych

W punkcie 1.6.2 przedstawiliśmy dwa podejścia do problemu rozwiązania układów równań liniowych o macierzach trójkątnych, który stanowi uogólnienie rozważanego w poprzednich rozdziałach problemu wyznaczania rozwiązania liniowych równań rekurencyjnych. W praktyce, często rozważa się problem rozwiązywania takich układów równań liniowych, ale o wielu prawych stronach [32]. Otrzymujemy wówczas układ równań $AX = B$ następującej postaci

$$\begin{pmatrix} a_{11} & & & 0 \\ a_{21} & a_{22} & & \\ \vdots & \vdots & \ddots & \\ a_{m1} & \dots & \dots & a_{mm} \end{pmatrix} \begin{pmatrix} x_{11} & \dots & x_{1n} \\ x_{21} & \dots & x_{2n} \\ \vdots & & \vdots \\ x_{m1} & \dots & x_{mn} \end{pmatrix} = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ b_{21} & \dots & b_{2n} \\ \vdots & & \vdots \\ b_{m1} & \dots & b_{mn} \end{pmatrix}. \tag{8.1}$$

Szybkie wyznaczenie rozwiązania układu (8.1) może być zrealizowane przy zastosowaniu algorytmów blokowych [32]. W takim przypadku układ (8.1) można zapisać jako

$$\begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ \vdots & \vdots & \ddots & \\ A_{M1} & \dots & \dots & A_{MM} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_M \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_M \end{pmatrix}, \tag{8.2}$$

gdzie wszystkie A_{ij}, X_i, B_i są macierzami (blokami) odpowiednich rozmiarów. Wyznaczenie rozwiązania układu (8.2) może być zrealizowane za pomocą blokowych wersji algorytmów opartych na wzorach (1.18) oraz (1.20). W praktyce (biblioteka LAPACK), wykorzystywana jest blokowa wersja algorytmu (1.20), którą opisujemy jako następujący algorytm 8.1.

Algorytm 8.1. Blokowe wyznaczanie rozwiązania układu równań liniowych (8.2).

Wejście: nieosobliwa macierz kwadratowa A, macierz B

Wyjście: $B = A^{-1}B$

1: **for** $i = 1$ to M **do**
2: $\quad B_i \leftarrow A_{ii}^{-1}Bi$ {operacja `TRSM`}
3: $\quad$ **for** $j = i + 1$ to M **do**
4: $\quad\quad B_j \leftarrow B_j - A_{ji}B_i$ {operacja `GEMM`}
5: $\quad$ **end for**
6: **end for**

Zauważmy, że wszystkie operacje składające się na algorytm 8.1 są zaczerpnięte z biblioteki BLAS poziomu 3. W praktyce oznacza to, że algorytm może być znacznie szybszy, niż omówione w punkcie 1.6.2 algorytmy wykorzystujące wzory (1.18) oraz (1.20). Oczywiście, w przypadku $n = 1$, operacje `TRSM` i `GEMM` zostaną zastąpione operacjami `TRSV` i `GEMV` z biblioteki BLAS poziomu 2.

W przypadku rozwiązywania układów równań liniowych w środowisku rozproszonym algorytmami z biblioteki ScaLAPACK [12] stosuje się organizację procesów w postaci dwuwymiarowej siatki $P \times Q$, gdzie każdy proces jest jednoznacznie identyfikowany przez parę współrzędnych (i, j), $i = 0, \ldots, P - 1$, $j = 0, \ldots, Q - 1$ [38]. Macierze prostokątne są rozmieszczane w sposób blokowo-cykliczny, gdzie lokalizację każdego bloku A_{ij} o zadanym rozmiarze $m_b \times m_b$ opisuje następujące odwzorowanie [12]:

$$loc(A_{ij}) = ((i - 1) \textbf{ mod } P, (j - 1) \textbf{ mod } Q) .$$

Rysunek 8.1 pokazuje przykład blokowo-cyklicznego rozmieszczenia macierzy prostokątnej na siatce procesów 2×2. W przypadku rozmieszczania macierzy trójkątnych (dolnych lub górnych) lub symetrycznych alokowany jest tylko jeden trójkąt (dolny lub górny). Na rysunku 8.2 pokazujemy przykładowe rozmieszczenie niezerowych bloków A_{ij}, $1 \leq j \leq i \leq M$, macierzy dolnotrójkątnej. W takim przypadku każdy proces alokuje tablicę dwuwymiarową, której duża część nie jest wykorzystana (nawet przeszło 50% – rysunek 8.2). Wybór rozmiaru bloku jest kompromisem pomiędzy efektywniejszym wykorzystaniem pamięci (dla mniejszych rozmiarów) a szybkością działania (dla większych rozmiarów). W praktyce zaleca się przyjęcie

rozmiaru bloku $m_b = 64$ lub większego dla komputerów z pamięcią wspólną bądź większym rozmiarem pamięci podręcznej oraz rozmiaru sieci $P = Q$ [12].

A_{11}	A_{13}	A_{15}	A_{17}	A_{12}	A_{14}	A_{16}	A_{18}
A_{31}	A_{33}	A_{35}	A_{37}	A_{32}	A_{34}	A_{36}	A_{38}
A_{51}	A_{53}	A_{55}	A_{57}	A_{52}	A_{54}	A_{56}	A_{58}
A_{71}	A_{73}	A_{75}	A_{77}	A_{72}	A_{74}	A_{76}	A_{78}
A_{21}	A_{23}	A_{25}	A_{27}	A_{22}	A_{24}	A_{26}	A_{28}
A_{41}	A_{43}	A_{45}	A_{47}	A_{42}	A_{44}	A_{46}	A_{48}
A_{61}	A_{63}	A_{65}	A_{67}	A_{62}	A_{64}	A_{66}	A_{68}
A_{81}	A_{83}	A_{85}	A_{87}	A_{82}	A_{84}	A_{86}	A_{88}

Rys. 8.1. Blokowo-cykliczny sposób rozmieszczenia macierzy prostokątnej na sieci procesów 2×2
Fig. 8.1. Block-cyclic distribution of a rectangular matrix over a 2×2 process grid

Specjalnym przypadkiem opisanej wyżej dwuwymiarowej sieci jest zalecana w przypadku mniejszej liczby dostępnych procesorów, bądź też rozwiązywania układów równań z pojedynczą prawą stroną, jednowymiarowa tablica $P \times 1$, gdzie poszczególne wiersze blokowe są rozmieszczane cyklicznie w pamięciach poszczególnych procesów [12]. Rysunek 8.3 pokazuje sposób rozmieszczenia bloków układu (8.2). Zauważmy, że w tym przypadku niewykorzystanie pamięci jest mniejsze niż w przypadku sieci dwuwymiarowej, jednak poszczególne procesy posiadają zróżnicowaną liczbę bloków (na rysunku 8.3 pierwszy ma sześć bloków, ostatni dwukrotnie więcej). Algorytm rozproszony rozwiązywania układu (8.2) wykorzystujący rozkład blokowo-cykliczny stanowi wersję SPMD algorytmu 8.1. Szczegóły dotyczące jego implementacji można znaleźć w podręczniku [12].

8.2. Nowy sposób rozmieszczania danych

W literaturze omawiane są modyfikacje reprezentacji macierzy trójkątnych i symetrycznych poprawiające efektywność wykorzystania pamięci [26, 6, 50]. Bazują jednak bezpośrednio na rozkładzie blokowo-cyklicznym, a zatem nie będą wpływać na zwiększenie szybkości działania algorytmów z biblioteki ScaLAPACK. Przedstawimy teraz nowy sposób rozmieszczenia bloków macierzy trójkątnej, który znacznie poprawia wykorzystanie pamięci oraz dzięki możliwości operowania na dużych blokach przyczynia się do konstrukcji szybszych algorytmów.

A_{11}							
A_{31}	A_{33}			A_{32}			
A_{51}	A_{53}	A_{55}		A_{52}	A_{54}		
A_{71}	A_{73}	A_{75}	A_{77}	A_{72}	A_{74}	A_{76}	
A_{21}				A_{22}			
A_{41}	A_{43}			A_{42}	A_{44}		
A_{61}	A_{63}	A_{65}		A_{62}	A_{64}	A_{66}	
A_{81}	A_{83}	A_{85}	A_{87}	A_{82}	A_{84}	A_{86}	A_{88}

Rys. 8.2. Blokowo-cykliczny sposób rozmieszczenia macierzy dolnotrójkątnej na sieci procesów 2×2

Fig. 8.2. Block-cyclic distribution of a triangular matrix over a 2×2 process grid

A_{11}				A_{51}	A_{52}	A_{53}	A_{54}	A_{55}			
A_{21}	A_{22}			A_{61}	A_{62}	A_{63}	A_{64}	A_{65}	A_{66}		
A_{31}	A_{32}	A_{33}		A_{71}	A_{72}	A_{73}	A_{74}	A_{75}	A_{76}	A_{77}	
A_{41}	A_{42}	A_{43}	A_{44}	A_{81}	A_{82}	A_{83}	A_{84}	A_{85}	A_{86}	A_{87}	A_{88}

B_1	B_5
B_2	B_6
B_3	B_7
B_4	B_8

Rys. 8.3. Blokowo-cykliczny sposób rozmieszczenia macierzy dolnotrójkątnej A i macierzy prostokątnej B na sieci procesów 4×1

Fig. 8.3. Block-cyclic distribution of a triangular matrix A and rectangular matrix B over a 4×1 process grid

Rozważmy układ równań o postaci (8.2). Niech $M = 2P$. Bloki A_{ij} są odwzorowywane na stosowaną w tym rozwiązaniu sieć $P \times 1$ przy wykorzystaniu następującego odwzorowania:

$$loc(A_{ij}) = \begin{cases} (i-1,0) & \text{dla } i = 1,\ldots,P \\ (2P-i,0) & \text{dla } i = P+1,\ldots,2P \end{cases} \tag{8.3}$$

Rysunek 8.4 pokazuje przykładowe rozmieszczenie dolnotrójkątnej macierzy A oraz prostokątnej macierzy B na siatce czterech procesów według wzoru (8.3). Zauważmy, że każdy proces posiada jednakową liczbę bloków macierzy A oraz każdy wiersz blokowy znajduje się w całości w pamięci lokalnej tylko jednego procesu. Na rysunku 8.5 pokazano wykorzystanie pamięci w lokalnych strukturach danych do przechowywania bloków macierzy A. Jak można zaobserwować, tylko niewielka ilość miejsca pozostaje niewykorzystana, a zatem istnieje potrzeba alokacji mniejszej ilości pamięci niż w przypadku rozkładu blokowo-cyklicznego. Będzie to ilość nie większa niż rozmiar jednego bloku. Każdy proces przechowuje dokładnie $2P + 1$ bloków macierzy trójkątnej i w przybliżeniu tylko jeden blok jest niewykorzystany. Zatem około

$$\frac{100}{2P+1}\%$$

miejsca w pamięci jest tracone. Przykładowo, dla $P = 4, 8, 16$, otrzymujemy odpowiednio 11%, 6%, 3% niewykorzystanej pamięci.

Zauważmy również, że jeśli dany proces posiada wiersz bloków o numerze i, $1 \leq i \leq P$, to posiada również wiersz bloków o numerze $2P - i + 1$. Podobnie na mocy wzoru (8.3) wnioskujemy, że proces $(i, 0)$ posiada wiersze blokowe o numerach $i - 1$ oraz $2P - i$. Liczba procesów P determinuje wybór rozmiaru bloku. Oczywiście liczba $2P$ nie musi być dzielnikiem m, czyli liczby wierszy macierzy A. W takim przypadku przyjmujemy podstawowy rozmiar bloków A_{ij}, $j < M$, jako $m_b \times m_b$, gdzie

$$m_b = \lfloor m/(2P) \rfloor.$$

Pozostałe bloki $A_{i,M}$, $i < M$ będą miały rozmiary określone następującym wzorem

$$(m - (P-1)m_b) \times m_b,$$

a blok A_{MM} rozmiar

$$(m - (P-1)m_b) \times (m - (P-1)m_b).$$

Bloki B_j, $j < M$ będą miały rozmiar $m_b \times n$, a blok B_M rozmiar określony wzorem

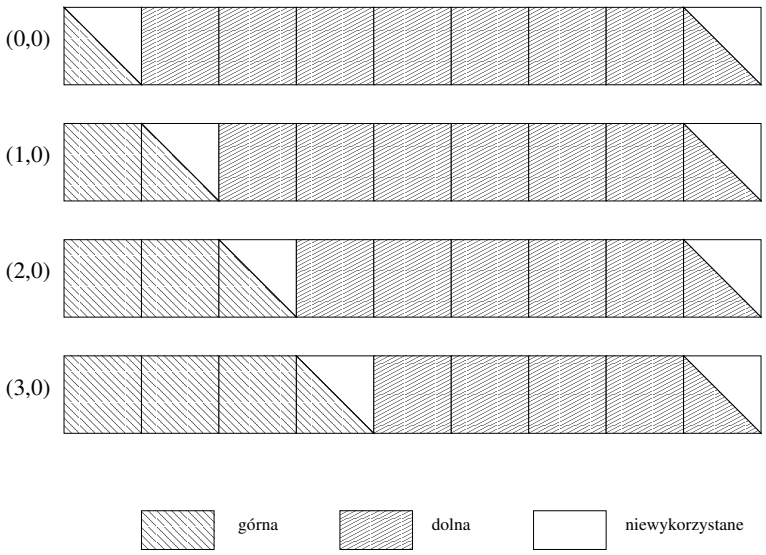
$$(m - (P-1)m_b) \times n.$$

A_{11}				A_{81}	A_{82}	A_{83}	A_{84}	A_{85}	A_{86}	A_{87}	A_{88}	
A_{21}	A_{22}			A_{71}	A_{72}	A_{73}	A_{74}	A_{75}	A_{76}	A_{77}		
A_{31}	A_{32}	A_{33}		A_{61}	A_{62}	A_{63}	A_{64}	A_{65}	A_{66}			
A_{41}	A_{42}	A_{43}	A_{44}	A_{51}	A_{52}	A_{53}	A_{54}	A_{55}				

B_1	B_8
B_2	B_7
B_3	B_6
B_4	B_5

Rys. 8.4. Nowy sposób rozmieszczenia bloków macierzy dolnotrójkątnej A i macierzy prostokątnej A i B na sieci procesów 4×1

Fig. 8.4. New distribution of a triangular matrix A and rectangular matrix B over a 4×1 process grid



Rys. 8.5. Lokalne struktury danych dla nowego sposobu rozmieszczenia części macierzy dolnotrójkątnej na sieci procesów 4×1.

Fig. 8.5. Local data structures for the new distribution of triangular matrices over a 4×1 process grid

Algorytm 8.2. Blokowe wyznaczanie rozwiązania układu równań liniowych (8.2) dla procesu o współrzędnych $(i, 0)$, $i = 0, \ldots, P - 1$.

Wejście: nieosobliwa kwadratowa macierz dolnotrójkątna A oraz macierz B podzielona na bloki według (8.3)

Wyjście: $B = A^{-1}B$

1: $M \leftarrow 2P$
2: **for** $j = 1$ to P **do**
3: **if** $loc(A_{jj}) = (i, 0)$ **then**
4: $B_j \leftarrow A_{jj}^{-1} B_j$ {operacja `TRSM`}
5: **send** B_j **to all** $\{(k, 0) : \ k = 0, \ldots, P - 1 \wedge k \neq i\}$
6: $B_{2P-j+1} \leftarrow B_{2P-j+1} - A_{2P-j+1,j} B_j$ {operacja `GEMM`}
7: **else**
8: **receive** X **from** $(loc(A_{jj}), 0)$
9: **if** $i > loc(A_{jj})$ **then**
10: $B_{i+1} \leftarrow B_{i+1} - A_{i+1,j} X$ {operacja `GEMM`}
11: **end if**
12: $B_{2P-i} \leftarrow B_{2P-i} - A_{2P-i,j} X$ {operacja `GEMM`}
13: **end if**
14: **end for**
15: **for** $j = P + 1$ to $M - 1$ **do**
16: **if** $loc(A_{jj}) = (i, 0)$ **then**
17: $B_j \leftarrow A_{jj}^{-1} B_j$ {operacja `TRSM`}
18: **send** B_j **to all** $\{(k, 0) : \ k = 0, \ldots, i - 1\}$
19: **else**
20: **if** $i < loc(A_{jj})$ **then**
21: **receive** X **from** $(loc(A_{jj}), 0)$
22: $B_{i+1} \leftarrow B_{i+1} - A_{i+1,j} X$ {operacja `GEMM`}
23: **end if**
24: **end if**
25: **end for**
26: **if** $loc(A_{MM}) = (i, 0)$ **then**
27: $B_M \leftarrow A_{MM}^{-1} B_M$ {operacja `TRSM`}
28: **end if**

Algorytm 8.2 wyznacza rozwiązanie blokowego układu równań (8.2) dla opisanego wyżej sposobu rozmieszczenia danych. W liniach 2-14 przetwarzana jest górna część trójkąta. Proces posiadający aktualny blok na przekątnej macierzy A rozwiązuje układ równań liniowych (linia 4), następnie rozsyła rozwiązanie do pozostałych procesów (linia 5), po czym aktualizuje posiadany blok macierzy B. Z kolei, każdy inny proces odbiera rozwiązanie od procesu wysyłającego, po czym aktualizuje swoje bloki macierzy B (blok górny tylko gdy jego numer jest mniejszy od numeru procesu wysyłającego). Przetwarzanie części dolnej (linie 15-28) jest realizowane podobnie, przy czym rozsyłanie wykonywane jest tylko do procesów posiadających bloki do przetworzenia. Rozwiązanie ostatniego układu równań (linia 27) jest wykorzystywane tylko do aktualizacji jednego bloku macierzy B, zatem nie występuje przesyłanie danych do innych procesów. Podkreślmy również, że podobnie jak w przypadku algorytmu 8.1, gdy $n = 1$, wówczas operacje `TRSM` i `GEMM` zostaną zastąpione przez `TRSV` i `GEMV`.

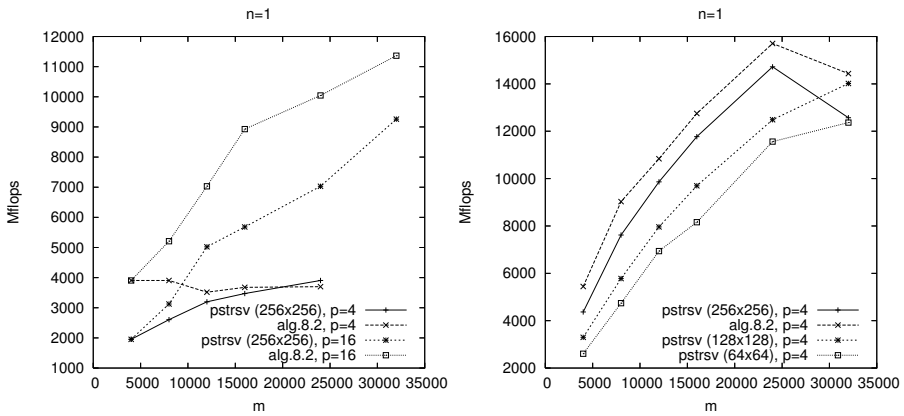
Na koniec zauważmy, że w modelu BSP koszt opisanego wyżej algorytm charakteryzuje się względnie małym współczynnikiem przy wielkości l (podrozdział 1.6.4). Istotnie, liczba superkroków w algorytmie 8.2 oraz algorytmie wykorzystującym rozkład blokowo-cykliczny wynosi M. Zatem, składnik uwzględniający parametr l będzie w obu algorytmach wynosił Ml. W przypadku algorytmu 8.2 będzie to zatem $2Pl$. W algorytmie wykorzystującym rozkład blokowo-cykliczny przyjmuje się zwykle rozmiar bloku równy 64×64, stąd dla dostatecznie dużych wartości m współczynnik przy l będzie na ogół większy niż $2P$. Oznacza to, że w przypadku bardzo dużych wartości parametru l, co ma miejsce w rozproszonych geograficznie klastrach, algorytm 8.2 może być znacznie szybszy.

8.3. Wyniki eksperymentów

Algorytm 8.2 został zaimplementowany w języku Fortran i uruchomiony na klastrze procesorów Itanium 2 oraz komputerze Cray X1. Rysunek 8.6 przedstawia porównanie wydajności osiąganej dla nowej metody (algorytm 8.2) z wydajnością dla algorytmu realizowanego przez podprogram `PSTRSV` z biblioteki PBLAS. Oba algorytmy charakteryzują się identyczną liczbą operacji zmiennopozycyjnych, stąd wydajność (Mflops) przekłada się bezpośrednio na czas działania algorytmów Na klastrze Itanium podprogram `PSTRSV` osiąga największą szybkość dla rozmiaru bloku 256×256. Nowa metoda jest szybsza na czterech i szesnastu procesorach. Podobnie na komputerze Cray X1.

Rysunek 8.7 pokazuje wydajność podprogramu `PSTRSM` przy optymalnym rozmiarze bloku 64×64 oraz algorytmu 8.2 na klastrze Itanium. Dla mniejszej liczby procesorów i rozmiarów problemu nowa metoda jest porównywalna z `PSTRSM`, jednak przy liczbie procesorów równej

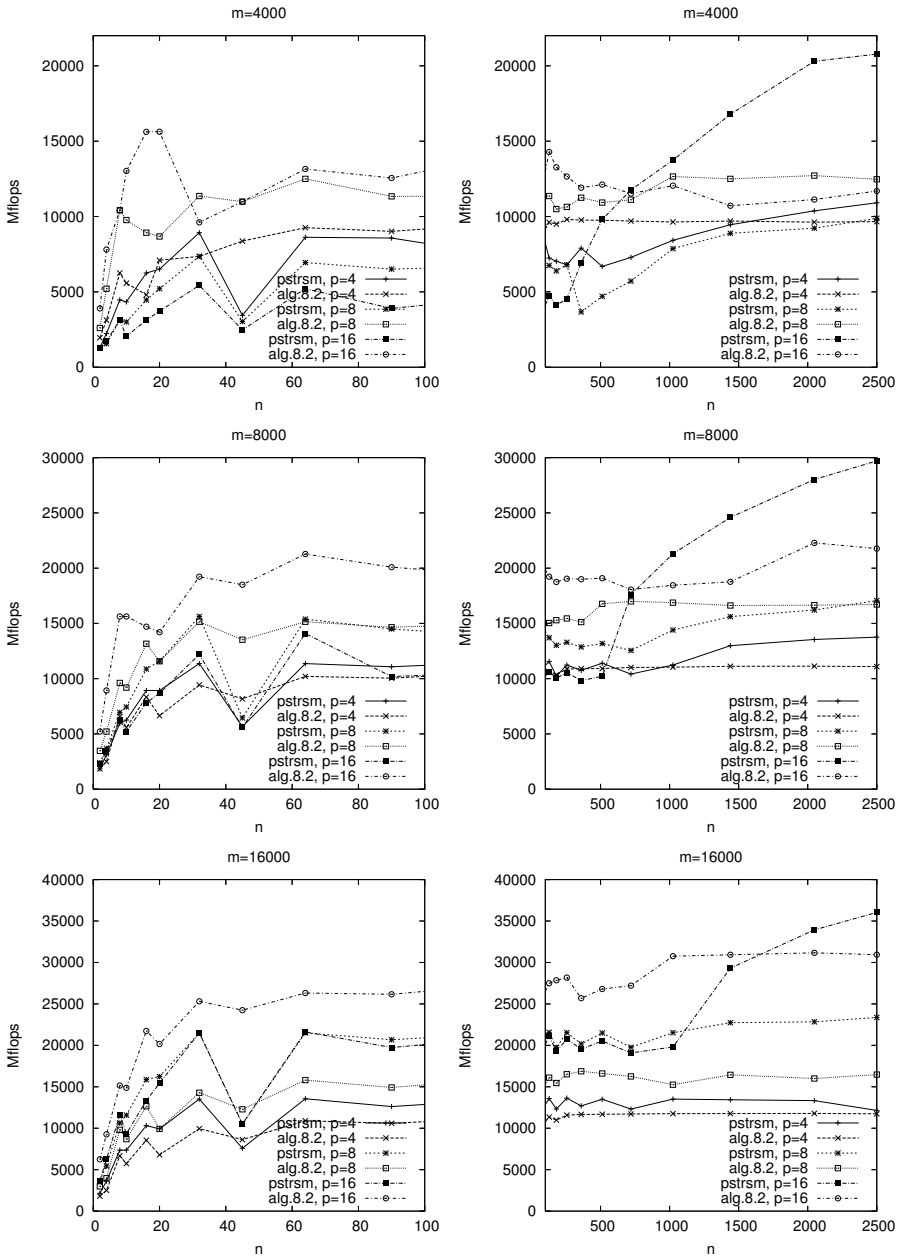
16 oraz większych rozmiarach problemu, wydajność nowego algorytmu jest istotnie większa. Rysunek 8.8 pokazuje analogiczne porównanie algorytmów na komputerze Cray X1 przy optymalnym rozmiarze bloków 128×128. Dla mniejszych wartości n wydajność nowej metody jest większa przy mniejszym rozmiarze bloku oraz porównywalna przy większym. Przy zwiększonej liczbie prawych stron (wartość n) można zauważyć znaczną przewagę nowej metody.



Rys. 8.6. Wydajność klastra Itanium 2 (strona lewa) oraz komputera Cray X1 (strona prawa) dla podprogramu `PSTRSV` oraz algorytmu 8.2 dla różnych m oraz $n = 1$

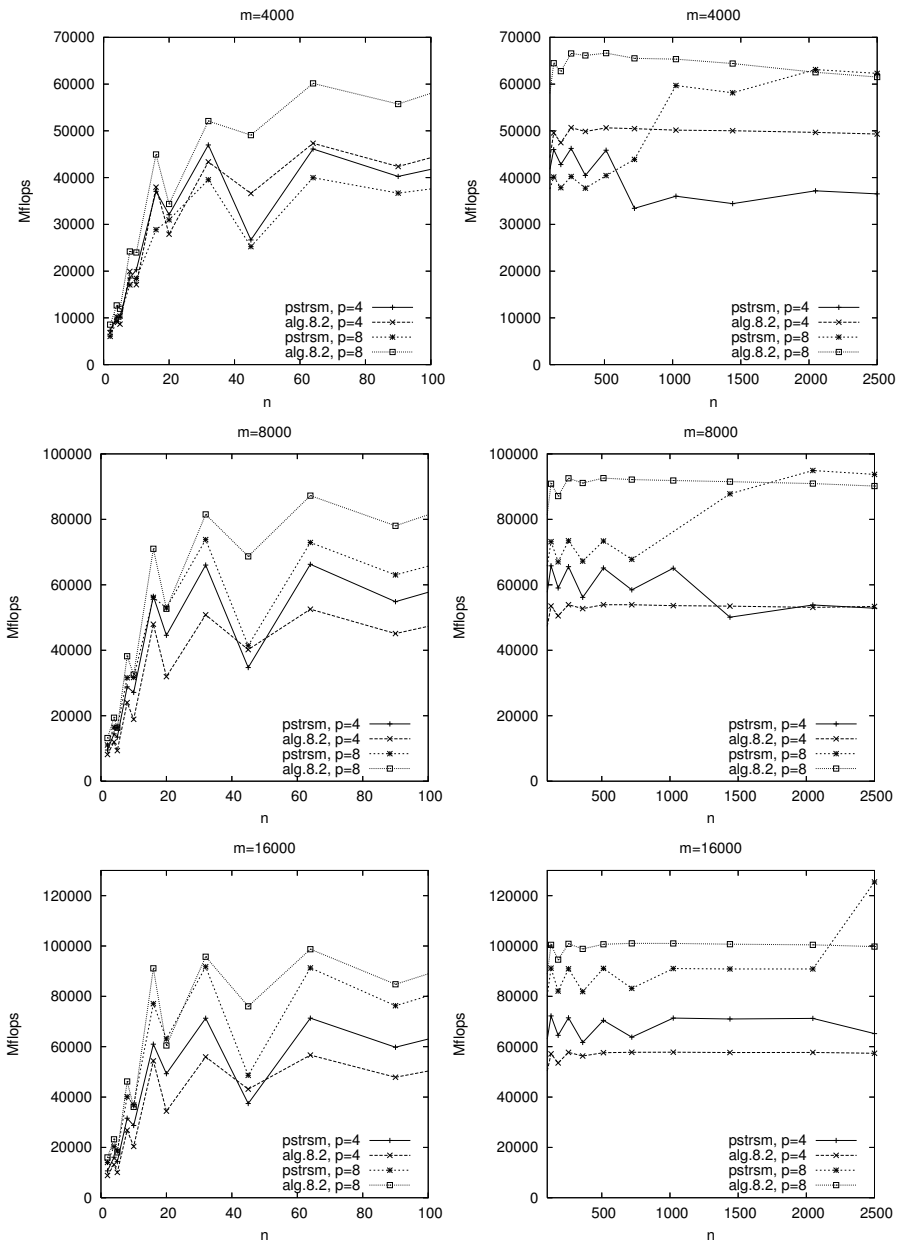
Fig. 8.6. Performance of the PBLAS routine `PSTRSV` and the new method on a cluster of Itanium 2 (left) and Cray X1 (right) for various matrix sizes (m) and $n = 1$

Podsumowując, nowa metoda alokacji danych pozwala na użycie mniejszej ilości pamięci, przy czym tylko niewielka jej część pozostaje niewykorzystana. Algorytmy wykorzystujące nową metodę mogą operować na większych blokach, co skutkuje ich szybszym działaniem w porównaniu do algorytmów wykorzystujących rozkład blokowo-cykliczny. Zauważmy na koniec, że nowa metoda alokacji danych może być zastosowana również dla macierzy symetrycznych.



Rys. 8.7. Wydajność klastra Itanium 2 dla podprogramu `PSTRSM` oraz nowego algorytmu 8.2 przy $n = 2, \ldots, 2500$ oraz $m = 4000,\ 8000,\ 16000$ przy liczbie procesorów $p = 4, 8, 16$

Fig. 8.7. Performance of the PBLAS routine `PSTRSM` and Algorithm 8.2 on a cluster of Itanium 2 for various matrix sizes $n = 2, \ldots, 2500$, $m = 4000, 8000, 16000$ and numbers of processors $p = 4, 8, 16$



Rys. 8.8. Wydajność podprogramu `PSTRSM` oraz nowego algorytmu 8.2 na komputerze Cray X1 dla $n = 2, \ldots, 2500$ oraz $m = 4000,\ 8000,\ 16000$ przy liczbie procesorów $p = 4, 8$

Fig. 8.8. Performance of the PBLAS routine `PSTRSM` and Algorithm 8.2 on a Cray X1 for various matrix sizes $n = 2, \ldots, 2500$, $m = 4000, 8000, 16000$ and numbers of processors $p = 4, 8$

9. PODSUMOWANIE I KIERUNKI DALSZYCH BADAŃ

W ostatnim rozdziale niniejszej pracy podsumujemy zaprezentowane w poprzednich rozdziałach wyniki oraz nakreślimy możliwości wykorzystania rezultatów badań w kierunkach stanowiących kontynuację wypracowanego podejścia do konstrukcji algorytmów obliczeń rekurencyjnych.

W rozdziale 1 sformułowaliśmy problem liniowych obliczeń rekurencyjnych o postaci (1.27) oraz (1.28) jako jeden z ważniejszych przypadków ograniczających wykorzystanie mocy obliczeniowych oferowanych przez współczesne komputery wektorowe i równoległe. Pokazaliśmy również, że najlepszą metodą konstrukcji efektywnych i przenośnych algorytmów, dobrze wykorzystujących możliwości obliczeniowe współczesnych procesorów, jest zastosowanie podprogramów z poziomów drugiego i trzeciego biblioteki BLAS, czyli sformułowanie algorytmów w terminach operacji macierzowych i wektorowych. Gwarantuje to możliwość użycia mechanizmów wektorowości oraz przede wszystkim daje szansę na realizację koncepcji lokalności danych, która ma na celu efektywne wykorzystanie pamięci podręcznej i przez to znaczne skrócenie czasu obliczeń. Jednocześnie trzeba zaznaczyć, że takie blokowe algorytmy mogą być w prosty sposób dostosowywane do wykorzystania nowych sposobów reprezentacji macierzy, a przez to jeszcze lepiej wykorzystać pamięć podręczną. Takie algorytmy można też łatwo i efektywnie zrównoleglić, co w obecnej chwili staje się szczególnie ważne z uwagi na upowszechnienie się procesorów wielordzeniowych.

Znane w literaturze praktyczne podejścia do realizacji liniowych obliczeń rekurencyjnych umożliwiają jedynie wektoryzację obliczeń, z możliwością zrównoleglenia dla równań rzędu pierwszego i drugiego bądź też ukierunkowane są na konkretne typy architektur komputerowych, oraz na ogół nie uwzględniają możliwości przyspieszenia obliczeń przez właściwe wykorzystanie pamięci podręcznej. Wadą tych algorytmów jest również brak skalowalności. Ich efektywność nie rośnie wraz ze wzrostem wartości m, w przypadku zaś obliczeń równoległych wymagają one większej liczby procesorów dla osiągnięcia zadowalającego przyspieszenia.

Podstawowym wynikiem uzyskanym w pierwszej części pracy jest pokazanie możliwości pełnej wektoryzacji obliczeń rekurencyjnych za pomocą operacji BLAS-u oraz sformułowanie

szybkich algorytmów blokowych realizujących obliczenia (1.28), przy wykorzystaniu podstawowych podprogramów algebry liniowej. Przerzucono w ten sposób optymalizację programów wykonujących obliczenia rekurencyjne na poziom zależnej od konkretnego sprzętu implementacji BLAS-u, co jednocześnie gwarantuje przenośność algorytmów. Osiągnięto ten cel przez potraktowanie wektora danych reprezentujących rozwiązywany problem jako tablicy dwuwymiarowej. W przypadku najważniejszego, prezentowanego w rozdziale 2, algorytmu 2.1, całość obliczeń została wyrażona w terminach operacji `AXPY`. Algorytm 3.1 wykorzystuje operacje BLAS-u poziomu drugiego: mnożenie macierzy pełnej przez wektor oraz mnożenie macierzy trójkątnej przez wektor. W przypadku konieczności wyznaczenia rozwiązania pełnego, blisko połowa obliczeń jest realizowana przez wywołanie jednej operacji mnożenia macierzy z BLAS-u poziomu trzeciego. Algorytm stanowi pierwszą próbę osiągnięcia tak dużej efektywności obliczeń rekurencyjnych dzięki zastosowaniu BLAS-u wyższych poziomów, a jak wykazaliśmy w rozdziale 1 konstrukcja algorytmów blokowych stanowi klucz do osiągnięcia dużej wydajności obliczeń.

Kolejnym ważnym wynikiem, jest analiza czasu wykonania algorytmu 2.1 na podstawie modelu Hockneya-Jesshope'a. Dzięki temu otrzymaliśmy teoretyczną predykcję optymalnych wartości parametrów metody. Na uwagę zasługuje udowodnienie faktu, że otrzymane wartości nie zależą od konkretnych, charakterystycznych dla użytego procesora, wartości r_∞ oraz $n_{1/2}$. Dzięki temu otrzymaliśmy kod źródłowy algorytmu, w którym wyznacza się otrzymane i zależne od rozmiaru zadania parametry, przy czym pozostają one słuszne dla dowolnych procesorów wektorowych. Sformułowany algorytm 2.2 stanowi modyfikację algorytmu 2.1, polegającą na wykorzystaniu operacji mnożenia macierzy przez wektor z BLAS-u poziomu drugiego, kosztem zastosowania dodatkowego miejsca w pamięci. Uzyskany w ten sposób algorytm działa szybciej dla $m > 2$. Dla większych wartości m należy stosować algorytm 3.1. Dodatkową zaletą podejścia blokowego jest możliwość łatwego i efektywnego zrównoleglenia algorytmu, dzięki czemu w prosty sposób otrzymaliśmy algorytmy 3.2 i 4.1.

Przeprowadzone eksperymenty obliczeniowe na różnych dostępnych architekturach komputerowych, włączając procesory wielordzeniowe, potwierdzają duże przyspieszenie sformułowanych algorytmów osiągane względem podstawowego algorytmu 1.5. Jednocześnie, w przypadku odpowiednio dużych rozmiarów problemu, pozwalają na wykorzystanie teoretycznej maksymalnej wydajności na poziomie 50%, co jest bardzo dobrym wynikiem dla komputerów z procesorami wektorowymi.

Na szczególne podkreślenie zasługują również wyniki, dotyczące analizy algorytmu 4.1, stanowiącego rozproszoną wersję algorytmu 3.1 w ujęciu SPMD. Analiza algorytmu oraz wy-

znaczenie optymalnych wartości parametrów zostało przeprowadzone na podstawie modelu BSP obliczeń rozproszonych. Uzyskano podobny jak w przypadku algorytmu 2.1 wynik, podający optymalne wartości parametrów metody niezależnie od charakterystyki użytego komputera. Wyniki eksperymentów przeprowadzonych na różnych rodzajach rozproszonych architektur wieloprocesorowych potwierdzają efektywność metody.

W dalszej części monografii zaprezentowane zostały zastosowania algorytmów przedstawionych we wcześniejszych rozdziałach dla rozwiązania wybranych problemów pojawiających się w różnych dziedzinach nauki i techniki, jednocześnie ilustrujące trafność zastosowanego podejścia. Pokazujemy, że algorytm 4.1 wraz z nowym, rozproszonym algorytmem mnożenia dolnotrójkątnych macierzy pasmowych przez wektor stanowi bardzo szybkie rozwiązanie problemu stosowania (obliczania) filtra rekurencyjnego dla zadanego ciągu sygnałów wejściowych, który jest ważnym zadaniem obliczeniowym w teorii sygnałów. Szczególnie ciekawym wynikiem jest dalsza optymalizacja obliczeń rekurencyjnych dzięki zastosowaniu nowych sposobów reprezentacji macierzy, które znacznie poprawiają wykorzystanie pamięci podręcznej.

Bardzo ważne wyniki aplikacyjne zawiera rozdział 6 poświęcony wyznaczaniu sum trygonometrycznych. Rozwiązanie problemu stanowi przykład zastosowania algorytmu opisanego w rozdziale 3 do pewnej ogólniejszej niż (1.28) postaci wzoru (1.27). Umożliwia to wykorzystanie otrzymanego efektywnego algorytmu do rozwiązania problemu numerycznego wyznaczania odwrotności transformanty Laplace'a. Pokazaliśmy, że liczba operacji używanych w algorytmie wyznaczania sum trygonometrycznych może być istotnie zmniejszona dzięki zastosowaniu pewnych wzorów analitycznych, a w przypadku numerycznego odwracania transformanty Laplace'a można przyspieszyć działanie algorytmu przez dalszą redukcję liczby odwołań do pamięci operacyjnej. Pokazaliśmy, że w pewnych przypadkach warto zastąpić operacje BLAS-u poziomu pierwszego dedykowanymi operacjami na wektorach, które dość dobrze poddają się optymalizacji kompilatorów. Rozdział zawiera testy numeryczne potwierdzające dużą efektywność metody oraz jej dobre własności numeryczne. Innym ciekawym wynikiem aplikacyjnym jest zastosowanie algorytmu 3.1 do problemu obliczania wartości wielomianu. Stanowi przykład algorytmu charakteryzującego się wymaganą większą liczbą operacji niż jego sekwencyjny odpowiednik, a mimo to posiadającym podobne własności numeryczne.

W rozdziale 8 podaliśmy nowy, oszczędny sposób rozmieszczenia danych stanowiących układ równań liniowych o macierzy trójkątnej i wielu prawych stronach. Podstawowy algorytm 8.1 wyznaczania rozwiązania takich układów stanowi uogólniony na bloki przypadek obliczeń o postaci (1.27) i znajduje się w centrum badań nad nowoczesnymi algorytmami równoległymi efektywnego rozwiązywania problemów z dziedziny algebry liniowej [26, 6, 50]. Zastosowa-

ne podejście umożliwia równomierne rozłożenie danych, redukcję niewykorzystanego miejsca w pamięci oraz prowadzi do konstrukcji na ogół szybszego, niż umieszczony w bibliotece ScaLAPACK [12], algorytmu 8.2 rozwiązywania układów równań liniowych o postaci (8.2).

Zwróćmy uwagę, że zaprezentowany w niniejszej monografii sposób podejścia do konstrukcji algorytmów, które dobrze wykorzystują własności współczesnych architektur komputerowych, może być zastosowany do rozwiązywania innych problemów. Po pierwsze, dla wielu pozornie różnych problemów obliczeniowych trzeba zidentyfikować ich wspólne cechy i sformułować ogólniejszy problem, który w szczególnych przypadkach sprowadza się do konkretnych zadań. W przypadku problemów przedstawionych w rozdziałach 5, 6, 7 takim ogólniejszym problemem są liniowe równania rekurencyjne o stałych współczynnikach (1.28) oraz pewne przypadki bardziej ogólnych równań o postaci (1.27). Następnym krokiem jest próba stworzenia algorytmu, który rozwiązywałby taki problem oraz w zadowalającym stopniu wykorzystywał możliwości oferowane przez współczesne komputery. Takim podejściem jest zdecydowanie wyrażenie algorytmu w terminach operacji macierzowych, co, jak wykazaliśmy, jest możliwe również dla problemów, które pozornie na to nie pozwalają. Przy implementacji takich algorytmów może być wykorzystana biblioteka BLAS oraz nowe sposoby reprezentacji macierzy, co gwarantuje dobre wykorzystanie mocy procesora oraz umożliwia łatwe zrównoleglenie algorytmu na dowolną architekturę wieloprocesorową, co jest szczególnie ważne z uwagi na upowszechnienie się procesorów wielordzeniowych. Dalsza optymalizacja może być przeprowadzona dla konkretnych zadań obliczeniowych i powinna dotyczyć zmniejszenia liczby odwołań do pamięci.

Autor niniejszej monografii rozpoczął już prace nad wykorzystaniem omówionego podejścia do rozwiązywania układów równań o macierzach pasmowych, które powstają przy dyskretyzacji równań różniczkowych zwyczajnych. Równie ważne, jak i możliwe do zrealizowania wydaje się zastosowanie omówionego podejścia do rozwiązywania równań różniczkowych cząstkowych przy wykorzystaniu nowych sposobów reprezentacji macierzy. Omówiona w rozdziale 8 nowa metoda rozmieszczania danych opisujących macierze trójkątne może być z powodzeniem zastosowana dla reprezentacji macierzy symetrycznych oraz implementacji algorytmów działających na takich macierzach, jak na przykład rozkład Choleskiego. Co więcej, nowa metoda umożliwia działanie na dużych blokach danych, co wydaje się być szczególnie przydatne w środowiskach gridowych. Dodatkowo, istnieje możliwość adaptacji takich algorytmów dla środowisk heterogenicznych, gdzie rozmiar bloku byłby zależny od szybkości działania danego węzła obliczeniowego.

BIBLIOGRAFIA

1. Abate J., Valko P.: Multi-precision Laplace transform inversion, Int. J. Numer. Mech. Eng., 60, s. 797÷993, 2004.

2. Aho A.V., Sethi R., Ullman J.D.: Kompilatory: reguły, metody i narzędzia, WNT, Warszawa 2002.

3. Allen R., Kennedy K.: Optimizing compilers for modern architectures: a dependence-based approach, Morgan Kaufmann, 2001.

4. Anderson E., Bai Z., Bischof C., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., Ostruchov S., Sorensen D.: LAPACK User's Guide, SIAM, Philadelphia 1992.

5. Axelsson O., Eijkhout V.: A note on the vectorization of scalar recursions., Parallel Comput., 3, s. 73÷83, 1986.

6. Baboulin M., Giraud L., Gratton S., Langou J.: A distributed packed storage for large dense parallel in-core calculations, Tech. Rep. TR/PA/05/30, CERFACS, 2005.

7. Baker A., Dennis J., Jessup E.R.: Toward memory-efficient linear solvers, Lecture Notes in Computer Science, 2565, s. 315÷238, 2003.

8. Bansal S.S., Vishal B., Gupta R.: Near optimal Cholesky factorization on orthogonal multiprocessors, Information Processing Letters, 84, s. 23÷30, 2002.

9. Bario R., Melendo B., Serrano S.: On the numerical evaluation of linear recurrences, J. Comput. Appl. Math., 150, s. 71÷86, 2003.

10. Bini D.A., Fiorentino G.: On the parallel evaluation of a sparse polynomial at a point., Numer. Algorithms, 20, s. 323÷329, 1999.

11. Bisseling R.H.: Parallel Scientific Computation. A Structured Approach Using BSP and MPI, Oxford University Press, 2004.

12. Blackford L., et al: ScaLAPACK user's guide, SIAM, Philadelphia 1997.

13. Blelloch G., Chatterjee S., Zagha M.: Solving linear lecurrences with loop raking, Journal of Parallel and Distributed Computing, 25, s. 91÷97, 1995.

14. Buttari A., Dongarra J., Kurzak J., Langou J., Luszczek P., Tomov S.: The impact of multicore on math software, Lecture Notes in Computer Science, 4699, s. 1÷10, 2007.

15. Carlson D.A.: Solving linear recurrence systems on mesh-connected computers with multiple global buses, Journal of Parallel and Distributed Computing, 8, s. 89÷95, 1990.

16. Carlson D.A., Sugla B.: Time and Processor Efficient Parallel Algorithms for a Recurrence Equations and Related Problems, Proceedings of the 1984 International Conference on Parallel Processing (ICPP'84), s. 310÷314, IEEE, Bellaire, Michigan 1984.

17. Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R.: Parallel Programming in OpenMP, Morgan Kaufmann Publishers, San Francisco 2001.

18. Chaudron M.R., van Duin A.C.: The formal derivation of parallel triangular system solvers using a coordination-based design method., Parallel Comput., 24, s. 1023÷1046, 1998.

19. Chen S.C., Kuck D.J.: Time and parallel processor bounds for linear recurrence systems., IEEE Trans. Comput., 24, s. 701÷717, 1975.

20. Chen S.C., Kuck D.J., Sameh A.H.: Practical parallel band triangular system solvers, ACM Trans. on Math. Soft., 4, s. 270÷277, 1978.

21. Choi J., Dongarra J., Ostrouchov S., Petitet A., Walker D., Whaley R.: LAPACK Working Note 100: a proposal for a set of parallel basic linear algebra subprograms, http://www.netlib.org/lapack/lawns, 1995.

22. Consortium UPC: UPC Language Specifications, 2005.

23. Cormen T., Leiserson C., Rivest R.: Introduction to Algorithms, MIT Press, 1994.

24. Daydé M.J., Duff I.S.: The RISC BLAS: a blocked implementation of level 3 BLAS for RISC processors, ACM Trans. Math. Soft., 25, s. 316÷340, 1999.

25. Daydé M.J., Duff I.S., Petitet A.: A parallel block implementation of Level-3 BLAS for MIMD vector processors, ACM Trans. Math. Soft., 20, s. 178÷193, 1994.

26. D'Azevedo E.F., Dongarra J.J.: LAPACK Working Notes 135: packed storage extension for ScaLAPACK, 1998.

27. de Rosa M.A., Giunta G., Rizzardi M.: Parallel Talbot's algorithm for distributed memory machines, Parallel Computing, 21, s. 783÷801, 1995.

28. Dongarra J.: Performance of Various Computer Using Standard Linear Algebra Software, http://www.netlib.org/benchmark/performance.ps.

29. Dongarra J., Bunsch J., Moler C., Stewart G.: LINPACK User's Guide, SIAM, Philadelphia 1979.

30. Dongarra J., DuCroz J., Duff I., Hammarling S.: A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Soft., 16, s. 1÷17, 1990.

31. Dongarra J., DuCroz J., Hammarling S., Hanson R.: An extended set of Fortran basic linear algebra subprograms, ACM Trans. Math. Soft., 14, s. 1÷17, 1988.

32. Dongarra J., Duff I., Sorensen D., Van der Vorst H.: Solving Linear Systems on Vector and Shared Memory Computers, SIAM, Philadelphia 1991.

33. Dongarra J., Duff I., Sorensen D., Van der Vorst H.: Numerical Linear Algebra for High Performance Computers, SIAM, Philadelphia 1998.

34. Dongarra J., Gustavson F., Karp A.: Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, SIAM Rev., 26, s. 91÷112, 1984.

35. Dongarra J., Hammarling S., Sorensen D.: Block reduction of matrices to condensed form for eigenvalue computations, J. Comp. Appl. Math, 27, s. 215÷227, 1989.

36. Dongarra J., Johnsson L.: Solving banded systems on parallel processor, Parallel Computing, 5, s. 219÷246, 1987.

37. Dongarra J., et al: PVM: A User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge 1994.

38. Dongarra J.J., Whaley R.C.: LAPACK Working Note 94: A User's Guide to the BLACS V1.1, http://www.netlib.org/blacs, 1997.

39. Elmroth E., Gustavson F., Jonsson I., Kågström B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software, SIAM Rev., 46, s. 3÷45, 2004.

40. et al. J.D., ed.: The Sourcebook of Parallel Computing, Morgan Kaufmann Publishers, 2003.

41. Flynn M.: Some computer organizations and their effectiveness, IEEE Trans. Comput., C–21, s. 94, 1972.

42. Gajski D.D.: Processor Arrays for Computing Linear Recurrence Systems, Proceedings of the 1978 International Conference on Parallel Processing (ICPP'78), s. 246÷256, IEEE, Bellaire, Michigan 1978.

43. Garbow B., Boyle J., Dongarra J., Moler C.: Matrix eigensystems routines – EISPACK guide extension, Lecture Notes in Computer Science, Springer-Verlag, New York 1977.

44. Grama A., Gupta A., Karypis G., Kumar V.: An Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, 2003.

45. Greenberg A., R.E.Lander, Paterson M., Galil Z.: Efficient parallel algorithms for linear recurrence computation, Inf. Proc. Letters, 15, s. 31÷35, 1982.

46. Gropp W., Lusk E., Skjellum A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, http://www-unix.mcs.anl.gov/mpi/mpich1/papers/mpicharticle/paper.html.

47. Gustavson F.G.: New generalized data structures for matrices lead to a variety of high performance algorithms, Lect. Notes Comput. Sci., 2328, s. 418÷436, 2002.

48. Gustavson F.G.: High-performance linear algebra algorithms using new generalized data structures for matrices, IBM J. Res. Dev., 47, s. 31÷56, 2003.

49. Gustavson F.G.: 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, s. 618÷621, 2008.

50. Gustavson F.G., Karlsson L., Kagstrom B.: Three algorithms for Cholesky factorization on distributed memory using packed storage, Lecture Notes in Computer Science, 4699, s. 550÷559, 2007.

51. Heath M., Romine C.: Parallel solution of triangular systems on distributed memory multiprocessors, SIAM J. Sci. Statist. Comput., 9, s. 558÷588, 1988.

52. Higham N.J.: Accuracy and Stability of Numerical Algorithms, SIAM, Philadelphia 1996.

53. Hill J., Donaldson S., Skillicorn D.: Portability of Performance with the *BSPlib* Communications Library, Programming Models for Massively Parallel Computers, (MPPM'97), IEEE Computer Society Press, London 1997.

54. Hockney R., Jesshope C.: Parallel Computers: Architecture, Programming and Algorithms, Adam Hilger Ltd., Bristol 1981.

55. Horvitz G., Bisseling R.H.: Designing a BSP Version of ScaLAPACK, B. Hendrickson, et al, eds., Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia 1999.

56. Hyafil L., Kung H.: The complexity of parallel evaluation of linear recurrences., J. Assoc. Comput. Mach., 24, s. 513÷521, 1977.

57. Intel Corporation: Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, 2002.

58. Intel Corporation: IA-32 Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture, 2003.

59. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual, 2007.

60. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture, 2008.

61. Jamieson L.H., Gannon D.B., Douglass R.J.: The Characteristics of Parallel Algorithms, MIT Press, 1987.

62. Johnsson S.: Solving narrow banded systems on ensemble architectures., ACM Trans. Math. Softw., 11, s. 271÷288, 1985.

63. Kågström B., Ling P., Loan C.V.: GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark, ACM Trans. Math. Soft., 24, s. 268÷302, 1998.

64. Kim H.S., Yoon Y.H., Han D.S.: Parallel processing of first order linear recurrence on SMP machines, The Journal of Supercomputing, 27, s. 295÷310, 2004.

65. Kincaid D., Chenay W.: Analiza numeryczna, WNT, Warszawa 2006.

66. Kitowski J.: Współczesne systemy komputerowe, CCNS, Kraków, 2000.

67. Kowarschik M., Weiss C.: An overview of cache optimization techniques and cache-aware numerical algorithms, Lecture Notes in Computer Science, 2625, s. 213÷232, 2003.

68. Kozielski S., Szczerbiński Z.: Komputery równoległe: architektura, elementy programowania, WNT, Warszawa 1994.

69. Kozniewska I.: Rownania rekurencyjne, PWN, Warszawa 1972.

70. Kuszmaul B.C., Henry D.S., Loh G.H.: A comparison of asymptotically scalable superscalar processors, Theory of Computing Systems, 35, s. 129÷150, 2002.

71. Lacoursière C.: A Parallel block iterative method for interactive contacting rigid multibody simulations on multicore PCs, Lecture Notes in Computer Science, 4699, s. 956÷965, 2007.

72. Larid M.: A Comparison of Three Current Superscalar Designs, Computer Architecture News (CAN), 20, s. 14÷21, 1992.

73. Larriba-Pey J.L., Navarro J.J., Jorba A., Roig O.: Review of general and Toeplitz vector bidiagonal solvers., Parallel Comput., 22, s. 1091÷1125, 1996.

74. Lawson C., Hanson R., Kincaid D., Krogh F.: Basic linear algebra subprograms for Fortran usage, ACM Trans. Math. Soft., 5, s. 308÷329, 1979.

75. Lee J.b., Sung W., Moon S.M.: 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.

76. Li G., Coleman T.F.: A new method for solving triangular systems on distributed-memory message-passing multiprocessors., SIAM J. Sci. Stat. Comput., 10, s. 382÷396, 1989.

77. Li L., Hu J., Nakamura T.: A simple parallel algorithm for polynomial evaluation., SIAM J. Sci. Comput., 17, s. 260÷262, 1996.

78. Lu M., Qiao X., Chen G.: A parallel algorithm for evaluating general linear recurrence equations., Circuits Syst. Signal Process., 15, s. 481÷504, 1996.

79. McCurdy C.W., Stevens R., Simon H., Kramer W., Bailey D., Johnston W., Catlett C., Lusk R., Morgan T., Meza J., Banda M., Leighton J., Hules J.: Creating Science-driven Computer Architecture: A New Path to Scientific Leadership, 2007, URL `http://www.osti.gov/servlets/purl/806195-zzoRAy/native/`.

80. Modi J.: Parallel Algorithms and Matrix Computation, Oxford University Press, Oxford, 1988.

81. Munro I., Paterson M.: Optimal algorithms for parallel polynomial evaluation, J. Comput. System Sci., 7, s. 189÷198, 1973.

82. Murli A., Rizzardi M.: Algorithm 682: Talbot's method for the Laplace inversion problem, ACM Trans. Math. Soft., 16, s. 158÷168, 1990.

83. Netwok Computer Services Inc.: The AHPCRC Cray X1 Primer, http://www.ahpcrc.org/publications/Primer.pdf.

84. Ortega J.M.: Introduction to Parallel and Vector Solution of Linear Systems, Springer, 1988.

85. Pacheco P.: Parallel Programming with MPI, Morgan Kaufmann, San Frncisco, 1996.

86. Pan X.: An improved recursive doubling algorithm for the parallel solution of linear recurrence $R\langle n, 1\rangle$., Trans. Nanjing Univ. Aeronaut. Astronaut., 12, s. 218÷220, 1995.

87. Paprzycki M., Cyphers C.: Gaussian elimination on Cray Y-MP, CHPC Newsletter, 6, s. 77÷82, 1991.

88. Paprzycki M., Cyphers C.: Multiplying matrices on the cray - practical considerations, CHPC Newsletter, 6, s. 43÷47, 1991.

89. Paprzycki M., Stpiczyński P.: Solving linear recurrence systems on the Cray Y-MP, Lecture Notes in Computer Science, 879, s. 416÷424, 1994.

90. Paprzycki M., Stpiczyński P.: Solving Linear Recurrence Systems on Parallel Computers, R. Kalia, P. Vashishta, eds., Toward Teraflop Computing and New Grand Challenge Applications – Proceedings of the Conference Mardi Gras'94, Baton Rouge, February 1994, s. 379÷384, Nova Science Publishers, New York 1995.

91. Paprzycki M., Stpiczyński P.: Parallel solution of linear recurrence systems, Z. Angew. Math. Mech., 76, s. 5÷8, 1996.

92. Paprzycki M., Stpiczyński P.: A Brief Introduction to Parallel Computing, E. Kontoghiorghes, ed., Parallel Computing and Statistics, s. 3÷42, Taylor & Francis, 2006.

93. Parhami B.: Introduction to Parallel Processing: Algorithms and Architectures, Plenum Press, 1999.

94. Parhi K.K., Messerschmitt D.: Pipeline interleaving and parallelism in recursive digital filters, part ii: Pipelined incremental block filtering, IEEE Trans. Acoust.,Speech Signal Processing, ASSP-37, s. 1118÷1135, 1989.

95. Quinn M.J.: Parallel Programming in C with MPI and OpenMP, McGraw-Hill Education, 2003.

96. Rahman N.: Algorithms for hardware caches and TLB, Lecture Notes in Computer Science, 2625, s. 171÷192, 2003.

97. Robelly J.P., G.Cichon, Seidel H., Fettweis G.: Implementation of Recursive Digital Filters into Vector SIMD DSP Architectures, Proc. International Conference on Acoustics, Speech and Signal Processing. ICASSP 2004. Montreal, Canada, 17.-21. May 2004.

98. Romine C., Ortega J.: Parallel solutions of triangular systems of equations, Parallel Comput., 6, s. 109÷114, 1988.

99. Sameh A.H., Brent R.P.: Solving triangular systems on a parallel computer., SIAM J. Numer. Anal., 14, s. 1101÷1113, 1977.

100. Schendel U.: Introduction to Numerical Methods for Parallel Computers, Ellis Horwood Limited, New York 1984.

101. Smith S.W.: The Scientist and Engineer's Guide to Digital Signal Processing, California Technical Publishing, San Diego 1997.

102. Stewart G.: Introduction to Matrix Computations, Academic Press, New York 1973.

103. Stoer J., Bulirsh R.: Introduction to Numerical Analysis, Springer, New York 1993.

104. Stpiczyński P.: Parallel algorithms for solving linear recurrence systems, Lecture Notes in Computer Science, 634, s. 343÷348, 1992.

105. Stpiczyński P.: Parallel Cholesky factorization on orthogonal multiprocessors, Parallel Computing, 18, s. 213÷219, 1992.

106. Stpiczyński P.: Efficient data-parallel algorithms for computing trigonometric sums, Ann. Univ. Mariae Curie-Sklodowska Sect. A, 56, s. 85÷96, 2002.

107. Stpiczyński P.: Fast Parallel Algorithms for Computing Trigonometric Sums, M. Tudruj, A. Jordan, eds., Proceedings of PARELEC 2002, International Conference on Parallel Computing in Electrical Engineering, s. 299÷304, IEEE Computer Society Press, 2002.

108. Stpiczyński P.: A new message passing algorithm for solving linear recurrence systems, Lecture Notes in Computer Science, 2328, s. 466÷473, 2002.

109. Stpiczyński P.: Fast parallel algorithm for polynomial evaluation, Parallel Algorithms and Applications, 18, s. 209÷216, 2003.

110. Stpiczyński P.: Fast solver for Toeplitz bidiagonal systems of linear equations, Ann. Univ. Mariae Curie-Sklodowska Informatica, 1, s. 81÷87, 2003.

111. Stpiczyński P.: Numerical Evaluation of Linear Recurrences on High Performance Computers and Clusters of Workstations, Proceedings of PARELEC 2004, International Conference on Parallel Computing in Electrical Engineering, s. 200÷205, IEEE Computer Society Press, 2004.

112. Stpiczyński P.: Numerical Evaluation of Linear Recurrences on Various Parallel Computers, M. Kovacova, ed., Proceedings of Aplimat 2004, 3rd International Conference, Bratislava, Slovakia, February 4–6, 2004, s. 889÷894, Technical Univerity of Bratislava, Bratysława 2004.

113. Stpiczyński P.: Solving linear recurrence rystems using level 2 and 3 BLAS routines, Lecture Notes in Computer Science, 3019, s. 1059÷1066, 2004.

114. Stpiczyński P.: Evaluating recursive filters on distributed memory parallel computers, Comm. Numer. Meth. Engng., 22, s. 1087÷1095, 2006.

115. Stpiczyński P.: A Note on the numerical inversion of the Laplace transform, Lecture Notes in Computer Science, 3911, s. 551÷558, 2006.

116. Stpiczyński P.: New data distribution for solving triangular systems on distributed memory machines, Lecture Notes in Computer Science, 4699, s. 589÷597, 2007.

117. Stpiczyński P.: Evaluating linear recursive filters using novel data formats for dense matrices, Lecture Notes in Computer Science, 4967, s. 688÷697, 2008.

118. Stpiczyński P., Paprzycki M.: Parallel Algorithms for Finding Trigonometric Sums, D. Bailey, et al, eds., Parallel Processing for Scientific Computing – Proceedings of the Sixth SIAM Conference on Parallel Computing, San Francisco, February 1995, s. 291÷292, SIAM, Philadelphia 1995.

119. Stpiczyński P., Paprzycki M.: 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, s. 541÷551, Facultade de Engerharia do Universidade do Porto, 2000.

120. Stpiczyński P., Paprzycki M.: Numerical Software for Solving Dense Linear Algebra Problems on High Performance Computers, M. Kovacova, ed., Proceedings of Aplimat 2005, 4th International Conference, Bratislava, Slovakia, February 2005, s. 207÷218, Technical Univerity of Bratislava, Bratysława 2005.

121. Talbot A.: The accurate numerical inversion of Laplace transforms, J. Inst. Maths. Applics., 23, s. 97÷120, 1979.

122. van de Geijn R.A., Overfelt J.: Advanced Linear Algebra Object Manipulation, R.A. van de Geijn, ed., Using PLAPACK: Parallel Linear Algebra Package, Scientific and Engineering Computing, s. 42÷57, MIT Press, Cambridge 1997.

123. van der Vorst H., Dekker K.: Vectorization of linear recurrence relations., SIAM J. Sci. Stat. Comput., 10, s. 27÷35, 1989.

124. Wang H.: A parallel method for tridiagonal equations., ACM Trans. Math. Softw., 7, s. 170÷183, 1981.

125. Wang H., Nicolau A.: Speedup of Banded Linear Recurrences in the Presence of Resource Constraints, 6th ACM International Conference on Supercomputing (6th ICS'92), s. 466÷477, Washington 1992.

126. Wang H., Nicolau A., Keung S., Siu K.Y.: Computing programs containing band linear recurrences on vector supercomputers, IEEE Trans. Parallel Distrib. Systems, 7, s. 769÷782, 1996.

127. Whaley R.C., Petitet A., Dongarra J.J.: Automated empirical optimizations of software and the ATLAS project, Parallel Computing, 27, s. 3÷35, 2001.

128. Wolfe M.: High Performance Compilers for Parallel Computing, Addison–Wesley, 1996.

129. Yoon Y., Kim H.S., Han D.S., Youn H.Y.: Parallel Processing of Recurrence Operations in SMP Machines, H.R. Arabnia, ed., Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, CSREA Press, 2000.

130. Zima H.: Supercompilers for Parallel and Vector Computers, ACM Press, 1990.

OPTYMALIZACJA OBLICZEŃ REKURENCYJNYCH NA KOMPUTERACH WEKTOROWYCH I RÓWNOLEGŁYCH

Streszczenie

W monografii zaprezentowano nowe, efektywne metody realizacji liniowych obliczeń rekurencyjnych na współczesnych komputerach wektorowych i równoległych. Na początku sformułowano nowe blokowe algorytmy, opierając się na metodach algebry liniowej. Dzieki temu możliwa stała się implementacja algorytmów, dobrze wykorzystująca możliwości oferowane przez współczesne procesory, poprzez użycie operacji zdefiniowanych w ramach biblioteki BLAS, zawierającej podstawowe podprogramy algebry liniowej. W dalszym ciągu pokazano, jak zrównoleglić wprowadzone algorytmy na komputery równoległe z pamięcią wspólną przy użyciu standardu OpenMP oraz komputery z pamięcią rozproszoną i klastry przy użyciu standardu MPI. Druga część monografii pokazuje przykładowe zastosowania wprowadzonych algorytmów blokowych i równoległych do rozwiązywania ważnych problemów obliczeniowych, takich jak wyznaczanie wartości liniowych filtrów rekurencyjnych, interpolacja trygonometryczna, numeryczne odwracanie transformanty Laplace'a oraz obliczanie wartości wielomianów. Podano również nowy oszczędny sposób reprezentacji macierzy trójkątnych i symetrycznych w obliczeniach rozproszonych. Wszystkie zaprezentowane w monografii algorytmy zostały zaimplementowane i przetestowane na różnych systemach komputerowych. Podano wyniki eksperymentów i wnioski potwierdzające wysoką efektywność algorytmów.

W rozdziale 1 omawione są podstawowe zagadnienia związane ze współczesnymi architekturami komputerów dużej mocy obliczeniowej. Przedstawiono wykorzystywane w pracy metody analizy algorytmów, formalnie definiowano problem liniowych obliczeń rekurencyjnych oraz zaprezentowano w skrócie najważniejsze znane metody jego rozwiązania. Rozdział 2 opisuje efektywną metodę wektoryzacji obliczeń rekurencyjnych. Rozdziały 3 i 4 pokazują metodę, pozwalającą na wykorzystanie operacji macierzowych oraz jej zrównoleglenie na komputery z pamięcią wspólną i rozproszoną. Rozdział 5 omawia zastosowanie wprowadzonych we wcześniejszych rozdziałach metod dla szybkiego wyznaczania liniowych filtrów rekurencyjnych. W rozdziale 6 omawiane jest zastosowanie algorytmów z poprzednich rozdziałów do

rozwiązania problemu wyznaczania sum trygonometrycznych oraz optymalizacji numerycznego wyznaczania odwrotności transformanty Laplace'a. Rozdział 7 omawia algorytm szybkiego obliczania wartości wielomianów, bazujący na metodzie przedstawionej w rozdziale 2, wraz z jego analizą numeryczną. Na koniec w rozdziale 8 podajemy nową metodę rozmieszczenia danych dla problemu rozwiązywania układów równań liniowych o macierzach trójkątnych, która pozwala na oszczędniejsze wykorzystanie pamięci oraz konstrukcję szybszych algorytmów.

OPTIMIZATION OF RECURRENCE COMPUTATIONS ON VECTOR AND PARALLEL COMPUTERS

Abstract

The aim of this monograph is to present the author's contribution to the fields of designing vector and parallel algorithms for solving problems based on linear recurrence computations and optimizing such software for modern vector and parallel computer architectures.

In the first chapter, we give a concise overview of the fundamentals of of moder computer architectures, performance analysis and methods for vector parallel programming. We also define the problem of solving linear recurrence systems and present some basic results which can be found in the literature.

Chapter 2 describes the use of the Level 1 BLAS operation `AXPY` as a key to efficient vectorization of m-th order linear recurrence systems with constant coefficients. Applying the Hockney-Jesshope model of vector computations, we present the performance analysis of the algorithm. We also consider the influence of memory bank conflicts. The theoretical analysis is supported by the experimental results collected on two vector supercomputers manufactured by Cray.

The aim of Chapter 3 is to present a new efficient BLAS-based algorithm for solving linear recurrence systems with constant coefficients, which can be easily and efficiently implemented on shared memory machines. The algorithm is based on Level 3 and Level 2 BLAS routines `GEMM`, `GEMV` and `TRMV`, which are crucial for its efficiency. The results of experiments performed on a various shared memory computers are also presented and discussed. The can be efficiently implemented on high performance message passing parallel and vector computers and clusters of workstations. In Chapter 4 we analyze the performance of the algorithm using two well known models: BSP and Hockney-Jesshope. Finally, we present the results of experiments performed of a Cray X1 and two different clusters running under Linux.

The aim of the chapters 5, 6 and 7 is to show that the introduced algorithms for solving linear recurrence systems can be applied for solving a number of problems which arise in scientific computing. In Chapter 5 we introduce a new BLAS-based algorithm for narrow-banded

triangular Toeplitz matrix-vector multiplication and show how to evaluate linear recursive filters efficiently on distributed memory parallel computers. We apply the BSP model of parallel computing to predict the behavior of the algorithm and to find the optimal values of the method's parameters. The results of experiments performed on a cluster of twelve dual-processor Itanium 2 computers and Cray X1 are also presented and discussed. The algorithm allows to utilize up to 30% of the peak performance of 24 Itanium processors, while a simple scalar algorithm can only utilize about 4% of the peak performance of a single processor. Next, we show that the performance of the algorithm for evaluating linear recursive filters can be increased by using new generalized data structures for dense matrices introduced by F. G. Gustavson. The results of experiments performed on Intel Itanium 2, Cray X1 and dual-processor Quad-Core Xeon are also presented and discussed. In Chapter 6 we introduce a new high performance *divide and conquer* algorithm for finding trigonometric sums which can be applied to improve the performance of the Talbot's method for the numerical inversion of the Laplace Transform on modern computer architectures including shared memory parallel computers. We also show how to vectorize the first stage of the Talbot's method, namely computing all coefficients of the trigonometric sums used by the method. Numerical tests show that the improved method gives the same accuracy as the standard algorithm and it allows to utilize parallel processors. In Chapter 7 we show hot to apply our algorithms for polynomial evaluation.

Finally, Chapter 8 presents the new data distribution of triangular matrices that provides steady distribution of blocks among processes and reduces memory wasting in comparison with the standard block-cyclic data layout that is used in the ScaLAPACK library for dense matrix computations. The new algorithm for solving triangular systems of linear equations is also introduced. The results of experiments performed on a cluster of Itanium 2 processors and Cray X1 show that in some cases, the new method is faster than corresponding PBLAS routines `PSTRSV` and `PSTRSM`.

SPIS RYSUNKÓW

SPIS TABEL

INFORMATION FOR AUTHORS

The journal *STUDIA INFORMATICA* publishes both fundamental and applied Memoirs and Notes in the field of informatics. The Editors' aim is to provide an active forum for disseminating the original results of theoretical research and applications practice of informatics understood as a discipline focused on the investigations of laws that rule processes of coding, storing, processing, and transferring of information or data.

Papers are welcome from fields of informatics inclusive of, but not restricted to *Computer Science*, *Engineering*, and *Life and Physical Sciences*.

All manuscripts submitted for publication will be subject to critical review. Acceptability will be judged according to the paper's contribution to the art and science of informatics.

In the first instance, all text should be submitted as hardcopy, conventionally mailed, and for accepted paper accompanying with the electronically readable manuscript to:

Dr. Marcin SKOWRONEK
Institute of Informatics
Silesian University of Technology
ul. Akademicka 16
44-100 Gliwice, Poland
Tel.: +48 32 237-12-15
Fax: +48 32 237-27-33
e-mail: marcins@polsl.pl

MANUSCRIPT REQUIREMENTS

All manuscripts should be written in Polish or in English. Manuscript should be typed on one side paper only, and submitted in duplicate. The name and affiliation of each author should be followed by the title of the paper (as brief as possible). An abstract of not more than 50 words is required. The text should be logically divided under numbered headings and subheadings (up to four levels). Each table must have a title and should be cited in the text. Each figure should have a caption and have to be cited in the text. References should be cited with a number in square brackets that corresponds to a proper number in the reference list. The accuracy of the references is the author's responsibility. Abbreviations should be used sparingly and given in full at first mention (e.g. "Central Processing Unit (CPU)"). In case when the manuscript is provided in Polish (English) language, the summary and additional abstract (up to 300 words with reference to the equations, tables and figures) in English (Polish) should be added.

After the paper has been reviewed and accepted for publication, the author has to submit to the Editor a hardcopy and electronic version of the manuscript.

It is strongly recommended to submit the manuscript in a form downloadable from web site http://zti.iinf.polsl.gliwice.pl/makiety/.

To subscribe: *STUDIA INFORMATICA* (PL ISSN 0208-7286) is published by Silesian University of Technology Press (Wydawnictwo Politechniki Śląskiej) ul. Akademicka 5, 44-100 Gliwice, Poland, Tel./Fax +48 32 237-13-81. 2008 annual subscription rate: US$40. Single number price approx. US$8-15 according to the issue volume.

INSTYTUT INFORMATYKI prowadzi:

- Studia Stacjonarne
  - na poziomie inżynierskim (7-semestralne)
  - na poziomie magisterskim (10-semestralne)
- Studia Niestacjonarne
  - na poziomie inżynierskim (8-semestralne)
  - na poziomie magisterskim (8- i 4-semestralne)
- Niestacjonarne Studia Magisterskie drugiego stopnia (4-semestralne)
- Studia Podyplomowe (2-semestralne) z zakresu:

  SIECI KOMPUTEROWE,
  SYSTEMY MIKROKOMPUTEROWE i BAZY DANYCH

Informacje:

POLITECHNIKA ŚLĄSKA
Instytut Informatyki

44-100 Gliwice, ul. Akademicka 16
tel. (032) 237 24 05; 237 21 51;
fax (032) 237 27 33 (czynny całą dobę)

e-mail: rau2@polsl.pl

http://www.inf.polsl.pl/ (dydaktyka)