Instytut Podstawowych Problemów Techniki
Polskiej Akademii Nauk

Praca doktorska

Zastosowanie algorytmów wielowątkowych i rozproszonych do zwiększenia efektywności Metody Elementów Skończonych

Paweł Jarzębski

Promotor: prof. dr hab. inż. Krzysztof Wiśniewski

Warszawa 2017

Rodzicom.

Podziękowania

Chciałbym serdecznie podziękować promotorowi niniejszej pracy profesorowi Krzysztofowi Wiśniewskiemu za okazaną przychylność, cierpliwość i pomoc w trakcie powstawania tej pracy.

Dziękuję również wszystkim współpracownikom i kolegom z IPPT PAN za stworzenie miłej atmosfery, w której wszystkie pomysły mogły powstać.

Pracę dedykuję rodzicom za wychowanie i pokierowanie w pierwszym etapie mojego życia w taki sposób, że teraz dotarłem do tego miejsca.

Cichym bohaterem tej pracy jest moja żona Ewelina, która cały czas mnie wspierała i pchała do zakończenia, dziękuję!

Autor

Streszczenie

Rozprawa dotyczy efektywnego wykorzystania procesorów wielordzeniowych i klastra komputerów w obliczeniach Metodą Elementów Skończonych (MES) problemów brzegowych z mechaniki ciał trójwymiarowych i konstrukcji powłokowych.

Celem rozprawy jest stworzenie efektywnych i skalowalnych algorytmów wielowątkowych (z użyciem OpenMP [83]) i rozproszonych (z użyciem MPI [80]). Przeanalizowano dotychczasowy stan wiedzy w dziedzinie zrównoleglenia i zaproponowano ulepszenia w dwóch obszarach kodu MES - pętli po elementach i rozwiązywania układu równań liniowych.

Opracowany algorytm zrównoleglenia pętli po elementach i redukcji macierzy elementowych do macierzy globalnej z wykorzystaniem OpenMP osiąga efektywność ok. 95%. Algorytm został zaimplementowany w złożonym pod względem programistycznym kodzie FEAP [25], który jest wykorzystywany w wielu ośrodkach akademickich na świecie.

W ramach rozwiązywania układu równań na pojedynczym węźle obliczeniowym, najpierw porównano zestaw wiodących solwerów do rozwiązywania rzadkich układów równań liniowych. Dla każdego z nich przedstawiono czas, skalowalność i zapotrzebowanie na pamięć. Opisano również analizę solwerów wg modelu „Roofline”. Przeanalizowano kod źródłowy solwera HSL MA86 [43] i zaproponowano kilka ulepszeń, dzięki którym przyspieszono obliczenia o ok. 11.5% i poprawiono skalowalność o ok. 6.5%. Opracowano wersję tego solwera wykorzystującą mieszaną precyzją, z faktoryzacją w pojedynczej precyzji i iteracyjnym poprawianiem w podwójnej precyzji, co przyspieszyło obliczenia 2.17 razy i poprawiło skalowalność do 10.28 na 12 rdzeniach.

W celu dalszego zwiększenia efektywności rozwiązywania układu równań opracowano solwer na klaster, bazujący na dekompozycji obszaru i obliczaniu uzupełnienia Schura za pomocą częściowej faktoryzacji (ang. *partial factorization*), którą zaimplementowano w solwerze HSL MA86. Dzięki temu, obliczenia dla podobszarów są bardzo efektywne ale faktoryzacja macierzy interfejsowej pozostaje barierą dla skalowalnych obliczeń, nawet dla HSL MA64, który jest najszybszy z przetestowanych solwerów dla macierzy gęstych. Aby przyśpieszyć faktoryzację zaproponowano wykorzystanie dwóch innych technik: hierarchicznej faktoryzacji oraz mieszanej precyzji i iteracyjnego poprawiania. Wykorzystując OpenMP i MPI przyśpieszono obliczenia ok. 36 razy na 8 węzłach z 12 rdzeniami każdy, co jest wydajnością lepszą niż profesjonalnego solwera WSMP [37]. Dodatkowo, do celów porównawczych, opracowano solwer FETI-DP wykorzystujący metody bezpośrednie na węzłach i iteracyjny solwer PCG do rozwiązania układu równań dla interfejsów.

Opracowane algorytmy zostały przetestowane na modelach MES dla jednorodnego materiałowo sześcianu i dwóch materiałów o złożonej mikrostrukturze: piance korundowej i metalowo-ceramicznym kompozycie. Największy obliczony model zawierał ponad 30 mln niewiadomych. Testy potwierdziły bardzo dobrą jakość opracowanych algorytmów.

Abstract

The thesis concerns the effective use of multicore processors and a cluster of computers in Finite Element (FE) analyses of boundary value problems in mechanics of 3D bodies and shell-like structures.

The aim is to develop effective and scalable multithreaded (using OpenMP [83]) and distributed (using MPI [80]) algorithms. The current state of knowledge about parallelization is analyzed and improvements are proposed in two areas of the FE code - the loop over elements and the solvers of a linear system of equations.

The proposed algorithm of parallelization of the loop over elements and the reduction of elemental matrices to the global matrix using OpenMP has effectiveness around 95%. It was implemented in a complex, in terms of programming, code FEAP [25], used in many academic centers in the world.

Concerning the solvers for one computational node, first, several well known sparse parallel solvers are compared. For each of them time, scalability, and the memory usage are presented. Additionally, the “Roofline Model” for these solvers is provided. Next, the source code of the solver HSL MA86 [43] is analyzed and several improvements, which speed up computation by about 11.5% and scalability by about 6.5%, are proposed. Next, the mixed precision version of this solver is developed, with factorization in single precision and iterative refinement in double precision, speeding up computations 2.17 times and providing the scalability 10.28 on 12 cores.

To further improve the effectiveness of solving a system of linear equations, a solver for a computer cluster is developed. It is based on the domain decomposition and the computation of the Schur complement by partial factorization, which is implemented in the solver HSL MA86. This makes the domain computations very efficient although factorization of the interface matrix is a bottleneck even for HSL MA64, which is the fastest of several tested solvers for dense matrices. To speed up the factorization, the use of two other techniques is proposed: the hierarchical factorization and the mixed precision with iterative refinement. The total speedup achieved using OpenMP and MPI, on 8 nodes 12 cores each, is about 36 times, and is better than of the professional solver WSMP [37]. Additionally, for comparison, the FETI-DP method, using direct methods on nodes and the iterative PCG solver for the interface equations, is implemented.

The developed algorithms are tested on FE models of a homogeneous cube and models of two materials with a complicated microstructure: a corundum foam and a metal-ceramic composite. The biggest model computed has over 30 mln unknowns. Tests prove that the implemented algorithms are of a very good quality.

Spis treści

Lista pojęć i skrótów 9

1 Wstęp 11
1.1 Wprowadzenie . . . 11
1.2 Cel rozprawy . . . 12
1.3 Zawartość rozprawy . . . 13

2 Obliczenia równoległe 15
2.1 Architektura komputerów . . . 15
2.2 Miary zrównoleglenia obliczeń . . . 16
2.2.1 Prawo Amdahla i prawo Gustafsona-Barsisa . . . 17
2.2.2 Skalowalność . . . 21
2.3 Ograniczenia sprzętowe . . . 23
2.3.1 Wprowadzenie . . . 23
2.3.2 Natężenie operacji . . . 23
2.3.3 Model „Roofline” . . . 24
2.3.4 Wyznaczenie modelu „Roofline” . . . 24
2.4 Programowanie równoległe na maszyny z pamięcią wspólną . . . 26
2.4.1 Wprowadzenie do OpenMP . . . 26
2.4.2 Dyrektywa „PARALLEL DO” . . . 27
2.4.3 Zasięg zmiennych . . . 29
2.4.4 Synchronizacja . . . 30
2.5 Programowanie równoległe na maszyny z pamięcią rozproszoną . . . 33
2.5.1 Wprowadzenie . . . 33
2.5.2 Programowanie równoległe z MPI . . . 34
2.6 Podstawowe problemy programowania równoległego . . . 37
2.6.1 Zakleszczenie . . . 37
2.6.2 Fałszywe współdzielenie . . . 37
2.6.3 Jednoczesna wielowątkowość . . . 39
2.6.4 Koligacja procesu/wątku . . . 39

3 Zrównoleglenie pętli po elementach w FEAPie 40
3.1 Opis problemu . . . 40
3.2 Porównanie sekwencyjnego kodu FEAP z równoległym kodem Warp3D . . . 42
3.2.1 Test kostki . . . 43
3.2.2 Metoda sparalelizowania pętli w Warp3D . . . 46
3.3 Równoległa pętla po elementach w FEAPie . . . 47
3.4 Testy numeryczne ompFEAPa . . . 50
3.4.1 Test kostki . . . 50
3.4.2 Test nieliniowej powłoki . . . 54
3.4.3 Model „Roofline” . . . 56
3.4.4 Podsumowanie . . . 57

4 Równoległe rozwiązywanie układów równań liniowych 58
4.1 Wprowadzenie . . . 58
4.1.1 Podstawowe algorytmy rozwiązywania układów równań liniowych 58
4.1.2 Inne rozkłady macierzy . . . 59
4.1.3 Przechowywanie macierzy rzadkiej w pamięci komputera . . . 60
4.1.4 Algorytmy rozwiązywania rzadkich układów równań liniowych . . 61
4.1.5 Grafowy model zadań dla rzadkiej faktoryzacji . . . 62
4.1.6 Superwęzły . . . 65
4.1.7 Rzadka faktoryzacja bazująca na zadaniach . . . 67
4.1.8 Wybór elementu głównego w równoległych rzadkich faktoryzacjach 69
4.1.9 Algorytm zagnieżdżonego podziału . . . 70
4.1.10 Inne aspekty optymalizacji faktoryzacji numerycznej . . . 72
4.2 Porównanie różnych implementacji . . . 72
4.2.1 Testowane implementacje . . . 72
4.2.2 Jednoczesna wielowątkowość . . . 75
4.2.3 Koligacja wątków . . . 75
4.2.4 Permutacja wierszy macierzy . . . 76
4.2.5 Rezultaty testów . . . 81
4.3 Paralelizacja w solwerze HSL MA86 . . . 83
4.3.1 Sposób działania . . . 83
4.3.2 Czas wykonania i skalowalność zadań w fazie faktoryzacji . . . 86
4.3.3 Przyspieszenie obliczeń i poprawa skalowalności . . . 86
4.3.4 Dostosowanie parametrów uporządkowania . . . 88
4.3.5 Dyskusja innych sposobów przyspieszenia . . . 88
4.3.6 Podsumowanie przyspieszenia solwera HSL MA86 . . . 89
4.4 Solwer z mieszaną precyzją . . . 91
4.4.1 Iteracyjne poprawianie rozwiązania . . . 91
4.4.2 Iteracyjne poprawianie rozwiązania dla solwera z mieszaną precyzją 92

5 Rozproszone rozwiązywanie układów równań liniowych 95
5.1 Wprowadzenie . . . 95
5.2 Uzupełnienie Schura . . . 96
5.2.1 Wprowadzenie . . . 96
5.2.2 Uzupełnienie Schura przy równoległym rozwiązywaniu układu równań liniowych dla MES . . . 97
5.2.3 Cześciowa faktoryzacja do obliczania uzupełnienia Schura . . . 99
5.2.4 Implementacja częściowej faktoryzacji w HSL MA86 . . . 99
5.2.5 Sekwencyjna częściowa faktoryzacja na pojedynczym węźle dla dwóch subdomen . . . 102
5.3 Rozwiązywanie bezpośrednie układów równań na wielu węzłach . . . 104
5.3.1 Testy WSMP . . . 104
5.3.2 Własny solwer na klaster . . . 105
5.3.3 Hierarchiczna faktoryzacja macierzy dla interfejsów . . . 110
5.3.4 Hierarchiczna faktoryzacja uzupełnienia Schura na wielu węzłach . 114
5.3.5 Solwer hierarchiczny z mieszaną precyzją na wielu węzłach . . . 117

5.4 Iteracyjne rozwiązywanie układów równań na wielu węzłach 119
5.4.1 Wprowadzenie do FETI 119
5.4.2 Wprowadzenie do FETI-DP 122
5.4.3 Implementacja FETI-DP 126
5.4.4 Wnioski 129

6 Przykłady rozwiązywania układów równań liniowych 130
6.1 Pianka korundowa 130
6.1.1 Niesymetryczna macierz konstytutywna 131
6.1.2 Wyniki testów solwera na jednym węźle 133
6.1.3 Wyniki testów solwera na wielu węzłach 134
6.2 Ceramiczny kompozyt 135
6.2.1 Wyniki testów solwera na jednym węźle 137
6.2.2 Wyniki testów solwera na wielu węzłach 138

7 Podsumowanie 140

Literatura 143

Dodatki 151

A Dodatek A
Instrukcja dodawania elementu do ompFEAP 151

B Dodatek B
Algorytm podziału grafu 153

C Dodatek C
Program Verde 156

D Dodatek D
Trójwymiarowy izoparametryczny element ośmiowęzłowy 157

E Dodatek E
Redukcja macierzy elementowych 160

F Dodatek F
FGMRES 162

G Dodatek G
PARFEAP i Warp3D 164

H Dodatek H
Procedury opakowujące bibliotekę PAPI 169

I Dodatek I
Blokowa faktoryzacja macierzy symetrycznej 178

Lista pojęć i skrótów

CF Częściowa faktoryzacja, patrz rozdz. 5.2.3. 99

CSC Compressed Sparse Column, patrz rozdz. 4.1.3. 60

CSR Compressed Sparse Row, patrz rozdz. 4.1.3. 60

DAG Directed Acyclic Graph, patrz rozdz. 4.1.5. 63

DM-MIMD Distributed Memory - Multiple Instruction Multiple Data, patrz rozdz. 2.1. 16

FEAP Finite Element Analysis Program [25]. 12

FLOP Floating point operation, operacje zmiennoprzecinkowe. 23

FLOPS Floating point operation per second, operacje zmiennoprzecinkowe na sekundę. 23

GCC GNU Compiler Collection. 43

GPU Graphics Processing Unit. 42

GRAFEN Klaster obliczeniowy, który znajduje się w IPPT PAN [29]. 25

HSL Harwell Subroutine Library [45]. 12

HSL MA86 Wielowątkowy solwer dla macierzy rzadkich z biblioteki HSL, patrz rozdz. 4.2.1. 12

HSL MA97 Wielowątkowy multifrontalny solwer dla macierzy rzadkich z biblioteki HSL, patrz rozdz. 4.2.1. 73

Klaster Maszyna obliczeniowa złożona z wielu komputerów (węzłów obliczeniowych), znajdujących się w tym samym miejscu, połączonych szybką siecią. 12

MA64 Wielowątkowy solwer dla macierzy gęstych z biblioteki HSL. 93

MES Metoda Elementów Skończonych. 11

METIS Biblioteka służąca do dzielenia grafów oraz znajdywania przenumerowania wierszy/kolumn macierzy [74]. 65

MKL Math Kernel Library [76]. 12

MPI Message Passing Interface, patrz rozdz. 2.5.1. 12

MUMPS a MUltifrontal Massively Parallel sparse direct Solver [78]. 41

NP Nondeterministic Polynomial; problem, dla którego rozwiązanie można zweryfikować w czasie wielomianowym. 70

ompFEAP wielowątkowa wersja programu FEAP. 40

OpenMP Open Multi-Processing, patrz rozdz. 2.4.1. 11

PAPI Performance Application Programming Interface [86]. 24

PARDISO Parallel Direct Solver [98]. 12

PETSc Portable, Extensible Toolkit for Scientific Computation [90]. 42

Procesor część komputera odpowiedzialna za wykonywanie instrukcji programów. 12

RAM RAM (ang. Random-access memory) pamięć operacyjna komputera. 15

Rdzeń Podstawowa jednostka procesora wykonująca instrukcje oraz operacje arytmetyczne. 12

SM-MIMD Shared Memory - Multiple Instruction Multiple Data, patrz rozdz. 2.1. 16

Solid-shell element powłokowy wykorzystujący tylko przemieszczeniowe stopnie swobody. 40

Solwer Program służący do rozwiązywania układów równań liniowych. 43

WSMP Watson Sparse Matrix Package [37]. 12

Węzeł obliczeniowy Komputer bedący częścią klastra. 13

1 Wstęp

1.1 Wprowadzenie

Modele obliczeniowe dla zaawansowanych zagadnień inżynierskich osiągają znaczne rozmiary, np. nawet miliardów stopni swobody (ang. *giga dofs problems*) w przypadku kompozytowych kadłubów samolotów, i ich obliczanie a pomocą klasycznych seryjnych algorytmów i pojedynczych komputerów jest niewykonalne. Z tego względu powstał nowoczesny dział informatyki stosowanej dotyczący algorytmów wielowątkowych i rozproszonych - niniejsza rozprawa należy do tego obszaru badań.

Motywacja do podjęcia tematu niniejszej rozprawy wywodzi się z badań nad wielowarstwowymi powłokami kompozytowymi, które prowadzi zespół, z którym współpracuje autor niniejszej rozprawy. Zespół rozwija metodologię i tworzy oprogramowanie umożliwiające analizę kompozytowych konstrukcji powłokowych, przy czym opracowywane są zarówno elementy skończone o nieliniowej kinematyce i jak i wieloskalowe modele materiału, uwzględniające wielowarstwowość przekroju i mikrostrukturę materiału.

Analizy tego typu konstrukcji powłokowych najczęściej przeprowadza się zakładając sprężyste właściwości materiałów kompozytowych, patrz np. [33] i [62]. Aby usunąć to ograniczenie i uzyskać możliwość modelowania złożonych zachowań materiału kompozytowego, np. uwzględnienia różnych mechanizmów jego zniszczenia, stosowane jest wieloskalowe bezpośrednie modelowanie Metodą Elementów Skończonych (MES), które polega na:

1. wykorzystaniu trójwymiarowych elementów skończonych i trójwymiarowych praw konstytutywnych do modelowania mikrostuktury materiału,
2. zastosowaniu elementów powłokowych typu *solid-shell*, tzn. mających tylko translacyjne stopnie swobody, na poziomie warstwy kompozytu. W elementach tego typu także używane są trójwymiarowe prawa konstytutywne.

Takie podejście zapewnia bardzo dobrą dokładność modelowania jednak prowadzi do dużych i wymagających obliczeniowo modeli, dla których pojedynczy komputer i seryjne kody są zbyt małe i zbyt nieefektywne. Dlatego trzeba wykorzystać wielordzeniowość procesora i wiele węzłów obliczeniowych (klastra), co jednak wymaga specjalistycznego oprogramowania. W programach MES wykorzystywane są następujące techniki przyspieszania obliczeń:

1. wielowątkowość w obliczeniach na pojedynczym węźle obliczeniowym stosowana jest w celu wykorzystania wielu rdzeni procesora, i dotyczy zarówno pętli po elementach jak i solwera.

   Do paralelizacji pętli po elementach stosowany może być np. OpenMP [83], który został wykorzystany w ostatnich latach do szerokiej gamy problemów, takich jak np.: symulacja uderzeń [85], rozwiązywanie równań Navier-Stoke's [107], kalibracja modelu dla przepływu wód gruntowych [105], i wiele innych. Przyspieszenie uzyskane dzięki wykorzystaniu wielu rdzeni, wynosiło w tych pracach ok. 3.5 dla

4 rdzeni, 10.6 dla 12 rdzeni i 11 dla 16 rdzeni, co ilustruje stopień użyteczności OpenMP.

Rozwiązywanie układu równań liniowych jest jednym z najbardziej czasochłonnych procesów w MES. Ze względu na to, że układy równań dla problemów konstrukcji powłokowych i nieliniowych materiałów są źle uwarunkowane, do ich rozwiązania zwykle wykorzystuje się metody bezpośrednie (nieiteracyjne). Dla macierzy rzadkich, które otrzymujemy w MES, najbardziej popularne są następujące solwery wielowątkowe: WSMP [37], PARDISO [98], Intel MKL PARDISO [76], HSL MA86 [43].

2. rozpraszanie obliczeń na wiele węzłów obliczeniowych, do którego stosuje się metodę dekompozycji obszaru i solwery hybrydowe.

   Metody dekompozycji obszaru (ang. *Domain Decomposition (DD) methods*) rozwiązują problemy brzegowe dzieląc je na mniejsze problemy i problem interfejsu. Praca rozdzielana jest na węzły obliczeniowe a komunikacja między procesami (węzłami) jest zrealizowana za pomocą np. MPI [80]. Na węzłach obliczeniowych stosowany może być np. OpenMP. Powstało wiele różnych metod dekompozycji obszaru, niektóre zostały wykorzystane również do rozwiązywania problemów konstrukcji powłokowych, np. [48], [106].

   Solwery hybrydowe wykorzystują wielowątkowość, np. OpenMP, i przesyłanie komunikatów, np. MPI; najbardziej popularne to PARDISO, Intel MKL PARDISO, WSMP i SuperLU_DIST.

Niniejsza rozprawa dotyczy obu powyższych technik przyspieszania obliczeń w celu efektywnego wykorzystania procesorów wielordzeniowych i klastra do przeprowadzania obliczeń MES problemów brzegowych z dziedziny mechaniki ciał trójwymiarowych i konstrukcji powłokowych.

1.2 Cel rozprawy

Głównym celem niniejszej rozprawy jest stworzenie efektywnych i skalowalnych algorytmów wielowątkowych (z użyciem OpenMP [83]) i rozproszonych (z użyciem MPI [80]), w celu zwiększenia efektywności Metody Elementów Skończonych (MES). Przeanalizowano dotychczasowy stan wiedzy w dziedzinie zrównoleglania i zaproponowano ulepszenia w dwóch obszarach kodu MES:

- pętla po elementach:

  Został opracowany algorytm zrównoleglenia pętli po elementach i redukcji macierzy elementowych do macierzy globalnej z wykorzystaniem OpenMP.

  Algorytm został zaimplementowany w skomplikowanym pod względem programistycznym kodzie FEAP [25] rozwijanym przez Prof. R. L. Taylora z Uniwersytetu Kalifornijskiego w Berkeley USA i wykorzystywanym w wielu ośrodkach akademickich na świecie. Pełny opis opracowanego algorytmu podano w artykule [54] opublikowanym w Computational Mechanics w 2015 r.

- rozwiązywanie układu równań liniowych:
    1. W ramach ulepszania rozwiązywania układu równań na pojedynczym węźle obliczeniowym, zaproponowano szereg ulepszeń dla solwera HLS MA86 [43], takich jak dynamiczne określenie wielkości prywatnej puli zadań dla wątku, wyłączenie opcji maksymalizacji lokalności pamięci podręcznej podczas fazy analizy, oraz określenie optymalnej liczby separatorów. Wyniki szczegółowe opublikowano w artykule [55] w wydawnictwie Taylor & Francis/CRC Press w 2016 r.

       Następnie opracowano wersję tego solwera w mieszanej precyzji, przy czym faktoryzację wykonuje się w pojedynczej precyzji, co istotnie ją przyspiesza, i iteracyjnie poprawia się rozwiązanie stosując podwójną precyzję.
    2. Opracowano solwer na klaster bazujący na dekompozycji obszaru i obliczaniu uzupełnienia Schura za pomocą techniki częściowej faktoryzacji (ang. *partial factorization*) - tę ostatnią zaimplementowano w solwerze HLS MA86, co jest zadaniem nietrywialnym z uwagi na jego skomplikowanie i techniki programistyczne w nim zastosowane. Dodatkowo zastosowano hierarchiczną faktoryzację oraz mieszaną precyzję wraz z iteracyjnym poprawianiem. Publikacja prezentująca nowy solwer jest obecnie w przygotowaniu [56].

1.3 Zawartość rozprawy

Rozprawa doktorska składa się z siedmiu zasadniczych rozdziałów:

1. *Wstęp.* Wstęp zawiera wprowadzenie do tematyki rozwijanej w rozprawie, cel pracy oraz metody, dzięki którym zostanie on osiągnięty.

2. *Obliczenia równoległe.* Przedstawione zostały podstawowe informacje dotyczące architektur komputerów, na których wykonuje się obliczenia równoległe. Opisana została teoria dotycząca miar zrównoleglenia, a także dwie wiodące technologie dla programowania równoległego i rozproszonego - OpenMP [83] i MPI [80].

3. *Zrównoleglenie pętli po elementach w FEAPie.* Generowanie macierzy elementowych i tworzenie macierzy globalnej może zajmować większą część czasu obliczeń MES, dlatego przedstawiono wielowątkowy algorytm obliczający macierze elementowe i redukujący je do macierzy globalnej wykorzystujący OpenMP. Implementacja została wykonana w programie FEAP [25]. Zamieszczono również przegląd literatury dotyczący redukcji macierzy elementowych, oraz umiejscowiono badania własne na tle prac innych badaczy.

4. *Równoległe rozwiązywanie układów równań liniowych.* Rozdział jest poświęcony solwerom wielowątkowym do rozwiązywania rzadkich układów równań liniowych na jednym węźle obliczeniowym. Przeprowadzono analizę efektywności oraz zapotrzebowania na pamięć 5-ciu takich solwerów. Badano wpływ przenumerowania kolumn i wierszy macierzy na czas faktoryzacji, oraz wybrano najlepszy solwer. Zaproponowano szereg ulepszeń dla solwera HLS MA86 [43]. Dodatkowo badano solwer w mieszanej precyzji.

5. *Rozproszone rozwiązywanie układów równań liniowych.* Rozdział dotyczy solwerów działających na wielu węzłach obliczeniowych i korzystających z przesyłania komunikatów MPI. Omówiono metodę dekompozycji obszaru (ang. *Domain Decomposition*) oraz obliczenie uzupełnienia Schura za pomocą częściowej faktoryzacji (ang. *partial factorization*). Posłużyły one do stworzenia dwóch algorytmów:

   a. wielowątkowego sekwencyjnego algorytmu do redukcji zapotrzebowania na pamięć podczas faktoryzacji macierzy na pojedyńczym węźle, który zostanie porównany z algorytmem bazującym na mieszanej precyzji.

   b. rozproszonego algorytmu na wiele węzłów wykorzystujący hierarchiczną faktoryzację macierzy blokowej, który zostanie porównany z solwerem WSMP [37]. Algorytm ten opracowano w podwójnej i mieszanej precyzji.

   Dodatkowo, dla porównania, zaprezentowano metodę FETI-DP wykorzystującą metody bezpośrednie na węzłach i solwer iteracyjny PCG dla równania interfejsów.

6. *Przykłady rozwiązywania układów równań liniowych.* Zaimplementowane algorytmy zostały przetestowane dla wybranych zadań MES, w celu weryfikacji ich efektywności, skalowalności oraz zapotrzebowania na pamięć. Przykładowe zadania to obliczenia dla ceramicznego kompozytu i pianki korundowej; wyniki opublikowano w artykule [55]. Największy obliczony model zawierał ponad 30 mln niewiadomych.

7. *Podsumowanie.* Omówiono rezultaty rozprawy, z podkreśleniem elementów oryginalnych. Podano listę opracowanych implementacji.

2 Obliczenia równoległe

2.1 Architektura komputerów

Większość współczesnych komputerów opartych jest na tzw. architekturze von Neumanna. Składa się ona z trzech podstawowych komponentów: procesora, pamięci RAM oraz urządzeń wejścia i wyjścia. Każdy z tych komponentów może być łączony z innymi na wiele różnych sposobów - tworząc architekturę komputera.

Aby sklasyfikować architektury komputerowe wymyślono różne taksonomie. Najpopularniejszą jest taksonomia Flynna z 1968 roku. Mimo, że powstała ponad pół wieku temu jej prostota i ogólność skłania do dalszego wykorzystywania podczas klasyfikacji architektur komputerowych. Taksonomia Flynna zakłada, że komputer przetwarza strumień danych i strumień instrukcji. W zależności od tego ile strumieni i jakiego rodzaju przetwarza, do innej grupy dana maszyna jest zaklasyfikowana. Biorąc powyższe pod uwagę występują podstawowe cztery grupy [92, s. 28]:

1. SISD - (ang. *Single Instruction, Single Data*) Klasyczne maszyny PC, już praktycznie nieprodukowane. Zawierają jeden procesor i jeden blok pamięci operacyjnej, w której znajduje się program - ciąg instrukcji wykonywanych sekwencyjnie.

2. SIMD - (ang. *Single Instruction, Multiple Data*) Maszyny, w których te same operacje wykonywane są jednocześnie na różnych danych. W dzisiejszych komputerach rozwiązanie to, jest stosowane w przypadku rozszerzeń strumieniowych dla pojedynczych procesorów, tj. SSE3 (*ang. Streaming SIMD Extensions 3*)

3. MISD - (ang. *Multiple Instruction, Single Data*) Maszyny wykonujące różne operacje na tych samych danych. Taką cechę mają systemy uczące się, które wykorzystują metodę gradientu stochastycznego, np. sieci neuronowe typu nieliniowego perceptronu z tzw. „regułą delta” jako strategię uczenia.

4. MIMD - (ang. *Multiple Instruction, Multiple Data*) Najważniejszy typ maszyn równoległych. Poszczególne procesory wykonują różne operacje na różnych danych. Dane te stanowią różne części tego samego zadania obliczeniowego. Dzisiejsze maszyny PC z procesorami wielordzeniowymi.

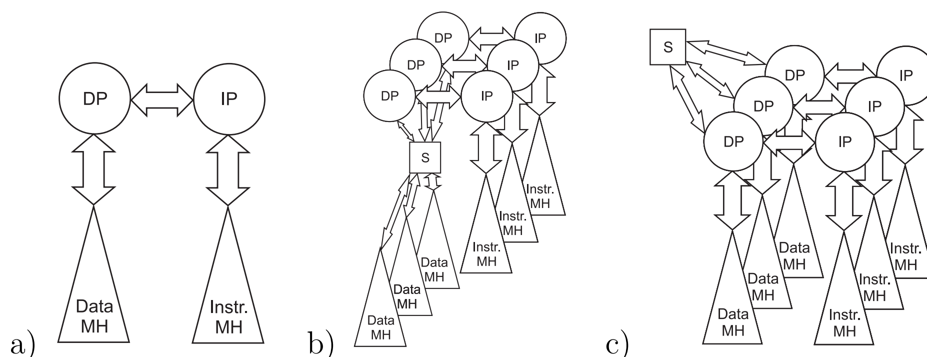
Istnieje także inny podział ze względu na dostęp do pamięci [92, s. 29]:

- SM - (ang. *global, shared memory*) Maszyny z pamięcią wspólną (współdzieloną, dzieloną, globalną), które mają tą samą przestrzeń adresową, programista nie wie, gdzie są przechowywane poszczególne struktury danych (obiekty), sięgając do nich zgodnie z regułami zasięgu obowiązującymi w stosowanym przez niego języku programowania.

- DM - (ang. *local, distributed memory*) Maszyny z pamięcią lokalną (rozproszoną), w których każdy procesor ma własną pamięć. Tę klasę tworzą również sieci komputerowe. W obu przypadkach to programista dokonuje podziału danych między jednostki obliczeniowe.

Oprócz powyższych podziałów istnieje również taksonomia przedstawiona przez Skillicorna [101]. Jej podstawowym założeniem jest to, że architektura stanowi połączenie pewnej liczby składników. Dzięki temu taksonomia Skillicorna nie służy do podziału na różne architektury, a do jej syntezy. Wyróżnia ona cztery abstrakcyjne składniki:

1. Procesor instrukcji (ang. *instruction processor*) - jednostka potrafiąca interpretować instrukcje.
2. Procesor danych (ang. *data processor*) - jednostka potrafiąca przetwarzać dane (np. wykonując operacje arytmetyczne).
3. Hierarchia pamięci (ang. *memory hierarchy*) - jednostka przetrzymująca dane.
4. Przekaźnik (ang. *switch*) - abstrakcyjna jednostka przekazująca dane.



Rysunek 2.1: Przykładowe architektury wg Skillcorna: a) maszyna SISD b) maszyna SM-MIMD c) maszyna DM-MIMD

W taksonomii Skillcorna jest możliwych około 30 różnym modeli, ale tylko 6 z nich ma sensowne znaczenie. Do tej szóstki należą maszyny z taksonomii Flynna (SISD, SIMD, MIMD, MISD) oraz maszyny działające tylko na danych (instrukcje są zawarte w rodzaju przetwarzanych danych) - np. uniprocesory dataflow oraz wieloprocesor dataflow.

W niniejszej pracy będą wykorzystywane dwa rodzaje maszyn SM-MIMD oraz DM-MIMD. Maszyny SM-MIMD to przede wszystkim większość dzisiejszych komputerów z procesorami, które posiadają wiele rdzeni. Z kolei maszyny DM-MIMD to np. klastry obliczeniowe, czyli zespół znajdujących się w pobliżu maszyn, najczęściej homogenicznych, połączonych szybką, dedykowaną siecią, które są jednolicie zarządzane i mogą być wykorzystywane do wspólnej pracy ([92, s. 45]). Obecnie klastry najczęściej składają się z wielu węzłów obliczeniowych, z których każdy jest maszyną typu SM-MIMD.

2.2 Miary zrównoleglenia obliczeń

Do oceny algorytmów równoległych wykorzystywane są różne miary. Podstawowymi miarami dla obliczeń równoległych są: (1) przyspieszenie, (2) efektywność, (3) sprawność, i (4) skalowalność. W niniejszym podrozdziale zostaną przedstawione definicje poszczególnych miar zrównoleglenia za [92] i [17], a także pewne wyniki teoretyczne (prawa), które uzyskano dla tych miar do tej pory.

Pomocne do tego będą pewne oznaczenia:

n - parametr opisujący wielkość zadania,

p - liczba użytych procesorów/rdzeni/procesów/wątków/,

$T(n, p)$ - czas wykonania algorytmu dla zadania o wielkości n na maszynie z p procesorami; **założenie:** algorytm został zaimplementowany w optymalny sposób, tzn. dla wszystkich par (n, p) czas ten jest możliwie najkrótszy,

$\beta(n, p)$ - udział czasu wykonania części sekwencyjnej w algorytmie o wielkości n na maszynie z p procesorami.

2.2.1 Prawo Amdahla i prawo Gustafsona-Barsisa

Definicja współczynnika przyspieszenia (ang. *speedup*) dla zadania o wielkości n, przy zrównolegleniu na p procesorów jest następująca:

$$S(n, p) = \frac{T(n, 1)}{T(n, p)}, \tag{2.1}$$

gdzie ze względu na założenie o optymalnej implementacji algorytmów musi być spełniona nierówność:

$$S(n, p) \leqslant p. \tag{2.2}$$

Gdyby powyższa nierówność nie była spełniona to znaczyłoby, że $T(n, 1) > pT(n, p)$. To z kolei oznacza, że wykorzystując algorytm wykonujący zadanie dla $T(n, p)$ i uruchomiając go sekwencyjnie p razy, uzyskanoby algorytm, którego czas wykonania byłby mniejszy niż dla $T(n, 1)$, a to przeczy definicji $T(n, p)$ zakładającej optymalność implementacji.

Czas wykonania części sekwencyjnej jest stały dla wszystkich maszyn, niezależnie od liczby procesorów p, czyli:

$$\beta(n, p) \cdot T(n, p) = const,\ p = 1, 2, 3, \dots\ . \tag{2.3}$$

Teraz zostanie wyznaczony czas wykonania algorytmu po zrównolegleniu na p procesów, przy założeniu, że jest znany czas wykonania programu na jednym procesorze $T(n, 1)$ oraz udział w tym czasie części sekwencyjnej $\beta(n, 1)$ (tzn. części $T(n, 1)$, której nie da się zrównoleglić). Dodatkowo zakłada się, że część równoległa $1 - \beta(n, 1)$ może być zrównoleglona idealnie, tzn. rozłożona równomiernie, bez żadnych narzutów na synchronizację i komunikację między dowolną liczbę procesorów, czyli

$$(1 - \beta(n, p))T(n, p) = \frac{(1 - \beta(n, 1))T(n, 1)}{p}. \tag{2.4}$$

Czas wykonania obliczeń na maszynie równoległej składającej się z p procesorów, przy powyższych założeniach, wynosi:

$$T(n,p) = (\beta(n,p) + 1 - \beta(n,p)) \cdot T(n,p)$$
$$= \underbrace{\beta(n,1) \cdot T(n,1)}_{\text{z równ. (2.3) dla } p=1} + \underbrace{\frac{(1-\beta(n,1))T(n,1)}{p}}_{\text{z równ. (2.4)}}. \quad (2.5)$$

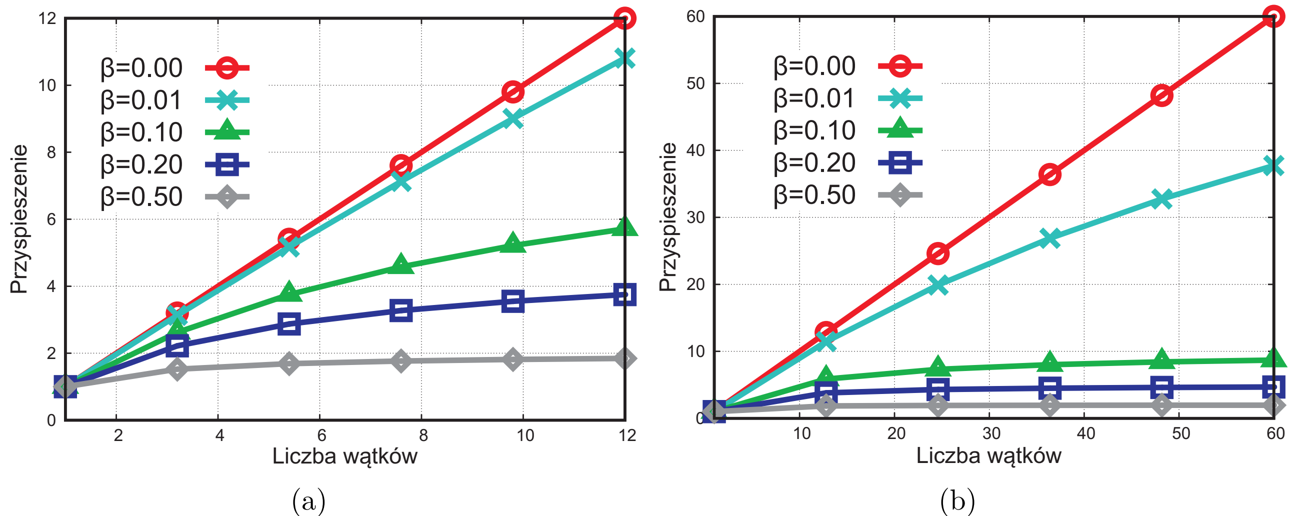
Po przekształceniach:

$$S(n,p) = \frac{T(n,1)}{T(n,p)} = \frac{1}{\beta(n,1) + \frac{(1-\beta(n,1))}{p}}. \quad (2.6)$$

Równanie (2.6) jest nazywane **prawem Amdahla** [1], z którego wynika, że dla każdego n przy $p \to \infty$

$$S(n,p) \to \frac{1}{\beta(n,1)}. \quad (2.7)$$

Oznacza to, że nawet przy użyciu dowolnie wielu procesorów, obliczeń nie da się przyśpieszyć bardziej, niż wynosi odwrotność udziału części sekwencyjnej w algorytmie wykonywanym na jednym procesorze. Na przykład jeśli udział ten wynosi 0.5, to algorytm można przyśpieszyć co najwyżej dwukrotnie, jeśli 0.10 to co najwyżej dziesięciokrotnie. Na rys. 2.2 przedstawiony jest wykres zależność oczekiwanego przyspieszenia z prawa Amdahla dla danej liczby wątków. Z rys. 2.2b dla $\beta = 0.10$ widać, że od ok. 30 wątków przyspieszenie nie wzrasta i jest stałe.



Rysunek 2.2: Wykres przyspieszenie wg prawa Amdahla. (a) dla 12 wątków i (b) dla 60 wątków.

Z drugiej strony udział części sekwencyjnej często zależy od wielkości zadania, co zauważył Gustafson w swoim artykule [40], krytykując założenia prawa Amdahla. W wielu

algorytmach czas spędzony w części sekwencyjnej jest stały i równy $\beta(n,1)\cdot T(n,1) = \beta_s \cdot T(1,1) \forall n$ (może być to związane na przykład z czytaniem danych), zaś czas spędzony w części równoległej jest proporcjonalny do wielkości zadania n, czyli

$$\beta(n,1) = \frac{\beta_s \cdot T(1,1)}{\beta_s \cdot T(1,1) + n \cdot (1-\beta_s) \cdot T(1,1)} = \frac{\beta_s}{\beta_s + n \cdot (1-\beta_s)} = \frac{1}{1 + n \cdot (\frac{1}{\beta_s} - 1)}. \tag{2.8}$$

W tym momencie, przy $n \to \infty$ jest

$$\beta(n,1) \to 0, \tag{2.9}$$

a to wspólnie z równ. (2.6) oznacza, że:

$$S(n,p) \to p \tag{2.10}$$

Widać z powyższych równań, że zwiększając rozmiar zadania można uzyskać przyspieszenie bliskie idealnego. Dlatego obliczenia równoległe są stosowane nie tylko w celu szybszego uzyskania wyników dla stałego wymiaru zadania, ale również w celu rozwiązania dużych (tzn. o dużej wymiarowości) przykładów tego samego zadania i o tym mówi prawo Gustafsona.

Aby wyprowadzić prawo Gustafsona wykorzystuje się własność z równ. (2.3). W szczególności wynika z niej, że:

$$\beta(n,1) \cdot T(n,1) = \beta(n,p) \cdot T(n,p), \tag{2.11}$$

a to podstawiając do równ. (2.5) da

$$T(n,p) = \beta(n,p) \cdot T(n,p) + \frac{T(n,1)}{p} - \frac{\beta(n,p) \cdot T(n,p)}{p}. \tag{2.12}$$

Stąd po obustronnym przemnożeniu przez p oraz przeniesieniu wyrazów z $T(n,p)$ na lewą stronę

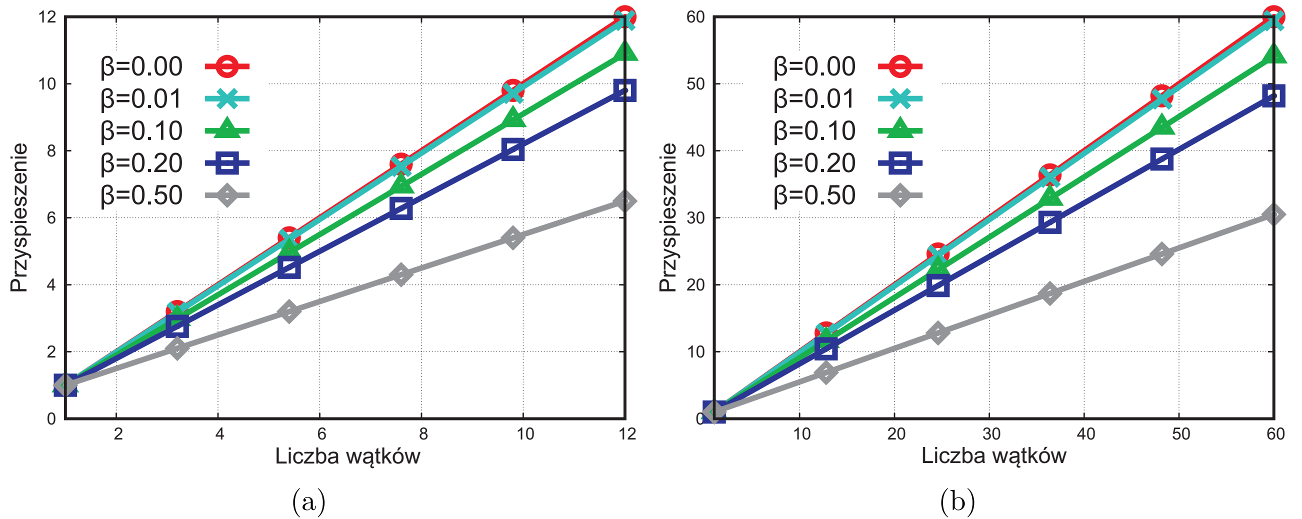
$$p \cdot T(n,p) - p \cdot \beta(n,p) \cdot T(n,p) + \beta(n,p) \cdot T(n,p) = T(n,1). \tag{2.13}$$

Wartość przyśpieszenia $S(n,p)$ uzyskanego dzięki zrównolegleniu, można otrzymać, dzieląc obydwie strony równ. (2.13) przez $T(n,p)$. Będzie ono zatem równe

$$S(n,p) = \frac{T(n,1)}{T(n,p)} = p - (p-1) \cdot \beta(n,p). \tag{2.14}$$

Równanie (2.14) nosi nazwę **prawa Gustafsona-Barsisa** i zostało ono przedstawione na rys. 2.3.

Warto tutaj zauważyć, że mimo iż oba prawa są wyprowadzone z tych samych zależności, tylko pozornie prowadzą do rozbieżnych wniosków. Według prawa Amdahla nasze przyspieszenie jest ograniczone, a według prawa Gustafsona-Barsisa przyspieszenie jest nieograniczone. Ale jak już wcześniej pokazano w równ. (2.8) - (2.10), przyspieszenie w



Rysunek 2.3: Wykres przyspieszenie wg prawa Gustafsona-Barsisa. (a) dla 12 wątków i (b) dla 60 wątków.

prawie Amdahl'a jest ograniczone tylko wtedy, gdy część sekwencyjna nie dąży do zera wraz z $n \to \infty$, a więc nie ma tutaj sprzeczności.

Podsumowując, prawo Amdahla pokazuje, że dla danej wielkości zadania n istnieje maksymalne przyspieszenie, a prawo Gustafsona-Barsisa pokazuje, że dla danego czasu na wykonanie zadania, można wykonać zadanie dowolnie duże. Obrazuje to rys. 2.4.

W obu podejściach pomija się złożoność związaną z prowadzeniem obliczeń równoległych, a także zakłada się, że część równoległą obliczeń da się przyspieszyć w stopniu maksymalnym, tj. p razy. Wskutek tego przyspieszenia te są większe niż uzyskiwane w praktyce. Karp i Flatt zaproponowali eksperymentalne wyznaczenie części sekwencyjnej obliczeń algorytmu równoległego, która pozwala dokonać oceny uzyskiwanych przyspieszeń. W niniejszej pracy nie stosowano tej miary, jej opis można znaleźć w [58].

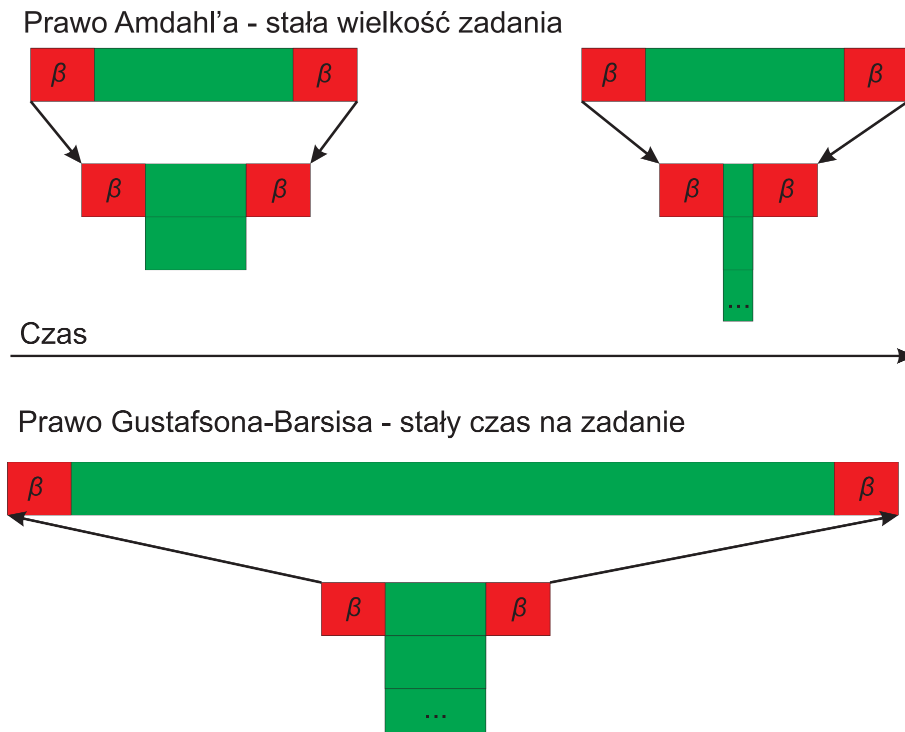
Współczynnik przyśpieszenia często podawany jest w wersji przeskalowanej w stosunku do przyśpieszenia idealnego. Otrzymuje się wówczas względny współczynnik przyspieszenia

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T(n,1)}{p \cdot T(n,p)}, \tag{2.15}$$

zwany także **wydajnością** lub **efektywnością mocną** (ang. *strong efficiency*). Mocną ponieważ sprawdzana jest efektywność dla stałej wielkości zadania n. **Efektywność słabą** (ang. *weak efficiency*) jest nazywany związek

$$E_w(n,p) = \frac{T(n,1)}{T(p \cdot n, p)}. \tag{2.16}$$

W tym przypadku każdy procesor (wątek, rdzeń) dostaje odpowiednio tyle samo obliczeń do wykonania.



Rysunek 2.4: Porównanie prawa Amdahl'a i Gustafsona-Barsisa.

2.2.2 Skalowalność

Jednym z najważniejszych pojęć w dziedzinie obliczeń równoległych jest skalowalność. Jest to własność systemu (sprzętu i oprogramowania) polegająca na elastycznym dostosowaniu się do zwiększonej liczby procesorów (wątków, rdzeni). Elastyczne dostosowywanie się oznacza tutaj zachowanie tej samej wydajności. Niech $C(n,p)$ będzie kosztem algorytmu o wielkości n, realizowanego na maszynie z p procesorami, zdefiniowanym następująco:

$$C(n,p) = p \cdot T(n,p). \tag{2.17}$$

Koszt to czas obliczeń wykonywany przez wszystkie procesory. Minimalną wartością kosztu jest złożoność $T(n,p)$, którą można uważać za koszt obliczeń sekwencyjnych przy użyciu jednego procesora. Równomierne rozłożenie operacji między procesory zapewnia osiągnięcie wydajności równej 1 oraz przyspieszenia równego p. Uzyskanie równości kosztów $p \cdot T(n,p) = T(n,1)$ jest jednak trudne. Wszelkie operacje związane np. z komunikacją i synchronizacją procesorów nie występują bowiem w koszcie $T(n,1)$, a jedynie w koszcie $p \cdot T(n,p)$. Równość kosztów może więc wystąpić wtedy, gdy procesory wykonujące algorytm równoległy nie komunikują się ze sobą (co zachodzi rzadko) lub komunikacja jest na tyle szybka, że jej koszty są do pominięcia. Ponieważ w praktyce koszty te nie są równe, ich różnicę definiuje się jako koszt organizacji obliczeń równoległych

$$C^o(n,p) = p \cdot T(n,p) - T(n,1). \tag{2.18}$$

W skład kosztu $C^o(n,p)$ wchodzą koszty obliczeń nadmiarowych, koszty operacji pustych wynikających z bezczynności procesorów, zwiększone koszty odwołań do pamięci wspólnej oraz koszty komunikacji między procesorami.

Po wyznaczeniu z równ. (2.18) $p \cdot T(n,p) = C^o(n,p) + T(n,1)$ i po wstawieniu do równ. (2.15) otrzyma się

$$E(n,p) = \frac{T(n,1)}{C^o(n,p) + T(n,1)} = \frac{1}{1 + \frac{C^o(n,p)}{T(n,1)}}. \tag{2.19}$$

Ponieważ narzuty na komunikację rosną ze wzrostem liczby procesorów p, równ. (2.19) pokazuje, że wydajność maleje gdy zwiększa się liczbę wykorzystywanych procesorów, nie zmieniając wielkości zadania. Przy czym, gdy zwiększa się wielkość zadania, narzuty $C^o(n,p)$ rosną najczęściej wolniej, niż czas $T(n,1)$ i wówczas sprawność rośnie.

Przy założeniu, że wydajność systemu równoległego powinna być równa pewnej stałej e, $0 < e \leqslant 1$, tj.:

$$e = \frac{1}{1 + \frac{C^o(n,p)}{T(n,1)}}, \tag{2.20}$$

wynika, że

$$T(n,1) = \frac{e}{1-e} \cdot C^o(n,p) \tag{2.21}$$

Równanie (2.21) pokazuje, że aby utrzymać tę samą wydajność, po zwiększeniu liczby wykorzystywanych procesorów p, należy zwiększyć wielkość zadania n. Dla danej liczby procesorów można więc wyznaczyć z równ. (2.21) wielkość zadania, przy której zachowana jest ta sama wydajność. Związek ten określa tzw. funkcję stałej wydajności (ang. *isoefficiency function*), która jest funkcją monotonicznie rosnącą. Na jej podstawie ocenia się skalowalność systemu.

Funkcja izoefektywności określa szybkość wzrostu wielkości zadania, tak aby utrzymać stałą wydajność. Jeśli funkcja ta rośnie wolno, to system równoległy jest łatwo skalowalny. Niewielki wzrost rozmiaru zadania wystarcza bowiem do efektywnego wykorzystania możliwości obliczeniowych rosnącej liczby procesorów. Im większy jest wzrost rozmiaru problemu gwarantujący stałą wydajność, tym system jest gorzej skalowalny. W przypadku skrajnym, przy wykładniczym wzroście rozmiaru problemu, żądaną wydajność uzyskuje się dla problemów o olbrzymich rozmiarach, dla których dane wejściowe nie mieszczą się w pamięciach operacyjnych procesorów. Uwzględniając objętość pamięci, można wówczas wyznaczyć zakres liczby procesorów, w których system jest skalowalny. Istnieją także systemy nieskalowalne, w których nie da się utrzymać wydajności niezależnie od szybkości wzrostu rozmiaru problemu.

Na zakończenie tego podrozdziału, warto zauważyć, że wielkości zdefiniowane wcześniej, tj. złożoność, przyspieszenie, koszt i wydajność, na podstawie których ocenia się algorytmy równoległe, są funkcjami dwóch zmiennych - liczby procesorów p i rozmiaru problemu n. Porównanie algorytmów ze sobą i wybór algorytmu najlepszego odbywa się przez porównanie tych funkcji, co generalnie nie jest zadaniem trywialnym. Dlatego stworzono modele, które mają za zadanie ułatwić rozpoznanie wydajności algorytmów,

jednym z nich jest model „Roofline”, który zostanie przedstawiony w następnym podrozdziale i wykorzystany w dalszej części pracy.

2.3 Ograniczenia sprzętowe

2.3.1 Wprowadzenie

Maszny równoległe są opisywane za pomocą dwóch zmiennych: (1) maksymalnej do wykonania liczby operacji zmiennoprzecinkowych na sekundę (ang. *FLOPS*), (2) maksymalnej liczby przetransferowanych bajtów z/do pamięci głównej na sekundę (ang. *bandwidth*). Obie zmienne wyznaczają nie tyle możliwości sprzętu, ale przede wszystkim jego ograniczenia. Model „Roofline” [113] łączy obie zmienne w sposób graficzny, dzięki czemu łatwiej jest zrozumieć, czy badany algorytm wykorzystuje w pełni możliwości sprzętu. Do opisu powyższego modelu definiuje się pojęcie natężenia operacji (ang. *operational intensity*), znanego w literaturze również jako natężenie arytmetyczne (ang. *arithmetic intensity*).

2.3.2 Natężenie operacji

Niech $W(n,p)$ będzie oznaczało wykonaną pracę przez dany algorytm na p procesorach dla rozmiaru zadania n. Praca to liczba operacji, które potrzebuje wykonać procesor aby ukończyć algorytm dla danego zadania. Operacje mogą być rozumiane, jako porównania, operacje arytmetyczne na liczbach całkowitych, czy operacje zmiennoprzecinkowe. W tej pracy jako operację uważa się operację zmiennoprzecinkową wykonaną przez rdzeń procesora. Z tym, że jedna operacja dodawania wykonana np. przez AVX (ang. *Advanced Vector Extensions*) dla wektora składającego się z czterech liczb podwójnej precyzji, jest liczona jako 4 operacje. Tak zdefiniowana praca $W(n,p)$ odpowiada operacji zmiennoprzecinkowej (ang. *FLOP*). Zmienna $W(n,p)$ to własność algorytmu i nie jest zależna od platformy na jakiej algorytm jest wykonywany. Wydajność $P(n,p)$ to praca podzielona przez czas trwania algorytmu, czyli $P(n,p) = W(n,p)/T(n,p)$, co odpowiada liczbie operacji zmiennoprzecinkowych na sekundę (ang. *FLOPS*).

Niech $Q(n,p)$ będzie liczbą bajtów, którą przetransferowano do/z pamięci głównej w trakcie wykonania algorytmu na p procesorach dla rozmiaru zadania n. W przeciwieństwie do W, wartość Q jest silnie związana z platformą na jakiej jest wykonywany algorytm. W szczególności duży wpływ ma hierarchia pamięci, jaka jest stosowana na konkretnej maszynie. Jako $B(n,p)$ oznaczono transfer danych z/do pamięci na sekundę (ang. *bandwidth*), wyznaczany z zależności $B(n,p) = Q(n,p)/T(n,p)$.

Można teraz zdefiniować natężenie operacji, jako

$$I(n,p) = \frac{W(n,p)}{Q(n,p)}, \tag{2.22}$$

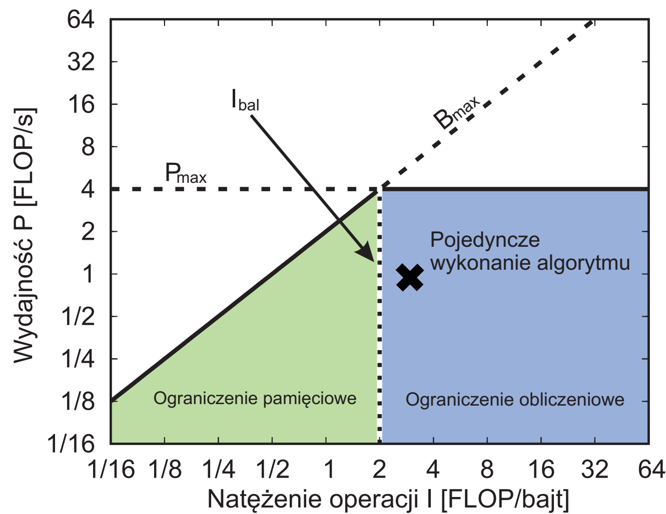
czyli natężenie operacji jest rozumiane jako liczbę wykonanych operacji na bajt pobrany lub zapisany do pamięci głównej.

2.3.3 Model „Roofline”

Model „Roofline” [113] graficznie przedstawia wydajność algorytmu P oraz jego natężenie operacji I dla określonej maszyny z pewną maksymalną wydajnością P_{max} oraz maksymalnym transferem danych B_{max}. Model ten można przedstawić za pomocą związku:

$$P(n,p) \leqslant min\{P_{max}, I(n,p) \times B_{max}\}. \tag{2.23}$$

Przedstawienie graficzne modelu zostało zaprezentowane na rys. 2.5. Na osi x w skali logarytmicznej jest przedstawione natężenie operacji. Na osi y także w skali logarytmicznej oznaczono wydajność. Dla pojedynczego wykonania pewnego algorytmu można otrzymać jego wydajności P oraz jego natężenie operacji I i będzie to jednym punktem na wykresie. Przecięcie ograniczenia wydajności P_{max} z ograniczeniem transferu danych B_{max}, występuje dla wartości natężenia operacji równej $I_{bal} = P_{max}/B_{max}$. Obliczenia z tą intensywnością są zrównoważone w sensie Kunga [64]. Obliczenia, dla których zachodzi $I \leqslant I_{bal}$ są ograniczone pamięciowo, a dla których jest $I \geqslant I_{bal}$ są ograniczone obliczeniowo.



Rysunek 2.5: Przykładowy wykres modelu „Roofline” dla $B_{max} = 2$ i $P_{max} = 4$, dla których $I_{bal} = 2$.

2.3.4 Wyznaczenie modelu „Roofline”

Nowoczesne maszyny równoległe wspierają sprzętowo monitorowanie jej wydajności. Platformy z procesorami firmy Intel posiadają specjalnie dedykowaną ku temu jednostkę - *Performance Monitoring Unit* (PMU), która zawiera zestaw rejestrów maszynowych (ang. *machine specific register* MSR). Rejestry można tak zaprogramować, aby zliczały pewne zdarzenia występujące w trakcie pracy procesora, tj. wykonane instrukcje, liczbę cykli procesora, próby dostępu do pamięci podręcznej itd. Biblioteka PAPI (*Performance Application Programming Interface*) [86], która została wykorzystana w tej pracy, jest w stanie odczytywać wyniki z tych rejestrów i udostępniać je użytkownikowi. Biblioteka ta

zawiera dostęp zarówno do zdarzeń predefiniowanych, takich które są obliczane na podstawie zdarzeń natywnych procesora, jak również można uzyskać bezpośrednio dostęp do zdarzeń natywnych i samodzielnie obliczać potrzebne metryki. Autor niniejszej pracy zaimplementował z wykorzystaniem biblioteki PAPI procedurę, która zlicza wszystkie wykonane określone operacje dla określonego rdzenia. Opis tej implementacji znajduje się w dodatku H.

W tej pracy korzystano z klastra obliczeniowego GRAFEN [29], który znajduje się w IPPT PAN. Jeden węzeł obliczeniowy zawiera 2 procesory Intel Xeon X5650 2.66 GHz z 6 rdzeniami i 24GB DDR3 1333MHz pamięci.

Maksymalną możliwą wydajność P_{max} uzyskano za pomocą testu LINPACK, który został zaimplementowany w bibliotece MKL [76]. Test ten jest wykorzystywany do tworzenia listy najszybszych i największych maszyn obliczeniowych TOP500 (top500.org). Natomiast maksymalny możliwy transfer danych B_{max} uzyskano za pomocą testu STREAM [102]. Oba testy wykonano zarówno dla wykonania sekwencyjnego oraz wykonania z 12 wątkami, po jednym wątku na każdym rdzeniu; otrzymane wyniki przedstawiono w tab. 2.1.

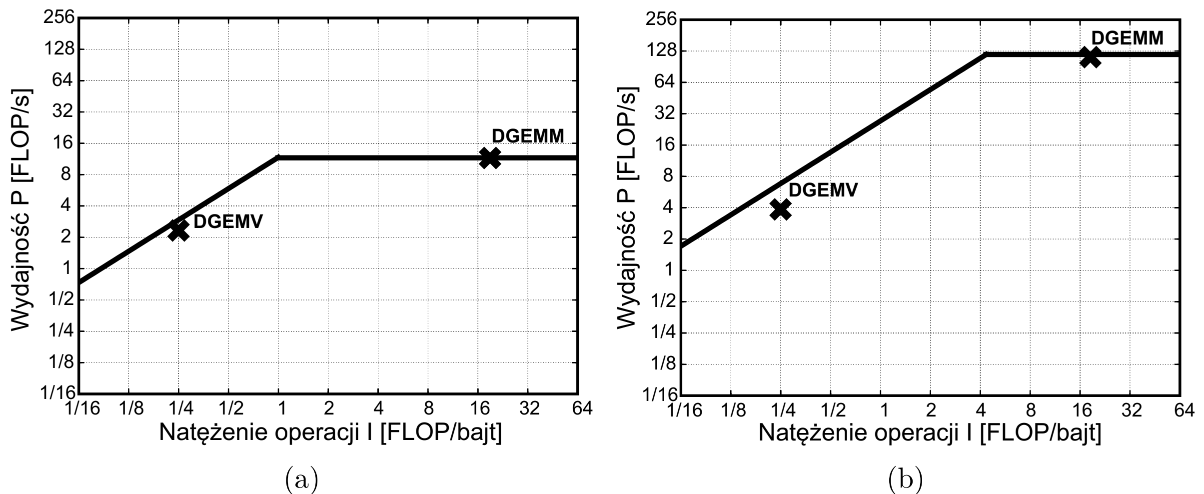
Tabela 2.1: Wydajność procesora Intel Xeon X5650.

Liczba wątków	P_{max}	B_{max}
1	11.62	11.75
12	118.15	27.32

Każda rodzina procesorów Intel posiada różne nazwy zdarzeń, dla podobnych operacji, głównie wynika to ze zmian w architekturze samych procesorów. Dla procesora Intel Xeon X5650 do wyznaczenia liczby operacji zmiennoprzecinkowych W, wykorzystano następujące zdarzenia: FP_COMP_OPS_EXE:SSE_DOUBLE_PRECISION oraz FP_COMP_OPS_EXE:SSE_FP_PACKED. Poprawność wybranych zdarzeń potwierdzono za pomocą procedury DGEMM (wykorzystano implementację z biblioteki MKL [76]), która jest algorytmem ograniczonym obliczeniowo [84]. Wynik zgadzał się z wynikiem otrzymanym w teście LINPACK.

Do wyznaczenia liczby przetransferowanych bajtów Q, wykorzystano następujące zdarzenia UNC_QMC_NORMAL_READS:ANY oraz UNC_QMC_WRITES:FULL:ANY. Poprawność wybranych zdarzeń potwierdzono za pomocą testu STREAM, gdzie porównano wynik testu z wynikiem otrzymanym za pomocą powyższych zdarzeń.

Model „Roofline” dla procesora Intel Xeon X5650 przedstawiono na rys. 2.6. Zamieszczono również punkty dla wykonania procedury DGEMM, DGEMV (implementacja z biblioteki MKL), dla rozmiaru macierzy 2800, analogiczne testy wykonano w pracy [84]. Otrzymano podobne rezultaty. Uzyskany model posłuży do przedstawienia wydajności algorytmów zaprezentowanych w niniejszej pracy.



Rysunek 2.6: Wykres „Roofline" dla procesora Intel Xeon X5650 : (a) dla 1 wątka i (b) dla 12 wątków.

2.4 Programowanie równoległe na maszyny z pamięcią wspólną

2.4.1 Wprowadzenie do OpenMP

OpenMP (ang. *Open Multi Processing*) to proste, a jednocześnie silne narzędzie do tworzenia aplikacji równoległych na maszynach z pamięcią wspólną [92, s. 163]. Jest to standard opracowany pod koniec lat 90. XX wieku przez największych producentów maszyn równoległych w USA (IBM, SGI, SUN, HP, Compaq), a następnie przejęty przez producentów oprogramowania (UNIX/LinuX, Windows, kompilatory Intela, kompilatory gcc).

OpenMP implementuje paradygmat programowania równoległego „rozdziel-łącz" (ang. fork-join), patrz rys. 2.7, na którym przedstawiono podstawy tego paradygmatu.

OpenMP składają się z trzech elementów:

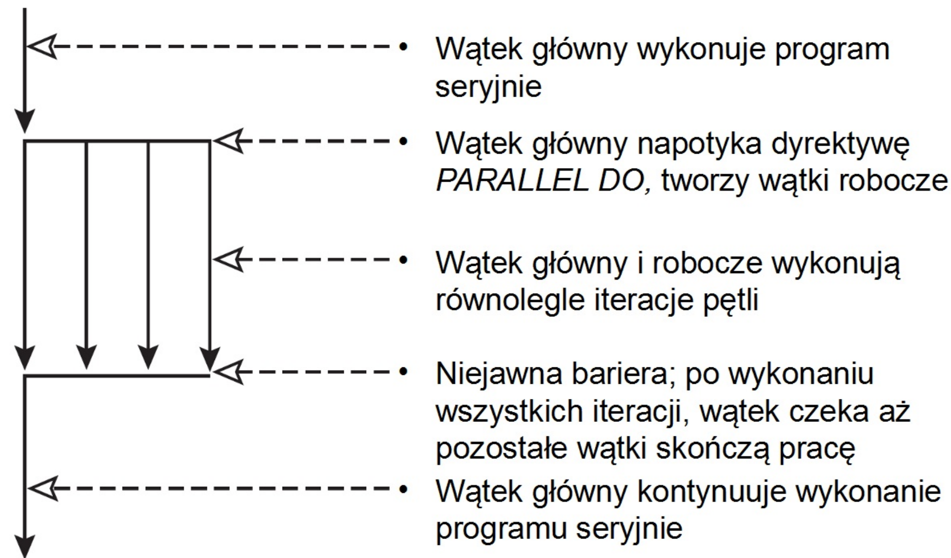
1. dyrektywy

   Instrukcje dla kompilatora, jak przeprowadzić równoległość w danym fragmencie kodu. Zaczynają się od znaku komentarza („c" lub „!" w Fortranie), co oznacza, że tylko kompilator znający dyrektywy będzie mógł je przetworzyć, a dla innych kompilatorów zostanie to uznane za komentarz. Dzięki temu można używać tego samego kodu do uruchomień kodu sekwencyjnego i równoległego patrz [16, s. 6].

2. zmienne środowiskowe

   Cztery zmienne środowiskowe służące do definiowania środowiska pracy dla OpenMP:

   - OMP_SCHEDULE - określa sposób przydziału iteracji dla dyrektywy „for".
   - OMP_DYNAMIC - określa czy środowisko OpenMP może zmieniać liczbę wątków, aby lepiej wykorzystać zasoby systemu.
   - OMP_NUM_THREADS - określa statycznie liczbę wątków



Rysunek 2.7: Równoległa pętla wg OpenMP.

- OMP_NESTED - umożliwia zagnieżdżanie wykonania równoległego. Domyślnie „false".

3. bibliotekę procedur

   Procedury biblioteczne służą do zmieniania środowiska obliczeń w trakcie działania aplikacji, identyfikacji maszyny, a także do pomiaru czasu. Są uzupełnieniem dla dyrektyw. Programy nie muszą z nich korzystać.

Kod 2.1 przedstawia prosty program z dyrektywami OpenMP. Fragment kodu między dyrektywami „PARALLEL" i „END PARALLEL" będzie wykonany równolegle na wielu wątkach.

Kod 2.1: Prosty program z dyrektywami OpenMP

```
1 program hello
2    print *, "Hello parallel world from threads:"
3 !$OMP PARALLEL
4    print *, omp_get_thread_num()
5 !$OMP END PARALLEL
6    print *, "Back to the sequential world."
7 end
```

2.4.2 Dyrektywa „PARALLEL DO"

Dyrektywa „PARALLEL DO" ułatwia tzw. paralelizację na poziomie pętli (ang. *loop-level parallelism*), tzn. iteracje pętli są wykonywane współbieżnie na różnych wątkach [16, s. 41]. Aby ją zastosować, wystarczy pętlę wyznaczoną do zrównoleglenia obudować w dyrektywę „PARALLEL DO", tak jak zostało to zrobione w Kodzie 2.2. Korzystanie

z tej dyrektywy wymusza na użytkowniku warunek, aby kolejne kroki w iteracjach były między sobą **niezależne**.

Kod 2.2: Równoległa pętla

```
1  program hello
2     integer index
3     print *, "Start␣of␣parallel␣world"
4  !$OMP PARALLEL DO
5     do index = 1, 10
6         work(index)
7     enddo
8  !$OMP END PARALLEL DO
9     print *, "Back␣to␣the␣sequential␣world."
10 end
```

Każdy z wątków wykonuje określoną liczbę iteracji. Po wykonaniu wszystkich przez dany wątek, czeka on na barierze, która jest niejawnie ustawiona na końcu bloku równoległego. Bariera zatrzymuje wykonanie głównego wątku programu do czasu, aż wszystkie wątki dotrą do bariery.

Wykonanie równoległej pętli jest kontrolowane przez opcjonalne klauzule ([16, s. 43]). W OpenMP istnieje pięć podstawowych rodzajów klauzul dla dyrektywy „PARALLEL DO":

1. Klauzule zasięgu (ang. *scoping clauses*) - klauzule:

   - *private* - zmienna jest prywatna dla każdego wątku
   - *shared* - zmienna jest współdzielona między wątkami
   - *firstprivate* - zmienna jest prywatna dla każdego wątku, ale przed wejściem do pętli zmienna jest inicjalizowana wartością z sekwencyjnego wykonania programu.
   - *lastprivate* - zmienna jest prywatna dla każdego wątku, ale zmienna po wykonaniu pętli posiada wartość taką, jaką miałaby, gdyby program był wykonywany sekwencyjnie.

2. Klauzule harmonogramu (ang. *schedule clause*) - kontrolują w jaki sposób poszczególne iteracje są przydzielane poszczególnym wątkom ([16, s. 86-87]).

   - brak klauzuli - liczba iteracji jest dzielona przez liczbę wątków i taka liczba jest przydzielana każdemu wątkowi. Może to się różnić w zależności od implementacji.
   - *static* - statycznie ustala się wielkość podziału - liczbę iteracji przydzielaną wątkowi.
   - *dynamic* - użytkownik ustala wielkość podziału (domyślnie 1), ale podziały są przydzielane wątkom dynamicznie na podstawie zużytych zasobów.

- *guided* - użytkownik ustala minimalną wielkość podziału. Podczas wykonania wielkość podziału maleje ekspotencjalnie od pewnej wartości (uzależnionej od implementacji) do minimalnej.
- *runtime* - wszystko jest ustalane podczas wykonywania programu na podstawie zmiennych środowiskowych („OMP_SCHEDULE”).

3. Klauzule wyboru (ang. *if clause*) - pętla jest wykonywana równolegle lub sekwencyjnie w zależności od zdefiniowanego przez użytkownika testu.

4. Klauzule kolejności (ang. *ordered clause*) - ustala kolejność w wykonywaniu iteracji w przypadku, gdy iteracje nie są całkowicie od siebie niezależne. Dzięki tej klauzuli możliwe jest umieszczanie regionów „ORDERED” w części leksykalnej i dynamicznej (patrz rozdz. 2.4.3 oraz rys. 2.8). Gdy wątek ma wykonać kod z regionu „ORDERED”, czeka aż zostanie wykonana iteracja z numerem o jeden mniejszym. Taki region może być tylko jeden dla danej pętli równoległej.

5. Klauzule kopiowania (ang. *copyin clause*) - kopiuje wartości zmiennych prywatnych dla wątku z wątku głównego, tak aby każdy wątek zaczynał z tymi samymi wartościami. Dotyczy tylko zmiennych oznaczonych atrybutem *„threadprivate”*.

Tabela 2.2: Porównanie rodzajów harmonogramów. N - liczba iteracji, P - liczba wątków.

Typ	Wielkość podziału	Liczba podziałów	Złożoność
none	N/P	P	najmniejsza
static	C	N/C	mała
dynamic	C	N/C	średnia
guided	N/P i zmniejsza się	$N/C <$	duża
runtime	zależy	zależy	zależy

2.4.3 Zasięg zmiennych

Wiele wątków w OpenMP wykonuje program równoległy w tej samej przestrzeni adresowej, co powoduje, że każdy wątek posiada dostęp do tych samych zmiennych. Współdzielenie zmiennych powoduje, że komunikacja między wątkami jest bardzo prosta: wątki wysyłają swoje dane podstawiając odpowiednie wartości współdzielonym zmiennym. Wątek odbiera dane czytając z takiej zmiennej ([16, s. 49]). Zmienne, które są dostępne dla wszystkich wątków nazwane są zmiennymi współdzielonymi (ang. *shared*). Zmienne dostępne tylko dla dane wątku to zmienne prywatne (ang. *private*).

Programista musi sam zdecydować, które zmienne powinny być prywatne, a które powinny być współdzielone. Domyślnie wszystkie zmienne są zmiennymi współdzielonymi. Za pomocą klauzul zasięgu można określić, czy zmienna jest prywatna czy współdzielona. Określenie to ma tylko znaczenie w zasięgu statycznym regionu równoległego.

Zasięg statyczny to miejsce, w kodzie między wywołaniami dyrektyw „PARALLEL" i „END PARALLEL". Zasięg dynamiczny to zasięg statyczny plus kod, który jest wywoływany z zasięgu statycznego - rys. 2.8. Oznacza to, że zmienne oznaczone jako prywatne w bloku *„common"* w Fortranie nie będą prywatne w zasięgu dynamicznym. Aby były one również prywatne w zasięgu dynamicznym trzeba skorzystać z dyrektywy „THREADPRIVATE". Za pomocą tej dyrektywy można oznaczyć bloki *„common"*, które będą prywatne dla wątku w całym programie. Dlatego też ta dyrektywa musi pojawić się wszędzie, gdzie wykorzystano dany blok *„common"*. Stąd warto używać polecenia *„include"* dla włączania bloku *„common"* do kodu programu.

Zmienne przekazywane do subroutine'y w zasięgu statycznym zachowują swój atrybut, tzn. jeżeli zmienna była prywatna w zasięgu statycznym i jest przekazana jako parametr do subroutine'y to w subroutine'ie pozostanie prywatna, a jeśli była współdzielona to pozostanie współdzielona. Zmienne lokalne dla subroutine'y są zmiennymi prywatnymi dla wątku. Trzeba pamiętać, że zmienne oznaczone atrybutem *„save"* w Fortranie traktowane są jako zmienne globalne i takie zmienne zawsze są współdzielone między wątkami, nawet jeżeli są wywoływane wewnątrz subroutine'y.

Kod 2.3: Przykład dla zasięgu zmiennych

```
1  subroutine caller(a, n)
2   integer n, a(n), i, j, m
3
4   m = 3
5  !$OMP PARALLEL DO
6   do i = 1, n
7     call callee(a(i), m, i)
8   enddo
9  end
10
11 subroutine callee(x, y, z)
12  common /com/ c
13  integer x, y, z, c, ii, cnt
14  save cnt
15
16  cnt = cnt + 1
17  do ii = 1, z
18   x = y + c
19  enddo
20 end
```

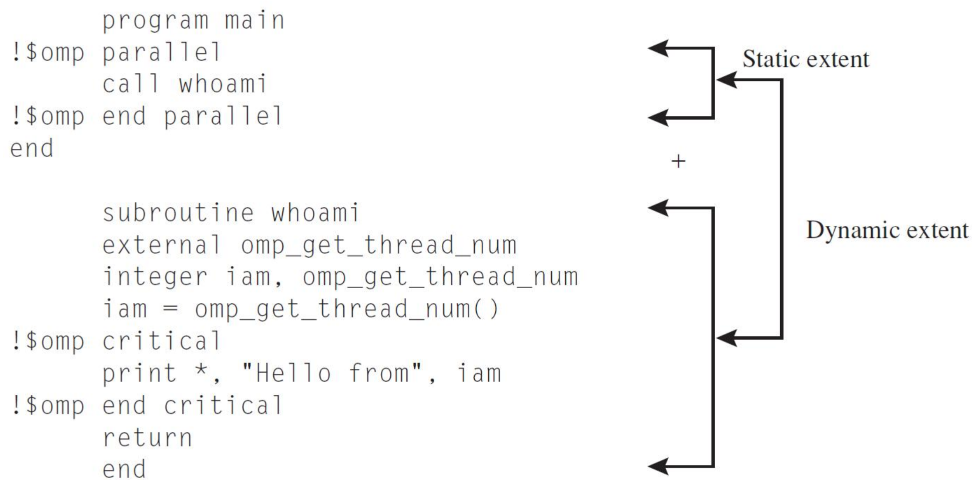
Kod 2.3 przedstawia program, dla którego w tab. 2.3 przedstawiony jest zasięg zmiennych oraz krótkie wyjaśnienie.

2.4.4 Synchronizacja

Programowanie współbieżne wiąże się z problem wzajemnego wykluczania (ang. *mutual exclusion* lub w skrócie *mutex*), czyli unikania równoczesnego użycia wspólnego zasobu.

Tabela 2.3: Zasięg zmiennych dla Kodu 2.3.

Zmienna	Zasięg	Bezpieczne użycie?	Powód takiego zasięgu
a	współdzielona	tak	zadeklarowana poza obszarem równoległym
n	współdzielona	tak	zadeklarowana poza obszarem równoległym
i	prywatna	tak	indeks równoległej pętli
j	prywatna	tak	indeks sekwencyjnej pętli wewnątrz pętli równoległej
m	współdzielona	tak	zadeklarowana poza obszarem równoległym
x	współdzielona	tak	faktycznym parametrem jest a, które jest współdzielone
y	współdzielona	tak	faktycznym parametrem jest m, które jest współdzielone
z	prywatna	tak	faktycznym parametrem jest i, które jest prywatne
c	współdzielona	tak	znajduje się w bloku *common*
ii	prywatna	tak	lokalna zmienna subroutine'y
cnt	współdzielona	nie	lokalna zmienna subroutine'y z atrybutem *save*



Rysunek 2.8: Region równoległy z wywołaniem subroutine'y. Zasięg statyczny i dynamiczny.

OpenMP dostarcza trzy podstawowe mechanizmy dla wzajemnego wykluczania.

1. sekcje krytyczne - podstawowa forma tej sekcji została przedstawiona w Kodzie 2.4. OpenMP zapewnia, że dostęp do tej sekcji będzie miał tylko i wyłącznie jeden wątek w danym momencie. Sekcje krytyczne dzielą się na nazwane i nienazwane. Sekcja nienazwana (bez atrybutu „name") powoduje wzajemne wykluczenie dla wszystkich sekcji krytycznych znajdujących się w programie, tzn. tylko jeden wątek może znajdować się w sekcji krytycznej nienazwanej. Można to porównać do sytuacji, w której sekcję krytyczną nienazwaną traktuje się jako jedną dużą sekcję krytyczną. Sekcje krytyczne nazwane to takie, które posiadają nazwę (atrybut „name"). W danym momencie może być tylko jeden wątek w sekcji krytycznej o tej samej nazwie, ale to nie powoduje, że inny wątek nie może trafić do innej sekcji krytycznej nazwanej (o innej nazwie).

Zagnieżdżenie sekcji krytycznych należy wykonywać z wielką rozwagą, ponieważ OpenMP nie zawiera żadnych mechanizmów zabezpieczających przed zakleszczeniami (ang. *„deadlock”*).

2. sekcja atomowa - została przedstawiona w Kodzie 2.5. Działa jak sekcja krytyczna, ale odnosi się tylko do następnej linii kodu. Warto jej używać ze względów wydajnościowych - jest szybsza i wykorzystuje sprzętowe wspomaganie takich operacji. Jej działanie jest bardzo ograniczone; x musi być skalarem o typie wbudowanym w język Fortran, *operator* to jeden z predefiniowanych operatorów (większość arytmetycznych i logicznych), *intrinsic* to wewnętrzne funkcje Fortrana (np. *min, max* oraz logiczne funkcje), *expr* to wyrażenie skalarne, które nie zawiera w sobie x. Poniższa lista zwiera akceptowane operatory dla atomowej sekcji krytycznej: +, *, –, /, .AND., .OR., .EQV., .NEQV., MAX, MIN, IAND, IOR, IEOR.

3. zamki uruchamiane dynamicznie - biblioteka OpenMP wprowadza również funkcje, które tworzą zamki w sposób dynamiczny. Funkcje te zostały przedstawione w tab. 2.4. Jest to dodatkowe narzędzie dla wzajemnego wykluczania, które wprowadza o wiele większą dowolność użycia. Przede wszystkim nie muszą znajdować w tej samej subroutine'ie, z tego powodu to na programiście spoczywa odpowiedzialność za włączanie i wyłączanie odpowiednich zamków.

Kod 2.4: Sekcja krytyczna w OpenMP

```
1 !$omp critical [(name)]
2  block
3 !$omp end critical [(name)]
```

Kod 2.5: Sekcja atomowa

```
1 !$omp atomic
2  x = x operator expr
3  ...
4 !$omp atomic
5  x = intrinsic (x, expr)
```

Tabela 2.4: Funkcje uruchamiające zamki dynamiczne.

Nazwa funkcji	Opis
omp_init_lock(var)	Tworzy i inicjalizuje zamek
omp_destroy_lock(var)	Niszczy i uwalnia zamek
omp_set_lock(var)	Zajmuje zamek jeśli jest wolny,
	w innym przypadku czeka aż zamek będzie wolny
omp_unset_lock(var)	Zwalnia zamek,
	uruchamia wątki (jeśli są), które czekają na zamku
logical omp_test_lock(var)	Próbuje zająć zamek,
	zwraca prawdę jeśli się udało inaczej fałsz

2.5 Programowanie równoległe na maszyny z pamięcią rozproszoną

2.5.1 Wprowadzenie

Obliczenia rozproszone to dziedzina informatyki, w której rozpatruje się systemy rozproszone, tzn. takie systemy, które posiadają połączenie między komponentami za pomocą sieci komputerowej, a komunikują się i koordynują swoje działania za pomocą przesyłania komunikatów. Mogą one być również realizowane w zintegrowanych komputerach z pamięcią rozproszoną [17]. Systemy, w których nie ma jednego wspólnego systemu adresowania, ale każda jednostka posiada swój system adresowania, nazywa się systemami z pamięcią lokalną (patrz rozdz. 2.1). Tworzenie aplikacji w takich środowiskach jest znacznie trudniejsze niż programowanie na maszynach wielordzeniowych z pamięcią wspólną. Przede wszystkim dlatego, że konieczne jest jawne przekazywanie danych między nadawcą i odbiorcą. Do przekazywania danych używa się komunikatów. Obecnie obliczenia rozproszone prowadzi się głównie w języku C lub Fortran, wzbogaconych o funkcje umożliwiające współpracę równolegle wykonywanych procesów za pomocą przesyłania komunikatów.

Istnieje wiele bibliotek dających takie możliwości (np. *PVM - Parallel Virtual Machine*), ale najpopularniejsza w tym momencie jest biblioteka MPI (ang. *Message Passing Interface*). MPI jest standardem interfejsu do przesyłania komunikatów w rzeczywistych i wirtualnych maszynach równoległych z pamięcią lokalną.

Historia powstania biblioteki MPI sięga wczesnych lat dziewięćdziesiątych ubiegłego wieku. W kwietniu 1992r. odbyły się warsztaty dotyczące standardów przesyłania wiadomości w środowiskach z pamięcią rozproszoną (Workshop on Standards for Message Passing in a Distributed Memory Environment, Williamsburg, Virginia), które sponsorowane były przez Center for Research on Parallel Computing (CRPC) Uniwersytetu Rice'a. W warsztatach uczestniczyło ponad 80 osób z około 40 organizacji zainteresowanych rozwojem obliczeń rozproszonych, m.in. przedstawiciele producentów komputerów równoległych (IBM, Intel, Cray), naukowcy z uniwersytetów (amerykańskich Rice, CIT, Yale, ale również europejskich Wiedeński, Bari), laboratoriów (NASA, Argonne). W trakcie spotkania przedstawiono wstępne założenia do standardu oraz wytypowano

grupę roboczą, której zadaniem było opracowanie standardu. Grupa przedstawiła swoją propozycję w listopadzie 1992r., która była dyskutowana i ulepszana do maja 1994r., kiedy to została opublikowana wersja MPI-1 standardu. Grupa robocza w dalszym ciągu rozwijała standard, w szczególności definiując możliwości równoległego wejścia/wyjścia, dynamicznego tworzenia zadań oraz powiązań z językami Fortran90 oraz C++. Wynikiem tych prac była wersja MPI-2 przyjęta w lipcu 1997r. oraz wersja MPI-2.2 przyjęta we wrześniu 2009 roku [17]. Obecnie najnowszą wersją standardu jest wersja MPI-3.1 opublikowana w czerwcu 2015. Włączyła ona do standardu nieblokujące funkcje kolektywne, dodała nowe jednostronne operacje komunikacyjne, a także wsparła połączenie z językiem Fortran2008 [80].

Implementacja biblioteki MPI wchodzi w skład oprogramowania praktycznie wszystkich komercyjnych komputerów równoległych, również tych ze wspólną pamięcią. Wolne od opłat, przenośne wersje biblioteki dostępne są w Internecie, m.in. wersje mpich oraz mpich2 (www.mpich.org) opracowane w Argonne National Laboratory, wersja Open MPI (www.open-mpi.org), która została opracowana przez konsorcjum jednostek akademickich, badawczych i przemysłowych [17]. Z ostatniej z wymienionych wersji korzysta autor niniejszej rozprawy.

2.5.2 Programowanie równoległe z MPI

Podstawowym pojęciem MPI jest komunikator, który może być rozumiany jako grupa procesów plus pewien kontekst [92, s. 199]. Kontekst to pewne dodatkowe informacje o procesach, np. topologia sieci. Komunikat wysłany z danym komunikatorem, może być odebrany tylko przy użyciu tego samego komunikatora. Komunikatory są identyfikowane przy pomocy uchwytu o typie MPI_Comm. Po uruchomieniu zawsze powstaje komunikator o uchwycie MPI_COMM_WORLD, który zawiera wszystkie procesy mogące się komunikować. W MPI istnieje również pojęcie grupy procesów, która jest zbiorem procesów, ale niemającego kontekstu. W grupie każdy proces ma swój indeks, który należy do zakresu od zera do liczby procesów pomniejszonej o jeden. Grupy są identyfikowane poprzez uchwyty o typie MPI_Group.

W MPI komunikacja między procesami może być zrealizowana na kilka różnych sposobów. Standard MPI rozróżnia sposoby komunikacji ze względu na:

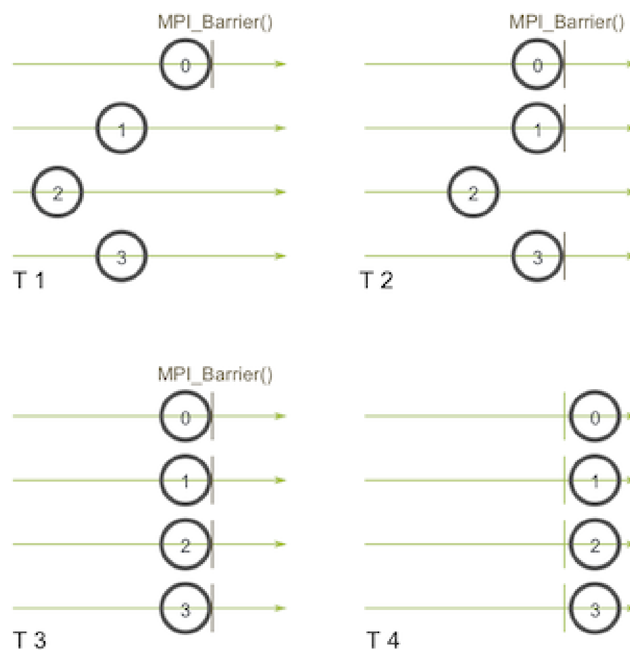
1. synchronizację wzajemną:
   (a) komunikacja synchroniczna - nadawca czeka na gotowość odbiorcy i kończy wysyłanie komunikatu, gdy odbiorca potwierdza jego odbiór - MPI_Ssend, MPI_Srecv.
   (b) komunikacja asynchroniczna - nadawca wysyła komunikat i nie czeka na potwierdzenie odbioru komunikatu przez odbiorcę. Odbiorca pobiera go w dowolnej, odpowiedniej dla niego chwili - MPI_Send, MPI_Recv.
2. warunek powrotu z funkcji wysłania lub odebrania:
   (a) komunikacja blokująca - powrót z funkcji następuje po zakończeniu operacji - MPI_Send, MPI_Recv.

(b) komunikacja nieblokująca - powrót z funkcji następuje natychmiast - MPI_Isend, MPI_Irecv.

Standard MPI umożliwia różne kombinacje trybów wysłania i odbierania komunikatów, np.: nieblokujące, synchroniczne wysyłanie a blokujące odbieranie, albo blokujące, asynchroniczne wysyłanie a nieblokujące odbieranie. To programista ma obowiązek wybrania odpowiedniego sposobu w zależności do sytuacji.

Powyższe sposoby komunikacji odnoszą się do przesyłania komunikacji między jednym procesem a drugim. Jeden proces był odbiorcą i drugi proces był nadawcą. Standard MPI definiuje również komunikację kolektywną służącą do komunikacji między więcej niż dwoma procesami. Komunikacja kolektywna zachodzi między jednym wyróżnionym procesem grupy a pozostałymi członkami grupy. Do głównych funkcji służących do komunikacji kolektywnej należą (rysunki zaczerpnięto z [81]):

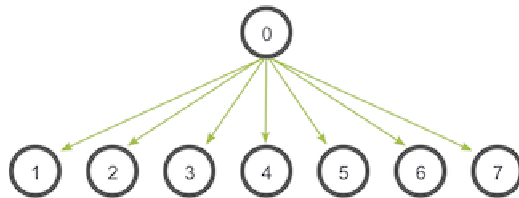
1. MPI_Barrier - służy do synchronizacji za pomocą bariery. Każdy proces, który ją wywoła czeka, aż wszystkie procesy z danego komunikatora ją wywołają - patrz rys. 2.9.



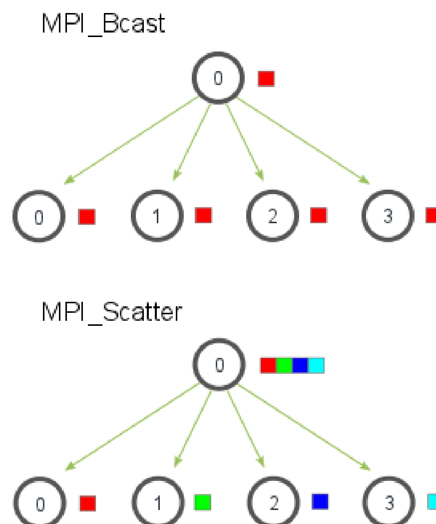
Rysunek 2.9: Zasada działania funkcji MPI_Barrier.

2. MPI_Bcast - służy do wysyłania komunikatu do grupy procesów. Funkcja wysyła dane z procesu lidera, a w pozostałych odbiera te dane - rys. 2.10.

3. MPI_Scatter - służy do rozsyłania danych między członków komunikatora. Lider dzieli dane na równe części (tyle ile jest członków komunikatora) i przekazuje każdą porcję innemu procesowi. Różnicę między MPI_Bcast a MPI_Scatter pokazano na rys. 2.11.

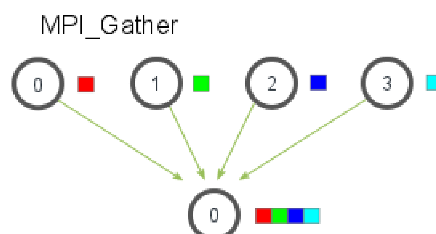


Rysunek 2.10: Zasada działania funkcji MPI_Bcast.

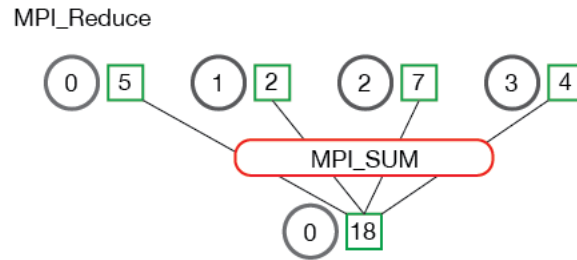


Rysunek 2.11: Porównanie zasady działania funkcji MPI_Bcast i MPI_Scatter.

4. MPI_Gather - służy do zbierania danych od członków komunikatora. Każdy proces łącznie z liderem przekazuje liderowi porcję danych - rys. 2.12.
5. MPI_Reduce - służy do zbierania danych od członków komunikatora, wykonania na tych danych operacji redukcji (jest wiele predefiniowanych operacji, jak suma, produkt, element maksymalny/minimalny, ale można również definiować własne) i przekazania wyniku operacji redukcji do lidera - rys. 2.13.



Rysunek 2.12: Zasada działania funkcji MPI_Gather.



Rysunek 2.13: Zasada działania funkcji MPI_Reduce, operacją redukcji jest sumowanie (MPI_SUM) .

2.6 Podstawowe problemy programowania równoległego

2.6.1 Zakleszczenie

Zakleszczenie lub blokada (ang. *deadlock*) to sytuacja, w której co najmniej dwa różne wątki (procesy) czekają na siebie nawzajem, na skutek czego żaden nie może kontynuować pracy. Najprostsze zakleszczenie powstaje gdy każdy z wątków utrzymuje w swojej wyłącznej dyspozycji pewien zasób i jednocześnie czeka na zwolnienie innego zasobu zajętego przez drugi z wątków. Druga możliwość wystąpienia jest gdy dwa lub większa liczba wątków czeka na zwolnienie zasobu w łańcuchu cyklicznym.

Do zakleszczenia dojdzie, jeśli spełnione będą cztery warunki:

1. Wzajemne wykluczenie - w danym czasie tylko jedno zadanie może z niego korzystać; w ogólności warunkiem do zakleszczenia jest też sytuacja w której do zasobu jest możliwy jednoczesny równoległy dostęp wielu zadań, lecz liczba jednocześnie zadanych żądań do zasobu jest większa od liczby maksymalnych równoległych dostępów do zasobu, które mogą zostać obsłużone;

2. Trzymanie zasobu i oczekiwanie - zadanie utrzymuje jeden z zasobów, ale do ukończenia pracy niezbędne jest także zaalokowanie zasobów innego typu;

3. Cykliczne oczekiwanie - zadania w taki sposób żądają zasobów, że powstaje cykliczny graf skierowany;

4. Brak wywłaszczania z zasobu - zadania dobrowolnie nie rezygnują z przydzielonych im zasobów; zwolnienie zasobów możliwe jest po zakończeniu zadania.

Aby unikać zakleszczeń potrzebne są odpowiednie mechanizmy; mechanizmy ze standardu OpenMP zostały opisane w rozdz. 2.4.4.

2.6.2 Fałszywe współdzielenie

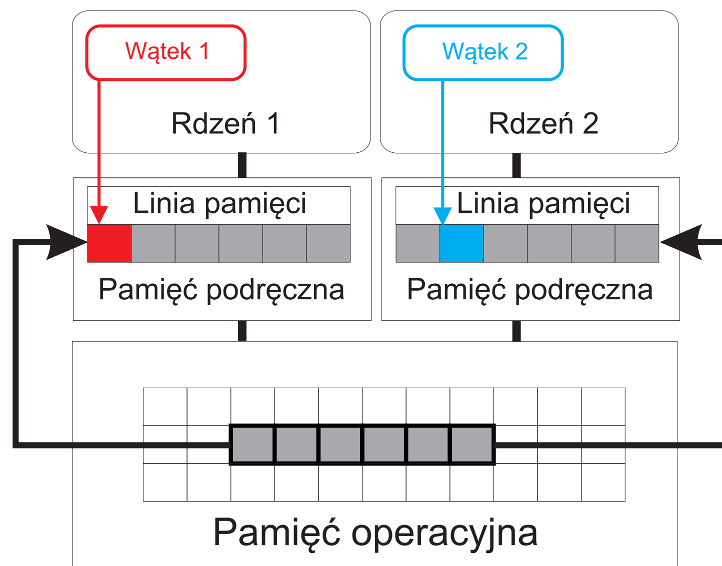
Fałszywe współdzielenie (ang. *false sharing*) to problem wydajnościowy na maszynach wielordzeniowych, w których każdy rdzeń posiada lokalną pamięć podręczną. Problem ten występuje, kiedy wątki z różnych rdzeni próbują modyfikować zmienne, które występują w tej samej linii pamięci podręcznej, jak na rys. 2.14. Taka sytuacja jest nazywana fałszywym współdzieleniem, ponieważ mimo że użytkownikowi wydaje się, że wątki

współdzielą dostęp do pewnej zmiennej, to przez to, że dane zmienne rezydują w tej samej linii pamięci podręcznej, procesor musi wykonać dodatkowe operacje synchronizujące dostęp do obu zmiennych.

W Kodzie 2.6 istnieje możliwość fałszywego współdzielenie tablicy sum_local. Ta tablica ma rozmiar równy liczbie wątków i jej wielkość jest wystarczająco mała aby zmieścić się w pojedynczej linii pamięci podręcznej. Gdy kod zostanie uruchomiony równolegle, każdy wątek zmodyfikuje inny, ale przyległy element tablicy, a to spowoduje, że linia pamięci podręcznej na której występuje tablica sum_local zostanie unieważniona dla wszystkich rdzeni. To z kolei powoduje, że każdy rdzeń otrzyma komunikat o niekonsystencji danych między pamięcią podręczną, a pamięcią główną i powstanie potrzeba przesłania pamięci podręcznej do pamięci głównej, nazywa się to pomyłką pamięci podręcznej (ang. *cache miss*). W efekcie kod, który miał być wykonany równolegle zostanie wykonany sekwencyjne.

Kod 2.6: Fałszywe współdzielenie

```
1   double precision sum, sum_local(NUM_THREADS)
2  !$omp parallel num_threads(NUM_THREADS)
3           me = omp_get_thread_num()
4           sum_local(me) = 0.0
5  !$omp for
6           do i = 1 , N
7            sum_local(me) += x(i) * y(i)
8       enddo
9  !$omp atomic
10          sum = sum + sum_local(me)
11 !$omp end parallel
```



Rysunek 2.14: Problem fałszywego współdzielenia.

2.6.3 Jednoczesna wielowątkowość

Jednoczesna wielowątkowość (SMT - ang. *simultaneous multithreading)* to technika służąca do podniesienia wydajności procesorów z wieloma rdzeniami. W normalnym przebiegu każdy rdzeń wykonuje jeden potok instrukcji w danym momencie. W jednoczesnej wielowątkowości rdzeń może wykonywać więcej niż jeden potok instrukcji. Dzięki temu gdy rdzeń wykonuje instrukcję, która powoduje, że musi czekać (np. na dostęp do innego urządzenia), może przełączyć się na drugi potok, co poprawia ogólną wydajność procesora. Zostało to zaimplementowane np. w procesorach Intel pod komercyjną nazwą *HyperThreading*. W rozdz. 4.2.2 zostanie pokazane, że w przypadku obliczeń numerycznych korzystanie z tej funkcjonalności pogarsza wydajność kodu.

2.6.4 Koligacja procesu/wątku

Koligacja procesu lub wątku (ang. *processor/thread affinity*) pozwala na łączenie procesu lub wątku z procesorem (rdzeniem), w taki sposób że danych wątek będzie mógł się wykonywać tylko na danym procesorze (rdzeniu).

Poprawnie stworzona koligacja procesu korzysta z faktu, że pewne dane, które proces poprzednio używał mogą pozostać w pamięci procesora (np. w pamięci podręcznej), dzięki czemu można zminimalizować efekt pomyłek pamięci podręcznej (ang. *cache miss*).

W przypadku OpenMP do koligacji wątków można wykorzystać zmienną środowiskową GOMP_CPU_AFFINITY, a dla kompilatora Intel można również skorzystać z interfejsu KMP_AFFINITY.

Koligacja wątków ma istotny wpływ na wykonanie programu, co zostanie pokazane w rozdz. 4.2.3.

3 Zrównoleglenie pętli po elementach w FEAPie

W tym rozdziale omówione zostanie zrównoleglenie pętli po elementach używając OpenMP w programie FEAP [25]. Program ten to kod MES, bardzo popularny w ośrodkach akademickich. Nawet dla sekwencyjnej wersji FEAPa (istnieje również wersja na klaster), taka paralelizacja jest zadaniem nietrywialnym ze względu na istniejącą architekturę kodu, która bardzo utrudnia efektywną paralelizację.

Najpierw zostanie porównana wersja seryjna FEAPa z równoległą wersją programu Warp3D [110]. Porównany zostanie czas wykonania i zapotrzebowanie pamięć operacyjną. Zostanie pokazane, że Warp3D jest szybszy, ale potrzebuje więcej pamięci niż FEAP. Analiza kodu Warp3D pomoże przy tworzeniu własnej metody paralelizacji pętli po elementach.

Następnie przedstawione zostaną zmiany dokonane w kodzie FEAPa, które były niezbędne do zrównoleglenia pętli po elementach używając OpenMP. W szczególności procedura tworzenia globalnej macierzy z macierzy elementowych została zidentyfikowana jako kluczowa dla uzyskania dobrej wydajności. Poza tym użyto kilku różnych dyrektyw synchronizacyjnych dla wzajemnego wykluczania z OpenMP, które również zostały przetestowane.

Na koniec tego rozdziału pokazano wydajność zrównoleglonego FEAPa, oznaczonego jako ompFEAP na numerycznych przykładach z elementami 3D i elementami powłokowymi, przy czym przetestowano elementy wbudowane FEAPa i elementy użytkownika (tzw. *user elements*). Z przeprowadzonych badań i testów można wyciągnąć wniosek, że używając dyrektywy *ATOMIC* do synchronizacji łączenia (redukcji) macierzy elementowych, można osiągnąć bardzo dobre przyspieszenie i efektywność.

3.1 Opis problemu

Motywacją do paralelizacji pętli po elementach w kodach MES jest fakt, że w wielu zaawansowanych zastosowaniach inżynieryjnych, czas obliczenia macierzy stycznej ma duży udział w całkowitym czasie obliczeń.

Dla przykładu w obliczeniach procesu formowania metalu, w którym używa się praw konstytutywnych uwzględniających teksturę, czas tworzenia macierzy stycznej jest dłuższy niż czas rozwiązywania układu równań liniowych, o czym wspomina praca [79].

Dla metod bezelementowych Galerkina (ang. *element-free method*) stosowanych do obliczeń elektromagnetycznych w [51], czas generowania macierzy stycznej to ok. 40% całkowitego czasu, 50% czasu to czas rozwiązywania układu równań liniowych, a 10% to czas na operacje wejścia/wyjścia i pozostałe operacje.

Również wielowarstwowe kompozytowe modele MES powłok z mikrostrukturą, które są w kręgu zainteresowań tej pracy, mają podobną charakterystykę. Aby móc używać zaawansowanych trójwymiarowych praw konstytutywnych potrzebne jest podejście bezpośrednie, w którym elementy trójwymiarowe są użyte w najbardziej wytężonych częściach powłoki. Są to obszary gdzie najprawdopodobniej wystąpi zniszczenie materiału, w pozostałych obszarach modelu użyte są elementy powłokowe typu „solid-shells”,

wykorzystujące także trójwymiarowe równania konstytutywne, podobne podejście było stosowane w [33], [53].

To podejście prowadzi do dużych, obliczeniowo kosztownych modeli MES, które wymagają zrównoleglonego kodu MES oraz użycia klastrów, przy czym najbardziej efektywne jest podejście hybrydowe .

Zrównoleglenie kodu MES wykonuje się zazwyczaj poprzez zastosowanie dwóch technik:

1. użycie równoległych solwerów, takich jak np. PARDISO, MUMPS lub PaStiX,
2. zrównoleglenie pętli po elementach, stosując OpenMP lub pthread.

W tym rozdziale opisano jak w efektywny sposób zrealizować punkt 2, wykorzystując OpenMP.

Od wprowadzenia standardu OpenMP w 1997, cieszył się on zainteresowaniem społeczności tworzącej oprogramowanie inżynieryjne. Znalazł zastosowanie w szerokiej gamie problemów, z ostatnich lat np.: jawny kod MES dla symulacji zderzeń [85], rozwiązywanie równań Navier-Stoke'sa [107], kalibracja modelu przepływu wód gruntowych [105], rozwiązywaniu układu równań liniowych [26], całkowanie po objętości w problemach elektromagnetycznych [97], triangularyzacja Delaunay'a [70] oraz optymalizacja topologii w ograniczeniami naprężeniowymi [88]. W pracach tych uzyskano przyspieszenie rzędu 3.5x dla 4 wątków, 10.6x dla 12 wątków i 11x dla 16 wątków, co udowadnia, że faktycznie OpenMP jest użytecznym narzędziem.

Najważniejszym problemem przy zrównoleglaniu pętli po elementach jest łączenie lokalnych macierzy w globalną macierz styczną, w niektórych pracach operacja ta jest określana jako „redukcja". Podczas tego procesu występuje tak zwany „wyścig po zasoby" (ang. *race condition*), tzn. dwa wątki mogą nadpisać równocześnie tę samą lokalizację w pamięci, co prowadzi do błędnych rezultatów. W literaturze przedstawiono kilka metod uniknięcia tego problemu, ale żadne nie prowadzi do skalowalnego kodu.

W pracy [8], zostało zaprezentowanych kilka sposobów, a wśród nich podejście z użyciem dyrektywy *ATOMIC* z OpenMP i jako usprawnienie metoda bazująca na podejściu wykluczonego posiadania (ang. *exclusive ownership*). W tym podejściu dyrektywa *ATOMIC* jest wyłączana, gdy węzeł z modelu MES należy tylko do jednego elementu. Dzięki tym metodom osiągnięto przyspieszenie odpowiednio 9x i 15x na 32 rdzeniach.

W [109], redukcja była przeprowadzona używając sekcji krytycznych z OpenMP, przeprowadzono testy dla adaptacyjnej metody elementów skończonych. Aby ulepszyć efektywność paralelizacji, użyto hierarchiczne dodawanie macierzy, w którym zawsze dwa wątki mogą dodawać swoje macierze. Uzyskano przyspieszenie 5.2x dla 8 rdzeni, ale autorzy tej pracy uważają, że technika ta powinna dawać lepsze rezultaty wraz ze wzrostem liczby rdzeni na procesorze.

W pracy [39], użyto tzw. zachłanny algorytm (ang. *greedy algorithm*) kolorowania grafu do zidentyfikowania niezależnych zbiorów elementów, które mogą być przetwarzane jednocześnie. W tej metodzie dwa elementy o wspólnym kolorze nie współdzielą żadnego stopnia swobody. Osiągnięto przyspieszenie 8x dla 12 wątków i 19x dla 32 wątków na maszynie 32-rdzeniowej (2x16 rdzeni).

W przypadku GPU, proces redukcji spotyka dodatkową trudność, jaką jest gospodarowanie pamięcią (globalną, lokalną i współdzieloną). W [12], zostały porównane cztery strategie: (a) kolorowanie elementów,(b) uzupełnianie tylko niezerowych części macierzy używając pamięci globalnej, (c) uzupełnianie tylko niezerowych części macierzy używając pamięci współdzielonej, i (d) uzupełnianie tylko niezerowych części macierzy używając pamięci lokalnej.

W pracy [73] pokazano, że dla iteracyjnych metod rozwiązywania równań liniowych, podejście z lokalnymi macierzami jest bardziej efektywne niż metoda kolorowania grafów. Przyczyną tego jest fakt, że metody iteracyjne nie wykorzystują macierzy, ale iloczyn skalarny macierzy i wektora, co znacznie przyspiesza wykonanie kodu.

W niniejszym rozdziale omówiona zostanie implementacja równoległej pętli po elementach w sekwencyjnym kodzie FEAP [25] za pomocą OpenMP. Główna trudność to rozbudowana architektura istniejącego kodu, dlatego kod zostanie zrównoleglony używając standardowych dyrektyw OpenMP.

W rozdz. 3.2, została porównana wersja seryjna programu FEAP z równoległym kodem Warp3D [110]. Porównano wykorzystanie pamięci i czasu. Dzięki temu, że Warp3D to kod typu *open-source*, można było przeanalizować jego strukturę, co pomogło w zaprojektowaniu własnej metody paralelizacji pętli po elementach, innej niż w Warp3D.

W rozdz. 3.3 opisano zmiany jakie wprowadzono w FEAPie, które były niezbędne do zrównoleglenia pętli po elementach z użyciem OpenMP. W szczególności opisano kilka różnych metod redukcji macierzy elementowych do macierzy globalnej, które wykorzystują inne metody wzajemnego wykluczania z OpenMP.

A w rozdz. 3.4 równoległa wersja FEAPa, która została oznaczona jako „ompFEAP", została przetestowana i pokazano: (1) poprawność wyników obliczeń równoległych (są identyczne jak w sekwencyjnym wykonaniu) i (2) skalowalność obliczeń (pętli po elementach). Wydajność równoległego kodu FEAP jest pokazana na numerycznych przykładach z elementami trójwymiarowymi i powłokowymi; zarówno wbudowanymi w FEAPa, jak i tzw. elementami użytkownika (własnymi). Dodatkowo zaprezentowano wydajność ompFEAPa oraz Warp3D w modelu „Roofline".

3.2 Porównanie sekwencyjnego kodu FEAP z równoległym kodem Warp3D

W tym podrozdziale porównano sekwencyjną wersję programu FEAP z równoległym programem Warp3D, w szczególności wykorzystanie pamięci i czas wykonania. Następnie przeanalizowano metodę zrównoleglenia w Warp3D, co dało podstawy do użycia innej metody w FEAPie.

Krótka charakterystyka powyższych dwóch kodów:

1. FEAP [25] to kod akademicki używany w wielu ośrodkach na świecie, rozwijany przez prof. R.L. Taylora [119]. Istnieje wersja sekwencyjna i wersja na klaster, która używa bibliotek PETSc, ale pętla po elementach w żadnej z nich nie została zrównoleglona na maszyny z pamięcią wspólną.

Ten rozdział dotyczy paralelizacji pętli po elementach tylko w wersji sekwencyjnej. Wykorzystano FEAP w wersji 8.4, do którego podłączono zrównoleglony na maszyny z pamięcią wspólną solwer PARDISO z biblioteki MKL firmy Intel [76] w wersji 11.1.0.

2. Warp3D [110] to wolne oprogramowanie MES z udostępnionym kodem źródłowym rozwijane przez grupę prof. R. Doddsa. Może być skompilowany do następujących wersji: (1) z wątkami OpenMP [83], i (2) hybrydowej, w której wykorzystano również procesy MPI [80]. Do porównań została użyta wersja z wątkami OpenMP. Wykorzystano Warp3D w wersji 17.5.3. Parametry dla solwera PARDISO ustawiono dokładnie tak samo jak w FEAP.

Oba programy były skompilowane używając dwóch różnych kompilatorów dla języka Fortran: Intel Compiler 14.0.0 oraz GCC 4.8.2 z flagą optymalizacyjną 2 (w celu uzyskania najszybszego kodu wynikowego). Różnica w wydajności między oboma kompilatorami jest bardzo mała (por. tab. 3.1), dlatego w dalszych rozważaniach, przedstawiono tylko wyniki dla kompilatora firmy Intel. Wykorzystano najnowszy standard OpenMP w wersji 4.0. Ponadto kod został zweryfikowany narzędziem Intel Inspector 2015 [52]; narzędzie to nie raportowało żadnych błędów w pętli równoległej. Dodatkowo wykorzystano Thread Affinity Interface dla kompilatorów Intela, aby zminimalizować walkę o pasmo pamięci (ang. *memory bandwidth*) i do pamięci podręcznej (ang. *cache memory*) procesora. Dokładniej, zastosowano technikę rozproszenia przydziałów (każdy wątek na innym rdzeniu), co prowadzi do zwiększenia łącznego pasma pamięci oraz zwiększa pamięć podręczną dostępną wyłącznie dla danego wątku.

Implementację oraz testy zostały przeprowadzone na klastrze GRAFEN [29]. Wykorzystano jeden węzeł tego klastra, który zawiera 2 procesory Xeon X5650 2.66 GHz z 6 rdzeniami i 24GB DDR3 1333MHz pamięci.

Tabela 3.1: Test kostki. Porównanie czasów dla różnych kompilatorów. $N = 48$

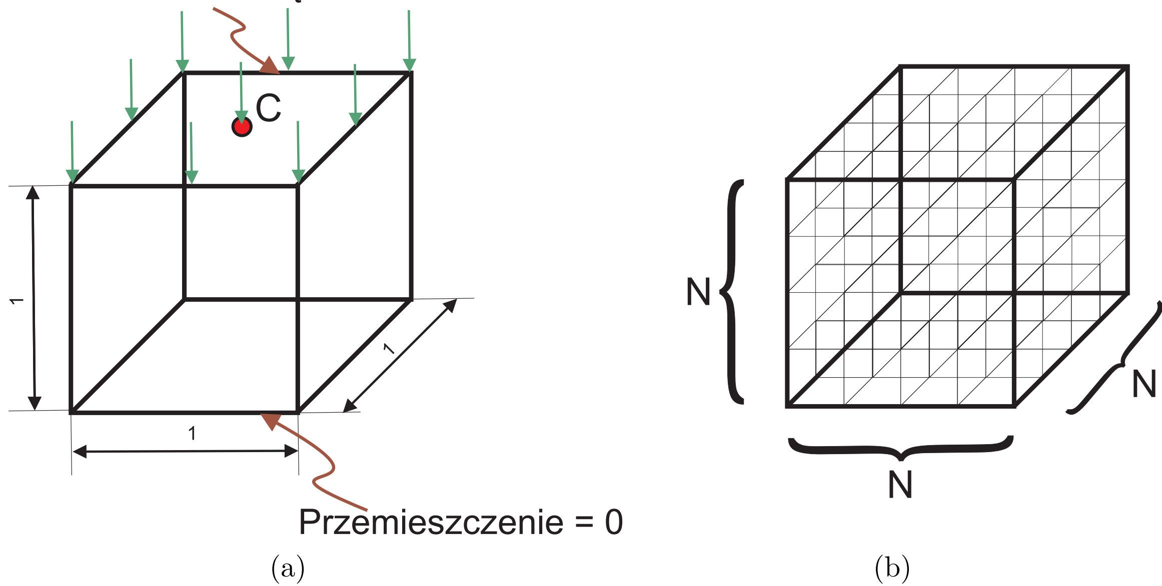
Liczba wątków	Czas [s]		Przyspieszenie	
	Intel	GCC	Intel	GCC
1	5.18	5.02		
12	0.47	0.47	11.02	10.68

3.2.1 Test kostki

W tym podrozdziale została porównana wydajność dwóch kodów na liniowym problemie trójwymiarowym z mechaniki.

Kostka wykonana z jednego materiału zostaje poddana równomiernemu obciążeniu od góry oraz utwierdzona u podstawy, patrz rys. 3.1a. Dane materiałowe są następujące: moduł Young'a $E = 207914.0$, współczynnik Poisson'a $\nu = 0.28342$. Kostka jest podzielona na $N \times N \times N$ elementów trójwymiarowych, patrz rys. 3.1b, i w każdym węźle na górze kostki została przyłożona siła $P = 10^6/(N+1)^2$. Wartość N może być zmieniana. W tym teście wykorzystano własny element trójwymiarowy 8-węzłowy, który został zaimplementowany w FEAPie jako element użytkownika (ang. *user element*). Opis

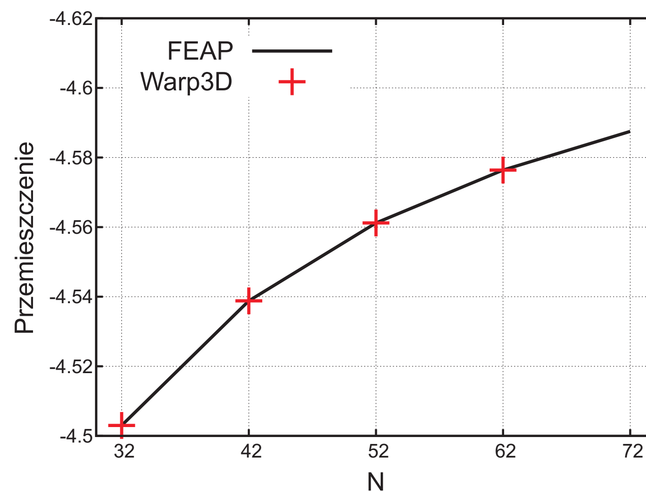
tego elementu znajduje się w dodatku D. W Warp3D wybrano standardowy trójwymiarowy element 8-węzłowy. Maksymalna liczba wątków została ustalona na 12.



(a) (b)

Rysunek 3.1: Test kostki. (a) Geometria początkowa i obciążenie. (b) Siatka MES.

Powyższy problem liniowy rozwiązano używając obu programów, przemieszczenia w punkcie centralnym na górze kostki o współrzędnych $[0.5, 0.5, 1.0]$ zaprezentowano na rys. 3.2. Oba programy dają dokładnie takie same rezultaty, ale wersja sekwencyjna FEAPa pozwalała uruchomić program również dla $N = 72$; dla tego przypadku Warp3D przekraczał dostępną pamięć (24GB). W tab. 3.2 podano liczbę równań, elementów i węzłów dla poszczególnych wartości N.



Rysunek 3.2: Test kostki. Przemieszczenia w węźle C.

Tabela 3.2: Liczba równań, elementów i węzłów dla poszczególnych wartości N.

N	Równania	Elementy	Węzły
30	86 490	27 000	29 791
32	104 544	32 768	35 937
38	173 394	54 872	59 319
42	232 974	74 088	79 507
48	345 774	110 592	117 649
52	438 204	140 608	140 608
54	490 050	157 464	166 375
60	669 780	216 000	226 981
62	738 234	238 328	250 047
64	811 200	262 144	274 625
68	971 244	314 432	328 509
72	1 151 064	373 248	389 017

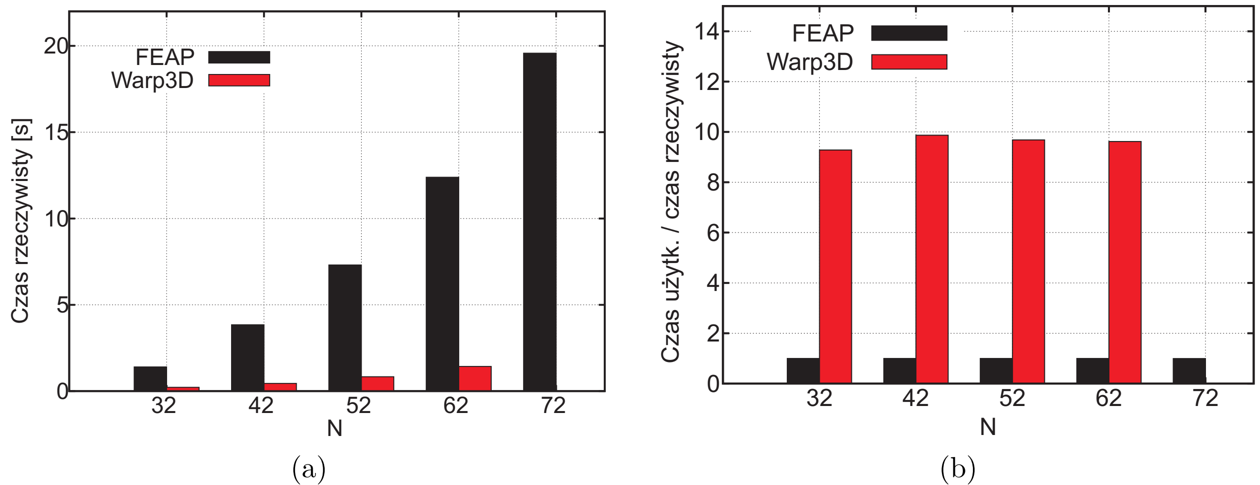
Następnie zostały porównane czasy wykonania dla obu programów wykorzystując standardową komendę systemu Unix - „time”, która zwraca dwie wartości: (1) czas rzeczywisty (ang. *real time*), to jest czas między rozpoczęciem a zakończeniem zadania, zwany również czasem zegarowym (ang. *wall-clock time*) oraz (2) czas użytkownika (ang. *user time*), to jest sumę czasów, które wykorzystały rdzenie procesora na wykonanie kodu programu. Przedstawiono tylko czas w pętli po elementach, natomiast czas w PARDISO i czas w sekwencyjnej części programu zostały wyłączone z dalszych rozważań.

1. Czas rzeczywisty to czas, który jest najistotniejszy dla użytkownika, został przedstawiony na rys. 3.3a. Widać, że wersja równoległa Warp3D jest zdecydowanie szybsza niż sekwencyjny FEAP, np. dla $N = 62$ jest 8 razy szybsza.

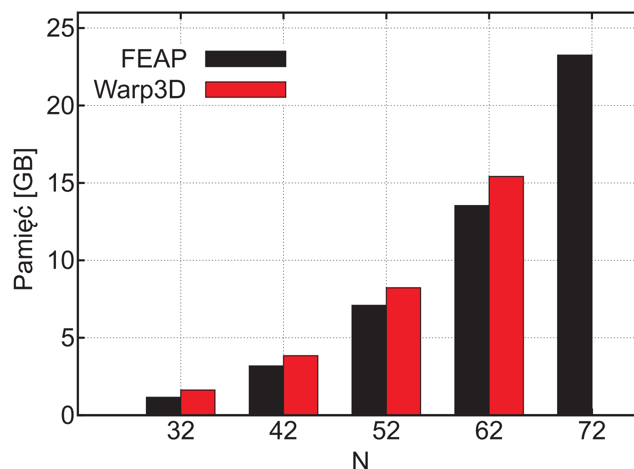
2. Użycie wątków scharakteryzowano jako iloraz czasu użytkownika przez czas rzeczywisty i przedstawione na rys. 3.3b. Widać, że Warp3D używa wątków w o wiele większym stopniu niż sekwencyjny FEAP, dla $N = 62$ dziewięciokrotnie.

Dobre czasy wykonania równoległego Warp3D stanowiły dodatkową motywację do stworzenia wersji FEAPa ze zrównolegloną pętlą po elementach.

Jednak można zauważyć, że Warp3D używa więcej pamięci niż FEAP, patrz rys. 3.4, i dlatego przypadek $N = 72$ nie mógł być przeliczony przez ten program. Rysunek ten uwzględnia również pamięć, która została użyta przez PARDISO, który został uruchomiony z identycznymi parametrami w obu programach. Aby wyjaśnić różnicę w użyciu pamięci, sprawdzono kod źródłowy Warp3D, a wyniki zostały przedstawione w następnym podrozdziale.



Rysunek 3.3: Test kostki. Porównanie: (a) czasu rzeczywistego, i (b) ilorazu czasu użytkownika przez czas rzeczywisty, dla pętli po elementach.



Rysunek 3.4: Test kostki. Porównanie użycia pamięci (wraz z PARDISO).

3.2.2 Metoda sparalelizowania pętli w Warp3D

Aby wybrać optymalny sposób sparalelizowania FEAPa przeanalizowano kod równoległej pętli po elementach w Warp3D (Warp3D używa OpenMP [83], który został opisany w rozdz. 2.4).

W Warp3D zastosowano interesujący sposób zrównoleglenia pętli po elementach. Została użyta standardowa dyrektywa PARALLEL DO, ale pętla została podzielona na dwie części:

1. obliczenie elementowych macierzy sztywności element po elemencie, co jest standardem w kodach MES. Różnica polega na tym, że Warp3D przechowuje wszystkie macierze elementowe w pamięci, aby je później użyć do tworzenia macierzy globalnej. Dlatego Warp3D potrzebuje więcej pamięci niż FEAP, jak pokazano na rys. 3.4.

2. redukcja elementowych macierzy sztywności do globalnej macierzy nie jest prze-

prowadzona element po elemencie ale wiersz po wierszu, gdzie wiersz to pojedynczy wiersz w macierzy globalnej. Wygląda to tak: jeden wiersz jest wybierany i znajdowane są wszystkie elementy, które powinny dodać swoje wartości do tego wiersza. Na końcu właściwy wiersz macierzy elementowej jest dodawany do wiersza macierzy globalnej. W ten sposób, gdy zostanie wykonane to równolegle, nie ma problemu synchronizacji i walki o zasoby przez wątki.

Ta metoda jest bardzo podobna do metody opisanej w [8] i nazwaną tam „array expansion", w której także proces redukcji jest rozłożony na dwie oddzielne pętle. Funkcjonalność pierwszej jest bardzo podobna, ale druga jest inna, ponieważ redukcja jest przeprowadzona sekwencyjnie w [8]. Druga pętla jest również omówiona dla GPU w [12], jako „redukcja po wierszach", gdzie jeden wątek jest odpowiedzialny za jeden wiersz układu równań, patrz str. 11 tamże.

3.3 Równoległa pętla po elementach w FEAPie

Zaimplementowana przez autora niniejszej pracy pętla po elementach w sekwencyjnym programie FEAP także używa OpenMP, ale w inny sposób niż Warp3D. Elementowa macierz lokalna i wektor nie są dodatkowo przechowywane jak w Warp3D ale redukowane (agregowane) natychmiast po tym jak zostaną obliczone. Inaczej potraktowano również proces redukcji.

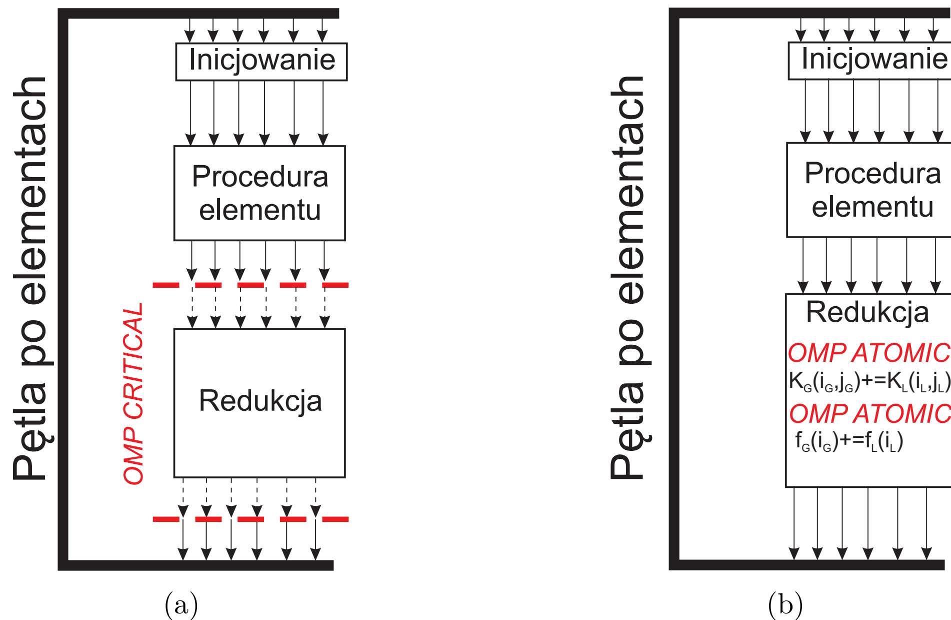
Pętle zrównoleglono za pomocą dyrektywy PARALLEL DO w procedurze „pform" w następujący sposób:

Kod 3.1: 'PARALLEL DO' w 'pform.f'

```
1 c      Loop over active elements
2 c$OMP PARALLEL DO NUM_THREADS( nthreads ),
3 c$OMP& PRIVATE( vvv ),
4 c$OMP& LASTPRIVATE( s ),
5 c$OMP& COPYIN( ccc )
6        do n = nn1,nn2,nn3
7        ...
8        end do ! n
9 c$OMP END PARALLEL DO
```

gdzie *vvv* to lista zmiennych, a *ccc* to lista bloków „common". Opis dyrektyw PRIVATE i COPYIN znajduje się w rozdz. 2.4.3. Klauzula LASTPRIVATE, zawiera tylko macierz styczną „s" i jest potrzebna aby komenda „EIGE" z programu FEAP wykonała się poprawnie.

Pętla po elementach w sekwencyjnym programie FEAP jest zorganizowana w następujący sposób: (1) lokalizacja macierzy elementowych, (2) obliczenia lokalnych macierzy stycznych i wektora residualnego, (3) redukcja lokalnej macierzy elementowej i wektora do globalnej macierzy i wektora. Redukcja ma największy wpływ na wydajność równoległego kodu, dlatego zaimplementowano i przetestowano wszystkie metody synchronizacji bazujące na wzajemnym wykluczaniu (ang. *mutual exclusion*), które są dostępne w



Rysunek 3.5: Dwa schematy różnych podejść do redukcji macierzy elementowych. (a) ompFEAPcs używa OMP CRITICAL i (b) ompFEAP używa OMP ATOMIC.

OpenMP. Najbardziej charakterystyczne dwie zostały przedstawione na rys. 3.5 i opisane poniżej.

1. Para dyrektyw *CRITICAL/END CRITICAL* została użyta w wersji programu oznaczonej jako ompFEAPcs, gdzie przedrostek „cs" oznacza „critical section". W procedurze „pform", procedura redukcji „passble" jest otoczona przez te dyrektywy w następujący sposób:

   Kod 3.2: Sekcja krytyczna w 'pform.f' dla ompFEAPcs

   ```
   c                    Assemble arrays as necessary
   c$OMP CRITICAL
                        call passble( ... )
   c$OMP END CRITICAL
   ```

   Jest to najprostszy sposób zastosowania tej dyrektywy, ale jak zostanie to pokazane w testach numerycznych, zapewnia on lepszą wydajność niż wiele małych sekcji krytycznych (nazwanych) zaimplementowanych prawie w tych samych miejscach co dyrektywa *ATOMIC* (opisana poniżej), z tym, że otaczających najbliższą pętlę.

2. Dyrektywy *ATOMIC* są użyte w wersji programu oznaczonej jako ompFEAP. Dyrektywę tę użyto dokładnie przed liniami, w których wykonuje się redukcję, co prowadzi do wielu sekcji krytycznych, ale bardzo krótkich. Wykonanie dyrektywy *ATOMIC* jest wspierane sprzętowo.

   W [16, str. 152] wytłumaczono, że dyrektywa ta zapewnia wyłączny dostęp do danej jednostki pamięci podczas wykonania aktualizacji wartości w tej jednostce.

Dodatkowo synchronizacja jest zintegrowana z aktualizacją wspólnych jednostek pamięci, dzięki temu nie jest potrzebne tworzenie dodatkowych zamków, czy sekcji krytycznych, co prowadzi do wyższej wydajności.

Na przykład, dyrektywa *ATOMIC* została użyta w procedurze „cassem" (podobnie w procedurze „dasble") dla redukcji rzadkiej w następujący sposób:

Kod 3.3: Sekcja krytyczna w 'cassem.f' dla ompFEAP

```
c               Assemble including diagonal
                if(diagin) then
                  if( ip(k).ge.ip(n) ) then
                    inz = inza(jc(n+neq),jc(n+neq+1)-1,ir,k,n)
c$OMP ATOMIC
                    ad(inz) = ad(inz) + s(j,i)
                    if(alfl) then
c$OMP ATOMIC
                      al(inz) = al(inz) + s(i,j)
                    endif
                  endif
                  ...
                endif
```

Należy zaznaczyć, że OpenMP zawiera również inne metody synchronizacji, jak np. procedury opisane w rozdz. 2.4.4. W tym rozdziale wykorzystano procedury *OMP_SET_LOCK* i *OMP_UNSET_LOCK*, które wstawiono w prawie tych samych miejscach co dyrektywę *ATOMIC*, z tym że otaczają one najbliższą pętlę. Wszystkie te metody zostaną przetestowane w rozdz. 3.4.1.

Poza tym, aby osiągnąć poprawnie działającą wersję równoległego ompFEAP trzeba było rozwiązać kilka innych problemów:

1. Podstawowym problemem był wybór czy zmienna powinna być prywatna czy współdzielona, tzn. czy powinna być wstawiona do listy „vvv" w klauzuli PRIVATE dla dyrektywy OMP PARALLEL DO. Jeśli za dużo zmiennych jest prywatnych wtedy w czasie inicjalizacji, podczas której OpenMP kopiuje dodatkowe zmienne dla każdego wątku, wykonana jest dodatkowa i niepotrzebna praca.

2. FEAP przekazuje wiele zmiennych do procedur poprzez bloki „common", więc trzeba zdecydować, czy dany blok „common" powinien być prywatny dla każdego wątku. Jeśli tak, to wtedy trzeba dodać dyrektywę *THREADPRIVATE* po definicji danego bloku w pliku „include" („*.h"). Na przykład zmodyfikowany plik „eldata.h" wygląda następująco:

Kod 3.4: Zmodyfikowany plik 'eldata.h' dla ompFEAP

```
1       real*8          dm
2       integer            n,ma,mct,iel,nel,pstyp,eltyp
3       common /eldata/ dm,n,ma,mct,iel,nel,pstyp,eltyp
4 c$omp THREADPRIVATE(/eldata/)
```

Celem jest oczywiście posiadanie minimalnego zestawu prywatnych parametrów, aby zapewnić minimalny czas inicjalizacji dla każdego wątku. Warto zauważyć, że jeśli blok „common" będzie oznaczony jako *THREADPRIVATE*, wtedy powinien on się znaleźć na liście „ccc" w klauzuli COPYIN z OMP PARALLEL DO, patrz Kod 3.1, aby zapewnić, że wszystkie wątki posiadają zainicjowaną wartość z wątku głównego.

3. Trzeba również zapewnić, by procedury wywoływane w równoległej części kodu były bezpieczne wątkowo (ang. *thread-safe*). Jest to pojęcie stosowane w kontekście programów wielowątkowych. Mówi się, że procedura jest bezpieczna wątkowo, jeśli ta procedura manipuluje wspólnymi strukturami danych w taki sposób, że gwarantuje ona bezpieczne wykonanie przez wiele wątków w tym samym czasie. Atrybut *save* z Fortrana nie jest bezpieczny wątkowo w OpenMP, dlatego trzeba było usunąć ten atrybut z kodu. OpenMP traktuje zmienne z atrybutem *save* jako zmienne globalne, które nie mogą być prywatne dla wątku, dlatego klauzule zasięgu, takie jak *PRIVATE*, pracują w inny sposób od oczekiwanego.

Przeprowadzono wiele testów zgodności, aby zapewnić, że powyższe zmiany nie wpłynęły negatywnie na wartości (wyniki) obliczane przez ompFEAPa, które powinny być identyczne z tymi obliczonymi przez standardowy sekwencyjny program FEAP. Testy te przedstawiono w rozdz. 3.4.

3.4 Testy numeryczne ompFEAPa

W tym podrozdziale przedstawione zostaną testy ompFEAPa, które pokazują, że równoległa pętla po elementach została zaimplementowana poprawnie. Badane są dwa aspekty: (1) poprawność równoległego rozwiązania, które powinno być identyczne z rozwiązaniem sekwencyjnym oraz (2) skalowalność obliczeń w pętli po elementach.

3.4.1 Test kostki

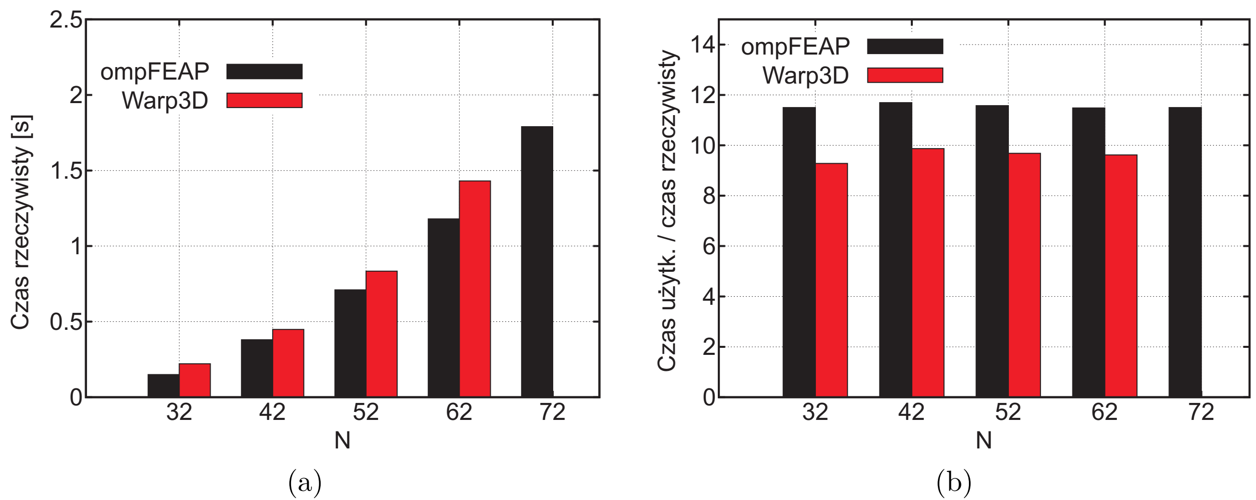
Najpierw zostaną przedstawione wyniki dla testu kostki opisanego w rozdz. 3.2.1 z użyciem równoległego ompFEAPa i standardowego elementu trójwymiarowego 8-węzłowego z FEAPa. Dodatkowo porównano wyniki z tymi uzyskanymi z autorskiej implementacji 8-węzłowego elementu trójwymiarowego, która została zaimplementowana jako „user element", aby pokazać, że również ta funkcjonalność w równoległym kodzie działa poprawnie.

Najpierw porównano czas i pamięć użytą przez ompFEAP i Warp3D.

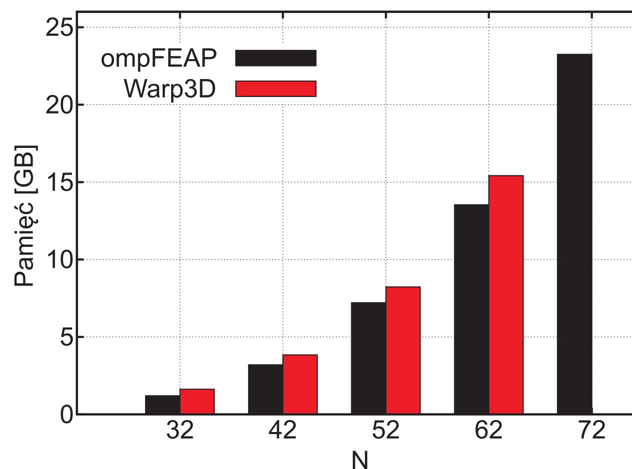
1. Czas rzeczywisty przedstawiono na rys. 3.6a i widać, że ompFEAP jest szybszy niż Warp3D o ok. 1.2 razy dla $N = 62$. Iloraz czasu użytkownika przez czas rzeczywisty pokazano na rys. 3.6b, i widać również, że ompFEAP używa wątków nieznacznie lepiej, tj. ~ 1.2 razy dla $N = 62$. W obu wypadkach monitorowany jest tylko czas dla pętli po elementach, tzn. czas w PARDISO nie jest uwzględniony.

2. Użycie pamięci przedstawiono na rys. 3.7 i widać, że Warp3D używa więcej pamięci niż ompFEAP, tj. 1.14 razy więcej dla $N = 62$. Dodatkowo zauważono, że ompFEAP używa prawie tyle samo pamięci co sekwencyjny FEAP.

Rysunki 3.6a, 3.6b i 3.7 mogą być porównane odpowiednio z rys. 3.3a, 3.3b i 3.4, gdzie przedstawiono wyniki dla sekwencyjnego programu FEAP.

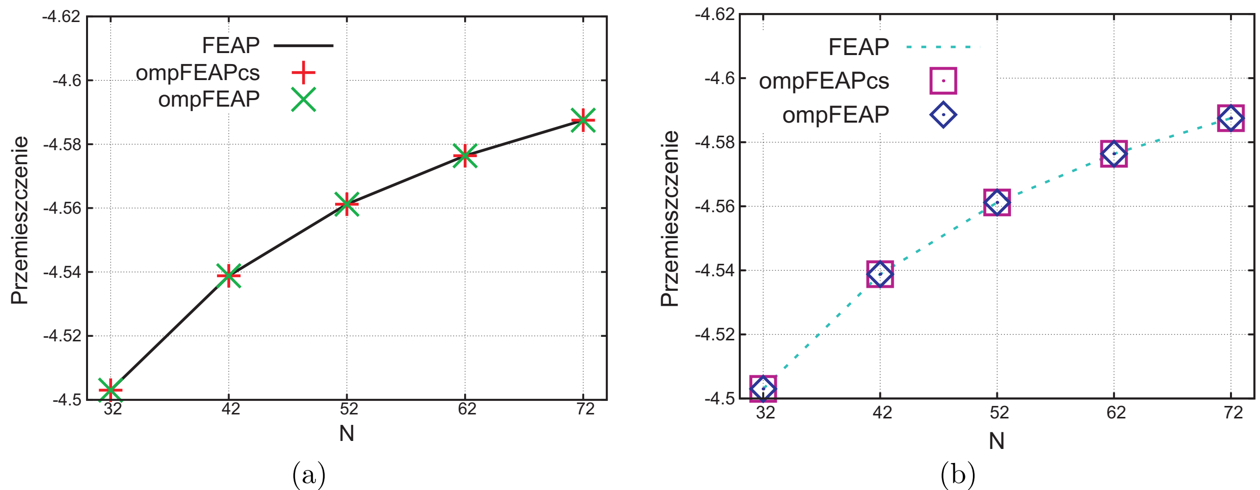


Rysunek 3.6: Test kostki. Porównanie czasu rzeczywistego i ilorazu czasu użytkownika przez czas rzeczywisty dla pętli po elementach.



Rysunek 3.7: Test kostki. Porównanie wykorzystania pamięci (wraz z PARDISO).

Poprawność. W tym teście badano czy wersja równoległa daje identyczne wyniki jak wersja sekwencyjna. Zostały przetestowane dwie wersje równoległe: (1) ompFEAPcs, która używa dyrektywy *CRITICAL* i (2) ompFEAP, która używa dyrektywy *ATOMIC*. Na rys. 3.8 przedstawiono przemieszczenia w węźle C i widać wyraźnie, że wszystkie wersje programu dają dokładnie te same wyniki dla wszystkich przetestowanych elementów, co dowodzi, że równoległa pętla została zaimplementowana poprawnie.



Rysunek 3.8: Test kostki. Przemieszczenia w węźle C. (a) 8-węzłowe trójwymiarowe elementy z FEAPa, i (b) własne 8-węzłowe elementy trójwymiarowe.

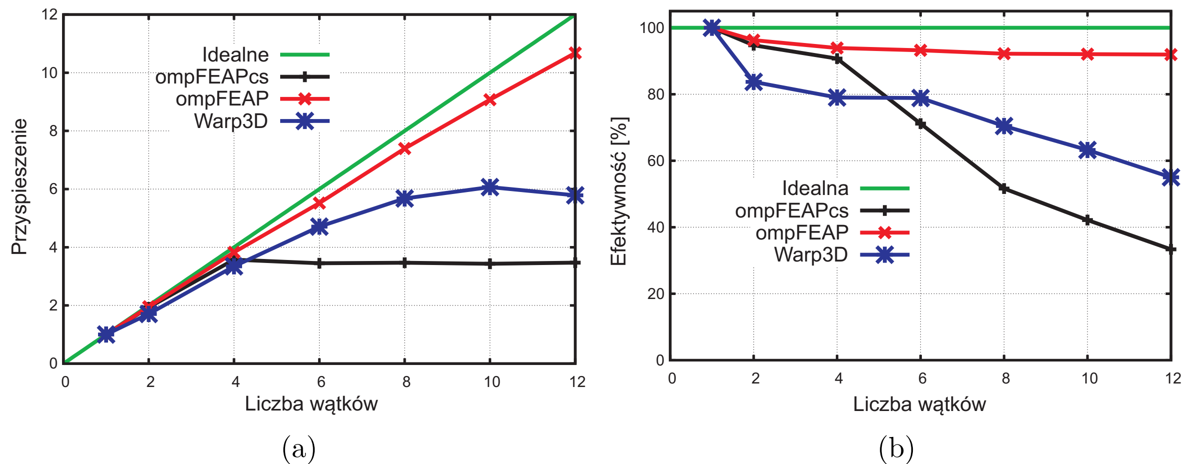
Skalowalność. Skalowalność została zmierzona używając dwóch miar zrównoleglenia - przyspieszenia i efektywności.

1. Do wyliczenia przyspieszenia używa się stałego rozmiaru zadania i sprawdzono jak czas rozwiązania zmienia się wraz ze zmianą liczby użytych wątków. Przyspieszenie to iloraz czasu dla jednego wątku przez zaalokowaną liczbę wątków, idealna sytuacja jest wtedy gdy przyspieszenie rośnie liniowo wraz z liczbą wątków.

2. Do wyliczenia efektywności używa się zadania o zmiennym rozmiarze. Rozmiar wzrasta liniowo wraz z liczbą wątków i wyliczany jest iloraz czasu wykonania dla jednego wątku przez czas wykonania dla wybranej liczby wątków. W ten sposób można sprawdzić jak czas rozwiązania zmienia się wraz ze zmianą liczby wątków, gdy każdy wątek jest równomiernie obciążony. Jest to tak zwana słaba efektywność (ang. *week efficiency*), patrz rozdz. 2.2.1 oraz [89]. W sytuacji idealnej, czas wykonania nie zmienia się wraz ze zmianą liczby wątków i rozmiarem zadania.

Badany jest tylko czas dla pętli po elementach a układ równań nie jest rozwiązywany. Przyspieszenie jest mierzone dla $N = 64$, a efektywność dla $N = 30, 38, 48, 54, 60, 64, 68$ (liczbę równań, elementów i węzłów dla wybranych wartości N przedstawiono w tab. 3.2). Wyniki testów przedstawiono na rys. 3.9a i 3.9b.

Dla ompFEAPcs, tzn. wersji używającej dyrektywy *CRITICAL*, patrz rozdz. 3.3, widać, że przyspieszenie jest poniżej 4. Kod w sekcji krytycznej działa sekwencyjne i testy



Rysunek 3.9: Test kostki. Skalowalność - przyspieszenie (a) i efektywność (b).

wykazały, że w sekwencyjnej wersji FEAPa, czas redukcji to ~ 0.24 całego czasu wykonania dla użytego elementu skończonego. W tym przypadku przyspieszenie ok. 4 potwierdza prawo Amdahla (patrz rozdz. 2.2.1) które mówi, że przyspieszenie jest ograniczone przez odwrotność współczynnika sekwencyjnej części kodu.

Dla ompFEAP, tzn. wersji z dyrektywą *ATOMIC* przyspieszenie wynosi ~ 10.7, i jest to widoczna poprawa w stosunku do poprzedniej wersji. Warto zauważyć, że przyspieszenie dla Warp3D wynosi ~ 6.2.

Podobne efekty można zaobserwować dla efektywności, patrz rys. 3.9b. Dla 12 wątków, ompFEAP pokazuje efektywność $\sim 95\%$, ompFEAPcs $\sim 35\%$, a Warp3D $\sim 55\%$. Instrukcja użytkownika dla Warp3D poleca ustawić liczbę wątków na 2-4, i w tym zakresie program ten faktycznie skaluje się liniowo, patrz [110, str. 86, 110].

Porównania wszystkich metod synchronizacji dla procesu redukcji, opisanych w rozdz. 3.3, zostały wykonane dla $N = 64$ z użyciem 12 wątków. Przeskalowane czasy oraz przyspieszenie przedstawiono na dwa sposoby: (A) tylko redukcja (wstawianie macierzy lokalnych do macierzy globalnej) w tab. 3.3, (B) budowa macierzy lokalnych i redukcja razem w tab. 3.4. Warto zauważyć, że rezultaty dla A pozostaną takie same bez względu na użyty rodzaj elementu, podczas gdy rezultaty dla B będą wykazywać coraz lepsze przyspieszenie, gdy zostaną użyte bardziej skomplikowane i czasowo złożone elementy.

Widać, że najlepszy czas i przyspieszenie daje dyrektywa *ATOMIC*, następne są procedury *LOCK*, podczas gdy implementacje z dyrektywą *CRITICAL* są gorsze. Dlatego w dalszych testach wykorzystano tylko wersję kodu z dyrektywą *ATOMIC*, oznaczoną jako ompFEAP.

Tabela 3.3: Test kostki. Porównanie różnych metod synchronizacji dla procesu redukcji. (A) Tylko redukcja. Wyniki dla $N = 64$ i 12 wątków.

Metoda	Przyspieszenie	Przeskalowany czas
ATOMIC	9.21	1.00
LOCK	5.20	1.77
jedna CRITICAL	1.19	7.75
wiele CRITICAL	0.70	13.07

Tabela 3.4: Test kostki. Porównanie różnych metod synchronizacji dla procesu redukcji. (B) Budowa macierzy lokalnych i redukcja. Wyniki dla $N = 64$ i 12 wątków.

Metoda	Przyspieszenie	Przeskalowany czas
ATOMIC	10.68	1.00
LOCK	9.12	1.24
jedna CRITICAL	3.47	3.10
wiele CRITICAL	2.22	4.76

3.4.2 Test nieliniowej powłoki

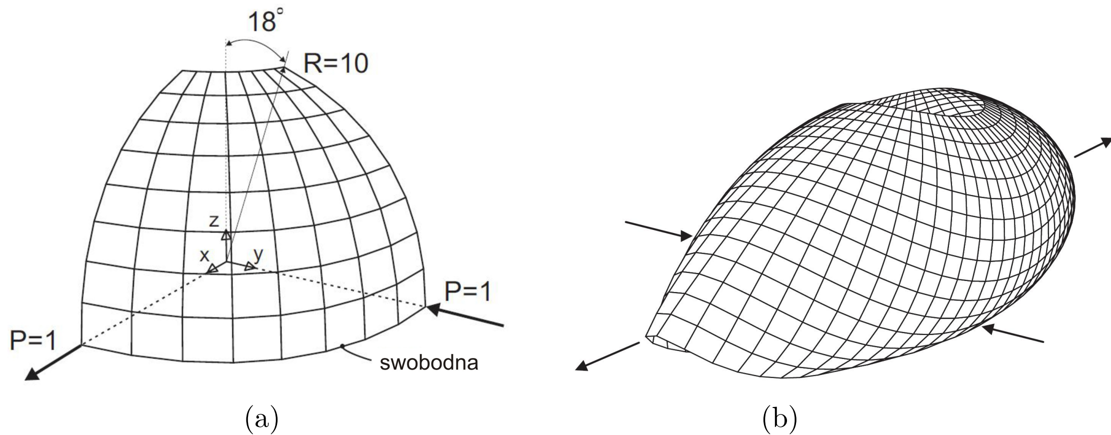
Tym nieliniowym testem pokazano, że zarówno elementy standardowe z FEAPa i elementy użytkownika działają w wersji ompFEAP. Zostały użyte dwa nieliniowe powłokowe elementy skończone:

1. z rotacyjnymi stopniami swobody. Jest to element z FEAPa z 5 stopniami swobody w węźle, i

2. typu „solid-shell”, tzn. tylko z translacyjnymi stopniami swobody zaimplementowany jako element użytkownika.

Element powłokowy z FEAPa [100] jest oparty na funkcjonale Hellingera-Reissnera, posiada 2 rotacyjne stopnie swobody w węźle i 14 wewnętrznych parametrów. (warto zaznaczyć, że to jest inny element niż ten z 6 stopniami swobody w węźle, również dostępny w FEAPie i opisany w [103]). Element HW43 to element własny bazujący na funkcjonale Hu-Washizu, który posiada tylko translacyjne stopnie swobody i ma 43 wewnętrzne parametry; więcej informacji o elementach powłokowych typu HW można znaleźć w [114] i [115].

Wykorzystano wersję programu ompFEAP, która używa dyrektyw *ATOMIC* i obliczono przykład półsfery z otworem. Półsfera jest obciążona przez dwie pary równych ale przeciwnie skierowanych sił zewnętrznych, które przyłożone są w płaszczyźnie $z = 0$ wzdłuż osi X i Y, więc jest poddawana silnemu skręcaniu i zginaniu, patrz [114, str. 445]. Dane materiałowe: $E = 6.825 \times 10^7, \nu = 0.3, h = 0.04$ lub $h = 0.4$. Geometria ćwiartki sfery i obciążenie przedstawiono na rys. 3.10a. Zdeformowaną konfigurację dla $P = 335$ pokazano na rys. 3.10b.

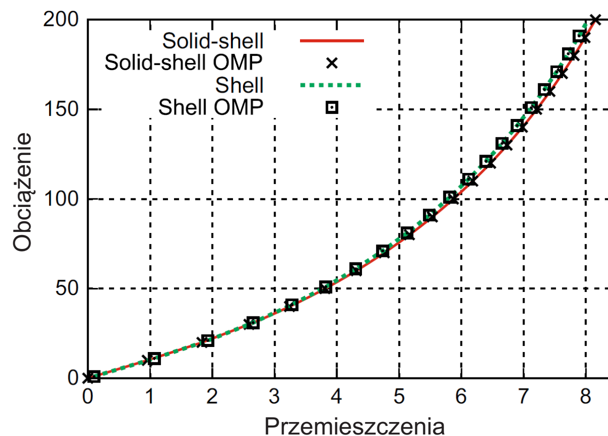


Rysunek 3.10: Półsfera obciążona dwoma parami sił. (a) Początkowa geometria i obciążenie (b) Zdeformowana konfiguracja dla $P = 335$.

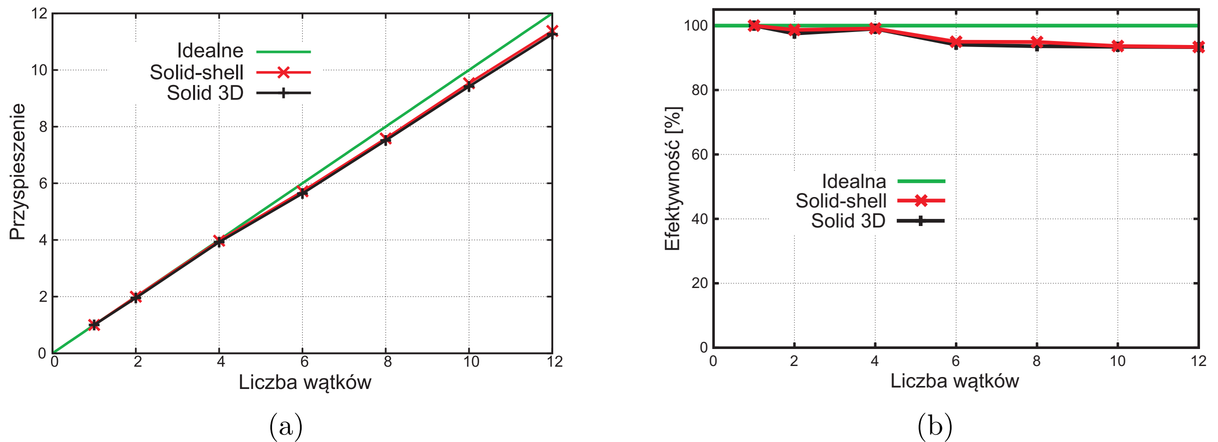
Poprawność. Grubość powłoki $h = 0.04$. Wykorzystano siatkę 16×16 elementów (z jednym elementem wzdłuż grubości) a do rozwiązania nieliniowych równań równowagi zastosowano metodę Newtona. Wyniki analizy nieliniowej pokazano na rys. 3.11. Widać, że obie wersje programu FEAP, sekwencyjna i zrównoleglona OMP, dają dokładnie takie same wyniki.

Skalowalność. Grubość powłoki $h = 0.4$ i użyto 10 elementów po grubości. Przyspieszenie jest mierzone dla siatki $316 \times 316 \times 10$, a efektywność jest mierzona dla siatek $N \times N \times 10$, gdzie $N = 91, 129, 181, 223, 258, 288, 316$. Maksymalnie wykorzystano ponad 3.3 miliona niewiadomych.

Wyniki przedstawiono na rys. 3.12a i 3.12b . Przyspieszenie dla 12 wątków to 11.38 dla elementów powłokowych typu „Solid-shell" i 11.26 dla elementów trójwymiarowych (Solid 3D). Natomiast wydajność to 93.36% dla elementów powłokowych i 93.33% dla elementów trójwymiarowych.



Rysunek 3.11: Półsfera obciążona dwoma parami sił. Przemieszczenia w kierunku środka półsfery.

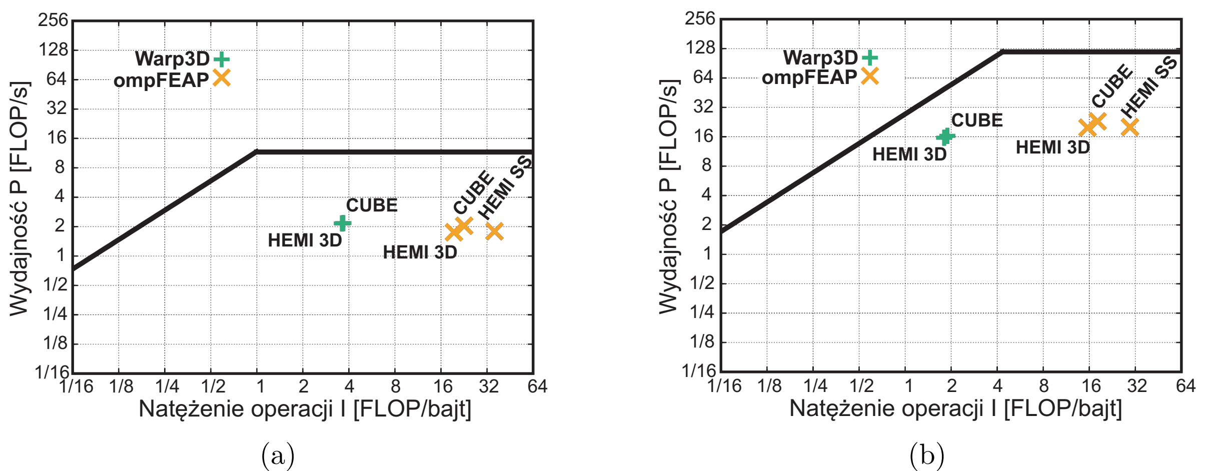


Rysunek 3.12: Półsfera obciążona dwoma parami sił. Przyspieszenie (a) i efektywność (b) ompFEAP dla elementów trójwymiarowych i powłokowych.

3.4.3 Model „Roofline"

Na rys. 3.13 przedstawiono model „Roofline" dla procesora Intel Xeon 5670 z zaznaczonymi wykonaniami pętli po elementach dla programów Warp3D oraz ompFEAP, z wykonaniem sekwencyjnym (rys. 3.13a) oraz równoległym (rys. 3.13b). Test kostki dla $N = 64$ został oznaczony jako „CUBE", test nieliniowej powłoki dla elementów 3D jako „HEMI 3D", a dla elementów typu „Solid Shell" jako „HEMI SS".

Można zaobserwować, że dla wykonania sekwencyjnego pętla po elementach dla obu programów jest ograniczona obliczeniowo. Jednak dla wykonania równoległego program Warp3D jest ograniczony pamięciowo, co się zgadza z naszymi obserwacjami w rozdz. 3.2.2. Widać również, że o ile dla wykonania sekwencyjnego programy Warp3D i ompFEAP mają podobną wydajność, to dla wykonania równoległego, wydajność ompFEAP jest wyższa o ok. 40%. Elementy typu „Solid Shell" wykazują większe natężenie operacji niż elementy 3D, co wynika z faktu, że elemeny „Solid Shell", są o wiele bardziej złożonymi elementami niż elementy 3D, a działają na podobnych danych wejściowych.



Rysunek 3.13: Wykres „Roofline" pętli po elementach: (a) dla 1 wątka i (b) 12 wątków.

3.4.4 Podsumowanie

W tym rozdziale przedstawiono podstawowe problemy związane z paralelizacją pętli po elementach w sekwencyjnej wersji programu FEAP [25] używając OpenMP; tę implementację oznaczono jako ompFEAP. Podstawową trudnością była skomplikowana architektura kodu oraz używanie wielu bloków „common”. Zaimplementowana wersja równoległej pętli po elementach ma następujące własności:

1. redukcja macierzy elementowych do macierzy globalnej jest wykonywana natychmiast po ich wygenerowaniu, bez dodatkowego magazynowania ich w pamięci; postępowanie to jest inne niż użyte np. w [110], czy np. w metodzie nazwanej „ekspansja tablic” w [8]. Użyte zostały tylko standardowe dyrektywy OpenMP.

2. Najistotniejszą częścią pętli jest redukcja macierzy elementowych do macierzy globalnej, dla której zaimplementowano i przetestowano wszystkie dostępne dyrektywy dostępne w OpenMP dla wspólnego wykluczania. Dyrektywa *ATOMIC* daje najlepszą skalowalność dla elementu trójwymiarowego, tj. przyspieszenie 10.7 i efektywność 95%. Dla porównania, dla Warp3D, otrzymano przyspieszenie 6.5 i efektywność 50%, oba przypadki dla 12 rdzeni. Dla elementów powłokowych typu „Solid Shell”, przyspieszenie jest nawet lepsze, ok. 11.38. Bardzo dobra skalowalność ompFEAP pokazuje, że nie są potrzebne dodatkowe, bardziej skomplikowane podejścia do zrównoleglania pętli po elementach.

Podsumowując zweryfikowano, że zarówno standardowe elementy trójwymiarowe, elementy powłokowe z FEAPa i elementy użytkownika działają w implementacji ompFEAP. Jednak potrzebna jest dalsza praca, aby sprawdzić inne funkcjonalności powyższego kodu.

4 Równoległe rozwiązywanie układów równań liniowych

4.1 Wprowadzenie

W tym podrozdziale przedstawiony zostanie krótki wstęp do algorytmów rozwiązywania rzadkich układów równań liniowych. Opracowano go na podstawie [38], uzupełniając o aspekty zapisu macierzy rzadkiej w pamięci komputera (opracowanie własne), dyskusji wyboru rozkładu **LU** (wg książki [104]) oraz opisu algorytmu zagnieżdżonego podziału (wg książki [96]).

4.1.1 Podstawowe algorytmy rozwiązywania układów równań liniowych

Bezpośrednie metody rozwiązywania równań liniowych w postaci $\mathbf{Ax} = \mathbf{b}$, są głównie oparte na rozkładzie macierzy $\mathbf{A}$ do postaci $\mathbf{A} = \mathbf{LU}$, gdzie $\mathbf{L}$ i $\mathbf{U}$ to odpowiednio macierz dolna i górna trójkątna.

Obliczanie współczynników dla tych macierzy trójkątnych jest też często nazywane rozkładem **LU** lub faktoryzacją. Po procesie faktoryzacji, pierwotny układ równań liniowych jest w prosty sposób rozwiązywany, poprzez rozwiązanie trójkątnych układów $\mathbf{Ly} = \mathbf{b}$ oraz $\mathbf{Ux} = \mathbf{y}$. Jeżeli $\mathbf{A}$ jest symetryczna, to proces faktoryzacji do postaci $\mathbf{A} = \mathbf{LL}^T$ lub $\mathbf{A} = \mathbf{LDL}^T$, może być zrealizowany za pomocą rozkładu Choleskiego, gdzie $\mathbf{L}$ jest dolną macierzą trójkątną (w przypadku faktoryzacji $\mathbf{A} = \mathbf{LDL}^T$ jest to macierz jednostkowa dolna trójkątna, tzn. taka, która ma jedynki na przekątnej) a $\mathbf{D}$ jest macierzą przekątniową.

Kod 4.1: Prosty algorytm kolumnowy do rozkładu **LU** dla macierzy $n \times n$.

```
1  subroutine LU_Decomp (A,n)
2   do i = 1 , n
3    do j = 1 , n
4     ! krok dzielenia oblicza kolumne i w L
5     A[j,i] = A[j,i]/A[i,i]
6    enddo
7
8    do k = i + 1 , n
9     do j = i + 1 , n
10     ! aktualizacja
11     A[j,k] = A[j,k] - A[j,i] * A[i,k]
12    enddo
13   enddo
14  enddo
15 end
```

Dwa przykładowe opisy algorytmów rozkładu **LU** i rozkładu Cholesky'ego zostały przedstawione jako Kod 4.1 i 4.2. Oczywiście możliwe jest wyprowadzenie innych sformułowań, które będą matematycznie tożsame, poprzez przegrupowanie pętli w tych algorytmach. Dla macierzy rzadkich, w których większość elementów jest równa zero,

algorytmy te muszą być odpowiednio dostosowane. Widać, że jeżeli $\mathbf{A}[j, i]$ jest zerowe w kroku dzielenia, to ta operacja nie musi być wykonana, podobnie krok aktualizacji może zostać pominięty, jeżeli $\mathbf{A}[j, i]$ lub $\mathbf{A}[i, k]$ ($\mathbf{A}[k, i]$ jeżeli $\mathbf{A}$ jest symetryczna) są zerami.

Kod 4.2: Prosty algorytm kolumnowy do rozkładu Choleskiego dla macierzy $n \times n$.

```
1  subroutine LU_Decomp (A,n)
2   do i = 1 , n
3    A[i,i] = Sqrt(A[i,i])
4    do j = j + 1 , n
5     ! krok dzielenia oblicza kolumne i w L
6     A[j,i] = A[j,i]/A[i,i]
7    enddo
8
9    do k = i + 1 , n
10    do j = k, n
11     ! aktualizacja
12     A[j,k] = A[j,k] - A[j,i] * A[k,i]
13    enddo
14   enddo
15  enddo
16 end
```

Jeżeli macierz $\mathbf{A}$ jest rzadka, to zazwyczaj macierze $\mathbf{L}$ i $\mathbf{U}$ mają więcej niezerowych elementów niż macierz $\mathbf{A}$. Ten efekt nazywa się wypełnieniem (ang. *fill-in*) i powoduje on, że zapotrzebowanie na pamięć oraz czas wykonania wzrasta ponadliniowo (w najbardziej negatywnym przypadku odpowiednio do $O(n^2)$ oraz $O(n^3)$ [18], gdzie n to liczba równań w układzie). Takie zachowanie można zaobserwować dla prawie wszystkich macierzy rzadkich wygenerowanych losowo [22], ale bardzo rzadko dla macierzy, które powstają w rzeczywistych zastosowaniach. Dlatego przy badaniu wydajności metod bezpośrednich dla macierzy rzadkich, zawsze trzeba odnosić się do macierzy pochodzących z rzeczywistych zastosowań. Odwrócenie całej macierzy rzadkiej zazwyczaj prowadzi do wzrostu wypełnienia do rzędu $O(n^2)$ i dlatego nie oblicza się macierzy odwróconej do rozwiązania układu $\mathbf{Ax} = \mathbf{b}$.

Pomimo wysokiego zapotrzebowania na pamięć, metody bezpośrednie są wykorzystywane w rzeczywistych zastosowaniach ze względu na ich ogólność i niezawodność (tzn. można je zastosować do większej grupy macierzy niż np. metody iteracyjne). Gdy potrzeba rozwiązać równanie z wieloma prawymi stronami ale z tą samą macierzą współczynników, metody bezpośrednie także pokazują swoją przydatność, ponieważ jednorazowy koszt faktoryzacji, może być zamortyzowany poprzez kilkukrotne wykonanie niekosztownych w realizacji rozwiązań trójkątnych układów.

4.1.2 Inne rozkłady macierzy

Oprócz rozkład **LU** oraz rozkładu Cholesky'ego istnieją również inne rozkłady macierzy służące do rozwiązywania układów równań liniowych. Takim rozkładem jest np. rozkład **QR**.

Rozkład **QR** macierzy rzeczywistej **A** jest to rozkład na dwie inne macierze oznaczone odpowiednio **Q** oraz **R**, gdzie macierz **Q** jest ortogonalna, natomiast **R** jest górną macierzą trójkątną (nazywana jest również prawą macierzą trójkątną - ang. *right triangular matrix*), co można zapisać $\mathbf{A} = \mathbf{QR}$. Macierz ortogonalna to macierz, której kolumny to jednostkowe wektory ortogonalne, co oznacza, że $\mathbf{QQ}^T = \mathbf{I}$. Po znalezieniu takiego rozkładu rozwiązanie układu $\mathbf{Ax} = \mathbf{b}$ również następuje w dwóch krokach: (1) najpierw trzeba rozwiązać równanie $\mathbf{Qy} = \mathbf{b}$, co sprowadza się do wyznaczenia iloczynu $\mathbf{y} = \mathbf{Q}^T\mathbf{b} = \mathbf{Qb}$, (2) następnie jest rozwiązywany układ trójkątny $\mathbf{Rx} = \mathbf{y}$.

Porównanie złożoności powyższego rozkładu z rozkładem **LU** jest następujące. Jeśli m jest wymiarem macierzy, to faktoryzacja **LU** wymaga $\frac{2}{3}m^3$ operacji. Faza rozwiązania (podstawiania wstecz i w przód) potrzebuje m^2 operacji, a więc cały proces wymaga $\frac{2}{3}m^3 + m^2$ operacji. Z kolei faktoryzacja **QR** wymaga $\frac{4}{3}m^3$. Wyprowadzenia tych oszacowań można znaleźć w [104, s. 151]. Widać, że współczynnik 2 jest tutaj podstawowym argumentem za rozkładem **LU** - trzeba wykonać dwa razy mniej operacji. Niektórzy autorzy [104, s. 152] wskazują również, na fakt, że eliminacja Gaussa jest znana od wielu stuleci, a faktoryzacja **QR** została wynaleziona dopiero po powstaniu komputerów, co spowodowało, że rozkład **LU** jest pierwszym wyborem dla większości badaczy. Aby faktoryzacja **QR** wyparła faktoryzację **LU**, musiałaby mieć istotną zaletę, która by przeważyła na jej korzyść. Dotychczas jej nie dostrzeżono.

4.1.3 Przechowywanie macierzy rzadkiej w pamięci komputera

Dzięki temu, że macierz **A** jest macierzą rzadką, nie trzeba zapisywać wszystkich elementów tej macierzy w pamięci komputera. Wystarczy pamiętać elementy niezerowe. Istnieje wiele formatów, które wykorzystują ten fakt np. format współrzędnych (ang. *coordinate format*), Skyline itp. - patrz [95].

Podstawowym formatem stosowanym w nowoczesnych algorytmach jest format spakowanych rzadkich kolumn - ang. *Compressed Sparse Column (CSC)* lub wierszy - ang. *Compressed Sparse Row (CSR)*. W tym formacie narzuca się pewną kolejność w uporządkowaniu elementów macierzy - przechowuje je się kolumnami, tak aby wygodnie było realizować np. działanie mnożenia macierzy przez wektor.

Każda kolumna jest reprezentowana poprzez listę wartości oraz odpowiadające im indeksy wierszy. Do przechowywania powyższych danych w pamięci komputera służą trzy tablice: `p` (wskaźniki - ang. *pointers*), `i` (indeksy) oraz `v` (wartości - ang. *values*).

Aby wygenerować taką reprezentację trzeba przejrzeć macierz dwa razy. Za pierwszym razem liczone są niezerowe elementy w każdej kolumnie, dzięki czemu można wypełnić tablicę wskaźników, jako kumulowana suma elementów niezerowych w kolumnie. W drugim przebiegu odpowiednie indeksy elementów niezerowych są zapisywane wraz z ich wartościami w odpowiednim miejscu w tablicach `i` i `v`.

W formacie CSC macierz rzadka $m \times n$, która zawiera nnz elementów niezerowych jest reprezentowana przez tablicę liczb całkowitych `p` o długości $n + 1$, tablicę liczb całkowitych `i` o długości nnz oraz tablicę liczb zmiennoprzecinkowych `v` o długości nnz. Indeksy wierszy dla elementów niezerowych kolumy j są przechowane między $\texttt{p}[j]]$, a $\texttt{p}[j+1]] - 1$, wartości tych elementów niezerowych są przechowywane w tych samych miejscach, ale w tablicy v. Jeśli liczono elementy tablicy od 1 (tak jak to jest w języku

Fortran), to pierwszy wpis w p[1] jest zawsze równy 1, a ostatni p$[n+1]$ pomniejszony o jeden jest równy liczbie elementów niezerowych w macierzy. Przykładowa macierz oraz jej reprezentacja w formacie CSC:

$$M = \begin{bmatrix} 3.2 & 6.7 & 0.0 & 1.6 \\ 1.2 & 3.1 & 2.8 & 0.0 \\ 0.0 & 7.2 & 0.0 & 7.3 \\ 0.0 & 0.0 & 3.4 & 5.0 \end{bmatrix} \tag{4.1}$$

```
p = (/   1,        3,              6,        8,          11 /)
i = (/   1,   2,   1,   2,   3,   2,   4,   2,   3,   4 /)
v = (/ 3.2, 1.2, 6.7, 3.1, 7.2, 2.8, 3.4, 1.6, 7.3, 5.0 /)
```

Format ten umożliwia dostęp do kolumn w bardzo szybki sposób, jednak bardzo kosztowne jest uzyskanie całego wiersza z takiej struktury. Podobnie modyfikacja struktury niezerowych elementów macierzy nie jest trywialna. Usuwanie bądź dodawanie pojedynczego wpisu do macierzy może zająć $O(nnz)$ czasu. Dlatego algorytmy wykorzystujące ten format muszą być odpowiednio dostosowane. Wszystkie, testowane w tej pracy, solwery korzystają z formatu CSR.

4.1.4 Algorytmy rozwiązywania rzadkich układów równań liniowych

Bezpośrednie rozwiązanie rzadkiego układu równań liniowych zazwyczaj składa się z pięciu faz. Dwie z nich to fazy obliczeniowe, które zostały wcześniej opisane - faktoryzacja numeryczna oraz rozwiązanie układów z macierzami trójkątnymi. Można zauważyć, że liczba niezerowych elementów po faktoryzacji i czasami również ich numeryczne właściwości, są funkcją początkowej permutacji układu wierszy i kolumn w macierzy współczynników. W wielu równoległych algorytmach rzadkiej faktoryzacji, ta początkowa permutacja układu ma również wpływ na równowagę obciążenia procesorów (wątków/rdzeni). Dlatego pierwszą fazą w bezpośredniej metodzie rozwiązania rzadkich układów równań liniowych, jest zastosowanie odpowiedniej permutacji do macierzy współczynników **A**. Fazę tą nazywa się przenumerowaniem (ang. *reordering*) i jeden ze sposobów jej wykonania zostanie omówiony w rozdz. 4.1.9.

Na macierz rzadką można również spojrzeć jak na macierz sąsiedztwa dla grafu. Algorytmy służące przenumerowaniu zazwyczaj używają grafowego przedstawienia macierzy i oznaczają wierzchołki w takim porządku jaki jest odpowiedni do obliczeń permutacji macierzy współczynników z odpowiednimi własnościami.

W drugiej fazie zwanej faktoryzacją symboliczną, oblicza się niezerowy wzorzec współczynników. Znajomość tego wzorca zanim współczynniki faktycznie będą obliczane, jest użyteczne z kilku powodów. Po pierwsze zapotrzebowanie na pamięć może być przewidziane przed wykonaniem obliczeń. Po drugie znajomość liczby i lokalizacji niezerowych elementów, może znacząco zmniejszyć liczbę odwołań do poszczególnych elementów, co ma znaczenie dla wydajności podczas faktoryzacji numerycznej. W równoległych implementacjach, faktoryzacja symboliczna pomaga w rozdzieleniu danych i obliczeń na poszczególne procesory. Przenumerowanie i faktoryzacja symboliczna czasami jest nazywana fazą wstępną (ang. *preprocessing*) lub fazą analizy (ang. *analysis*). Trzecią fazą jest wcześniej omawiana faktoryzacja numeryczna, a czwartą również już omówiona faza

rozwiązania układów trójkątnych. Czasami występuje piąta faza zwaną iteracyjnym poprawianiem (ang. *iterative refinement*), która jest wykonana po fazie rozwiązania, aby zwiększyć dokładność rozwiązania. Faza ta została opisana w rozdz. 4.4.1.

Z wyżej wymienionych faz zazwyczaj faza faktoryzacji numerycznej zajmuje najwięcej pamięci i czasu. W wielu zastosowaniach, np. w metodzie Newtona, stosowanej do rozwiązania nieliniowych równań, wielokrotnie faktoryzowana jest macierz o tej samej strukturze. W takich przypadkach pierwsze dwie fazy mogą zostać wykonane tylko raz, a później ich wyniki wykorzystywane wielokrotnie. Zazwyczaj te dwie fazy są skalowalne w mniejszym stopniu. Dzięki wykonaniu tych faz tylko jednokrotnie można utrzymać skalowalność całego procesu rozwiązania bliską temu jaką ma skalowalność faktoryzacji numerycznej. Z kolei faza czwarta jest bardzo zależna od zrównoleglenia trzeciej fazy faktoryzacji numerycznej. Równoległy algorytm fazy numerycznej od razu mówi, jak współczynniki są rozdzielone pomiędzy poszczególne procesory. A czwarta faza rozwiązania musi korzystać z takiego schematu równoległego, który działa na tym podziale danych. Z uwagi na to, że to właśnie faza faktoryzacji numerycznej ma największy wpływ na wydajność całego procesu rozwiązania, dalsze podrozdziały skupią się na tej fazie.

Algorytmy użyte dla fazy wstępnej i faktoryzacji rzadkiej macierzy współczynników, zależą głównie od własności tej macierzy, takich jak symetria, dominacja przekątniowa, dodatnia określoność itd. Jednakże w większości algorytmów faktoryzacji istnieją wspólne elementy. Dwa z nich, grafy zadań (ang. *task graphs*) i superwęzły (ang. *supernodes*) są kluczowe w dalszych rozważaniach.

4.1.5 Grafowy model zadań dla rzadkiej faktoryzacji

Obliczenia równoległe zazwyczaj są najbardziej efektywne jeśli zostaną uruchomione z maksymalnie możliwym poziomem granularyzacji, który zapewni dobry rozkład obciążenia pomiędzy wszystkie rdzenie. Faktoryzacja gęstej macierzy jest wymagająca obliczeniowo, tzn. potrzeba $O(n^3)$ operacji do sfaktoryzowania macierzy $n \times n$. Faktoryzacja macierzy rzadkich potrzebuje znacznie mniej operacji w wierszu czy kolumnie. Ponadto, rzadkość tych macierzy powoduje, że powstają dodatkowe wyzwania i możliwości przy paralelizacji.

Wyzwaniem jest znalezienie sposobu na organizację niestrukturalnych obliczeń, w taki sposób aby zrównoważyć pracę wszystkich rdzeni oraz zminimalizować konieczną komunikację między zadaniami przez zapewnienie małej liczby operacji w danym wierszu lub kolumnie macierzy. Dodatkowe możliwości wynikają z faktu, że w przeciwieństwie do standardowych algorytmów (przedstawionych w Kodach 4.1 i 4.2), w algorytmach dla rzadkich macierzy, kolumny współczynników nie muszą być obliczane jedna po drugiej. W Kodach 4.1 i 4.2, wiersz i kolumna i jest aktualizowana z użyciem wierszy i kolumn $1 \ldots i-1$. W przypadku rzadkich macierzy, kolumna i musi być zaktualizowana przez kolumnę $j < i$, tylko wtedy gdy $\mathbf{U}[j,i] \neq 0$ dla kolumn $1 \ldots i-1$. Ponadto w fazie faktoryzacji krok dzielenia może być wykonany równolegle dla wszystkich kolumn i, dla których $\mathbf{A}[i,j] = 0$ i $\mathbf{A}[j,i] = 0$ dla wszystkich $j < i$. Podobnie w każdym momencie faktoryzacji będą dostępne duże liczby kolumn, dla których można wykonać etap dzielenia. Do tych kolumn będzie można zaliczyć każdą niesfaktoryzowaną kolumnę i, jeśli wszystkie kolumny $j < i$ mają niezerowe elementy w wierszu i macierzy $\mathbf{L}$ i wszystkie wiersze $j < i$ z niezerowymi

elementami w kolumnie i macierzy $\mathbf{U}$ zostały już sfaktoryzowane.

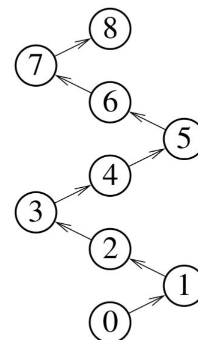
Graf zależności zadań (ang. *task dependency graph*) to narzędzie do szukania zależności w procesie faktoryzacji rzadkich macierzy. Jest to skierowany acykliczny graf (ang. *directed acyclic graph DAG*), którego wierzchołki to zadania, a krawędzie oznaczają zależności między zadaniami. Zadanie jest związane z każdym wierszem i kolumną (kolumną tylko w przypadku macierzy symetrycznych) macierzy, która podlega faktoryzacji. Wierzchołek i grafu oznacza zadanie odpowiedzialne za obliczenie kolumny i macierzy $\mathbf{L}$ i wiersza i macierzy $\mathbf{U}$. Zadanie jest gotowe do wykonania, tylko wtedy gdy wszystkie zadania, które są skierowane do tego zadania, zostały już ukończone. Graf zadań jest jawnie tworzony podczas symbolicznej faktoryzacji, aby wspomóc fazę faktoryzacji numerycznej. To pozwala faktoryzacji numerycznej uniknąć drogich przeszukiwań w celu znalezienia zadań gotowych do wykonania na każdym etapie równoległej faktoryzacji. Grafy zadań, które odpowiadają macierzom z symetryczną strukturą są drzewami i nazywa się je drzewami eliminacji (ang. *elimination trees*).

Aby otrzymać drzewo eliminacji dla macierzy symetrycznej, najpierw należy utworzyć macierz z wypełnieniami, przykłady takich macierzy przedstawiono na rys. 4.1a i rys. 4.2a, na których miejsca niezerowe oznaczono znakiem X. Macierz z wypełnieniami buduje się w następujący sposób. Dla każdej kolumny tworzy się, zbiór wierszy niezerowych, zostają wliczone również wiersze gdzie nastąpiło wypełnienie. Dla każdej pary z takiego zbioru, wiersz o numerze mniejszym oznacza kolumnę, a wiersz o numerze większym wiersz w macierzy, gdzie nastąpi wypełnienie, oznaczono je na rys. 4.1a i rys. 4.2a kwadratem.

Mając macierz z wypełnieniami, można przystąpić do budowania drzewa eliminacji. Algorytm tworzenia drzewa eliminacji dla macierzy symetrycznej jest następujący. Dla każdej kolumny macierzy z wypełnieniami należy wybrać wiersz o minimalnym numerze, w którym znajduje się niezerowy element. Wybrany wiersz będzie rodzicem dla danej kolumny w drzewie eliminacji. W ten sposób powstały drzewa eliminacji na rys. 4.1b i rys. 4.2b.

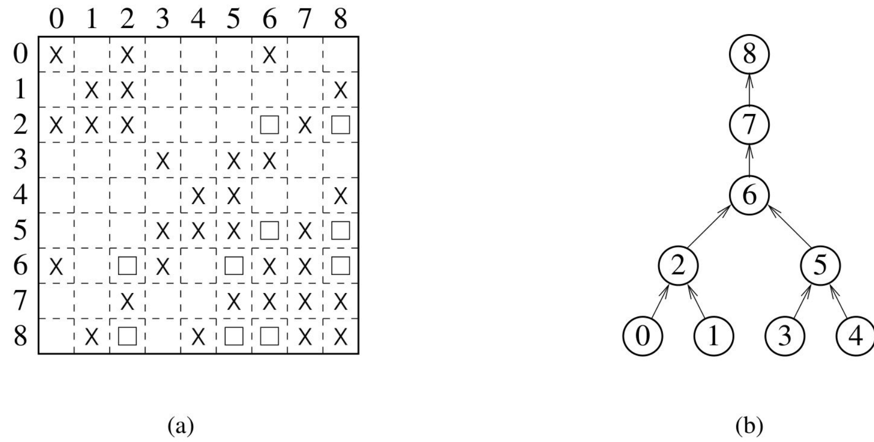
	0	1	2	3	4	5	6	7	8
0	X	X	X						
1	X	X	□	X	X				
2	X	□	X	□	X	X			
3		X	□	X	□	□	X		
4		X	X	□	X	□	X	X	
5			X	□	□	X	□	X	
6				X	X	□	X	□	X
7					X	X	□	X	X
8							X	X	X

(a)

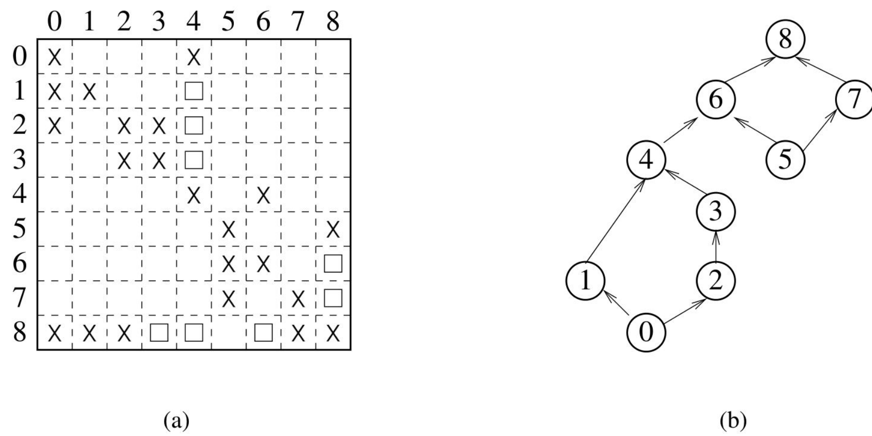


(b)

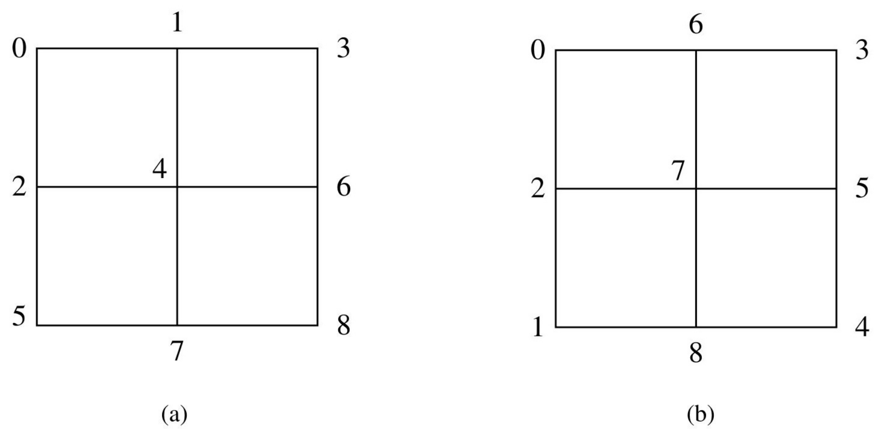
Rysunek 4.1: Strukturalnie symetryczna macierz rzadka (a) i jej drzewo eliminacji (b). X oznacza niezerowy element, a kwadrat dodatkowe wypełnienie po faktoryzacji.



Rysunek 4.2: Spermutowana macierz rzadka z rys. 4.1 - (a) i jej drzewo eliminacji - (b).



Rysunek 4.3: Niesymetryczna macierz rzadka (a) i graf zadań (b). X oznacza niezerowy element, a kwadrat dodatkowe wypełnienie po faktoryzacji.



Rysunek 4.4: Grafy przedstawiające dualność między macierzami z rys. 4.1 - (a) i rys. 4.2 - (b).

Rysunek 4.1. przedstawia przykład drzewa eliminacji dla macierzy symetrycznej rzadkiej, a rys. 4.3 ilustruje graf DAG dla macierzy strukturalnie niesymetrycznej. Gdy graf zadań jest skonstruowany, problem równoległej faktoryzacji (jak również faza równoległego rozwiązywania trójkątnego), może być postrzegana jako problem rozplanowania zadań na procesory. Dla maszyn z pamięcią lokalną preferowane jest statyczne planowanie, a dla maszyn z pamięcią wspólną planowanie dynamiczne. Kształt grafu zadań jest funkcją początkowej permutacji wierszy i kolumn rzadkiej macierzy, a to powoduje, że jest on wyznaczony w czasie fazy przenumerowania.

Rysunek 4.2 przedstawia drzewo eliminacji dla tej samej macierzy, co na rys. 4.1, ale z inną początkową permutacją. Struktura grafu DAG ma główny wpływ na to jak skutecznie można go rozplanować do wykonania równoległej faktoryzacji. Widać wyraźnie, że drzewo eliminacji z rys. 4.2b jest bardziej podatne na paralelizację niż drzewo dla tej samej macierzy ale z permutacją z rys. 4.1b. Rysunek 4.4 pokazuje, że obie macierze tworzą identyczny graf zależności, różnią się tylko oznaczeniem wierzchołków, co widać w różnej permutacji wierszy i kolumn macierzy, różnym poziomie wypełnienia oraz różnym wyglądzie grafów zadań. W ogólności można zauważyć, że długie i wąskie grafy zadań skutkują ograniczonymi możliwościami zrównoleglenia i długimi ścieżkami krytycznymi. Krótkie i szerokie grafy zadań posiadają wysoki stopień zrównoleglenia i krótsze ścieżki krytyczne.

Z uwagi na to, że kształt grafów zadań, a więc zdolność do efektywnej paralelizacji, jest bardzo czuła na odpowiednie przenumerowanie, trzeba wybrać taką heurystykę, która będzie tworzyła równomierne i szerokie grafy zależności. Najlepszą znaną do tej pory metodą jest zagnieżdżony bipodział (ang. *nested bisection*). Została ona zaimplementowana w bibliotece METIS [74] i opisana w rozdz. 4.1.9. Numeracja pokazana na rys. 4.2 bazuje na tym algorytmie.

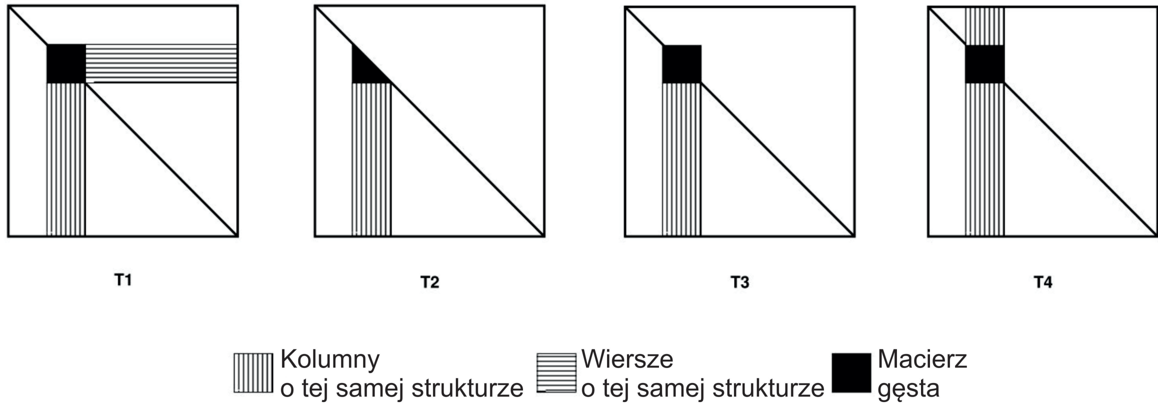
4.1.6 Superwęzły

Superwęzłem nazywa się zbiór kolejnych wierszy lub kolumn, które posiadają taką samą strukturę niezerowych elementów. W wielu zastosowaniach macierze współczynników mają naturalne superwęzły, np. w MES macierze elementowe często będą tworzyć jeden superwęzeł.

W terminologii grafowej zbiór wierzchołków o identycznej strukturze zależności nazywa się superwierzchołkiem (ang. *supervertex*). Gdy większość wierzchołków w grafie należy do superwierzchołków, a te superwierzchołki są podobnego rozmiaru (kumulują podobną liczbę wierzchołków) ze średnią m, to można pokazać, że skompresowany graf posiada $O(m)$ mniej wierzchołków i $O(m^2)$ mniej krawędzi niż w pierwotnym grafie. Można również wykazać, że z przenumerowania grafu skompresowanego wynika przenumerowanie grafu pierwotnego, z zachowaniem własności i jakości przenumerowania, poprzez proste oznaczanie wierzchołków z grafu pierwotnego w porządku w jakim występują superwierzchołki w grafie skompresowanym.

W ten sposób zapotrzebowanie na pamięć i czas może być bardzo zredukowane. Jest to szczególnie użyteczne dla równoległych metod rozwiązywania równań, ponieważ równoległe metody przenumerowania często dają przenumerowania gorszej jakości niż ich sekwencyjne odpowiedniki. Dla macierzy, których grafy zależności dają się bardzo skom-

presować, jest możliwe że faza przenumerowania wykonywana sekwencyjne ma niewielki wpływ na skalowalność całej metody, ponieważ przenumerowanie jest wykonane na grafie z mniejszą liczbą krawędzi o $O(m^2)$.



Rysunek 4.5: Cztery sposoby definiowania superwęzłów.

Oprócz tego, że superwęzły w macierzy współczynników przydają się w procesie przenumerowania, to obecność superwęzłów w macierzy po sfaktoryzowaniu ma największy wpływ na wydajność fazy faktoryzacji numerycznej i rozwiązania. Istnieje kilka sposobów na definiowanie superwęzłów dla niesymetrycznych macierzy. W pracy [19] przedstawia się 4 sposoby i są one pokazane na rys. 4.5. Najbardziej użyteczne sposoby to takie, gdzie grupa indeksów macierzy ma identyczny wzorzec elementów niezerowych odpowiednio w kolumnach **L** i wierszach **U**. Nawet jeśli nie ma superwęzłów w pierwotnej macierzy, to na pewno takie superwęzły pojawią po faktoryzacji w większości rzeczywistych zastosowań. Wynika to przede wszystkim ze zjawiska wypełnienia.

Przykładami superwęzłów po faktoryzacji są np. indeksy 6-8 na rys. 4.2, indeksy 2-3 i 7-8 na rys. 4.3 oraz indeksy 4-5 i 6-8 na rys. 4.1. Niektórzy praktycy sztucznie powiększają superwęzły poprzez dołączenie wierszy i kolumn, które nieznacznie różnią się pod względem wzorca niezerowych elementów. Superwęzły po faktoryzacji zazwyczaj są znajdywane podczas fazy symbolicznej faktoryzacji. W dalszej części tego rozdziału jako superwęzły będzie się rozumieć superwęzły po faktoryzacji.

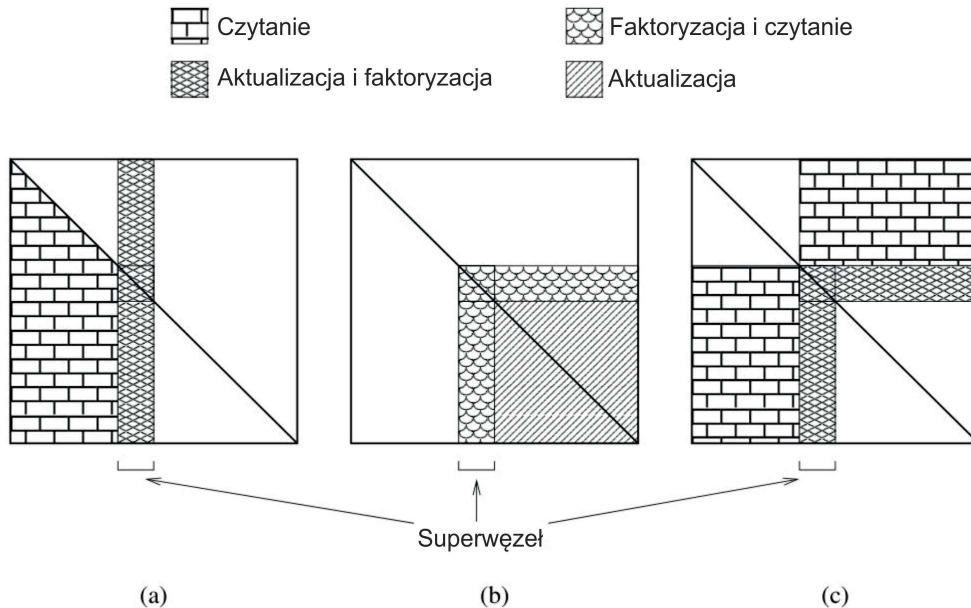
Jak można zauważyć w algorytmach z Kodu 4.1 i Kodu 4.2 są dwie podstawowe operacje: krok dzielenia i krok aktualizacji. Faktoryzacje bazujące na superwęzłach posiadają także te dwa kroki, ale teraz są one operacjami macierzowymi na blokach wierszy/kolumn na odpowiednich różnych superwęzłach.

Superwęzły podnoszą efektywność faktoryzacji numerycznej i fazy rozwiązywania układów trójkątnych, ponieważ zawierają operacje zmiennoprzecinkowe wykonywane na gęstych podmacierzach, a nie na pojedynczych niezerowych elementach, a to usprawnia hierarchiczne zarządzanie pamięcią. Wynika to z faktu, że wiersze i kolumny w superwęzłach mają tę samą strukturę niezerowych elementów; dodatkowo pośrednie adresowanie jest zminimalizowane, ponieważ struktura musi być zapamiętana tylko raz dla tych wierszy i kolumn. Superwęzły pomagają również zwiększyć ziarnistość obliczeń, co jest użyteczne przy polepszaniu obliczeń równoległych.

Grafowy model zadań rzadkich macierzy dla faktoryzacji, który został wcześniej opisany, zakładał że dane zadanie polega na faktoryzacji danej kolumny i wiersza macierzy. Z superwęzłami zadanie może być zdefiniowane, jako faktoryzacja wszystkich wierszy i kolumn związanych z superwęzłem. W praktyce grafy zadań są implementowane w większości solwerów jako grafy zadań superwęzłów.

4.1.7 Rzadka faktoryzacja bazująca na zadaniach

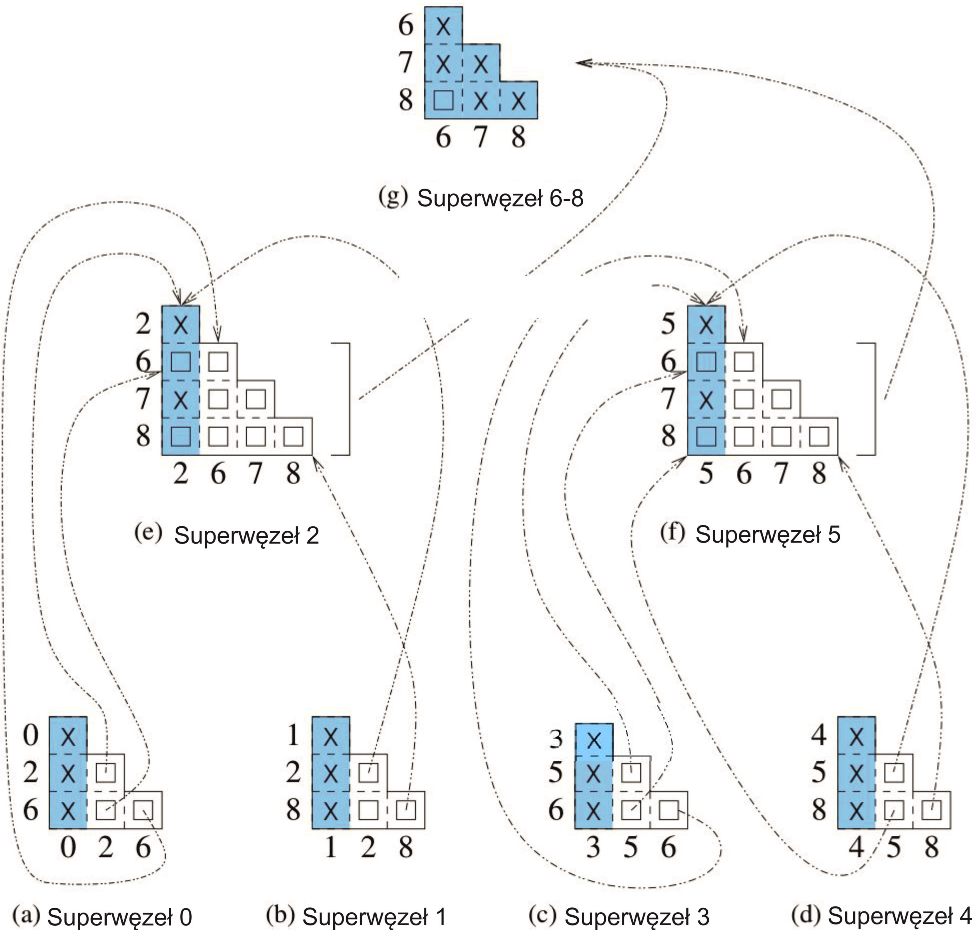
Istnieje wiele sposobów na dokładne zdefiniowanie zadań, każda z definicji będzie prowadziła do różnej równoległej implementacji faktoryzacji macierzy rzadkich. Jednak to zadanie związane z superwęzłem jest odpowiedzialne za obliczenie współczynników superwęzła po faktoryzacji. Z drugiej strony zadanie nie posiada wszystkich danych, które są potrzebne, żeby obliczyć wszystkie końcowe wartości wierszy i kolumn w superwęźle. Dane potrzebne do wykonanie kroku aktualizacji w danym superwęźle, mogą się znajdować w wielu innych superwęzłach. Można wyróżnić trzy metody, według których mogą działać zadania rzadkiej faktoryzacji **LU** : (1) lewe-przeszukiwanie, (2) prawe-przeszukiwanie, (3) Crout.



Rysunek 4.6: Trzy metody rzadkiej faktoryzacji **LU**: (a) lewe-przeszukiwanie, (b) prawe-przeszukiwanie, (c) Crout.

Powyższe warianty są przedstawione na rys. 4.6. Tradycyjna pierwsza metoda używa niezgodnych superwęzłów utworzonych zarówno z kolumn macierzy **L** i **U**, co nie jest za bardzo praktykowane. W tym przypadku zadanie jest odpowiedzialne za zebranie wszystkich potrzebnych danych z innych zadań, za aktualizację oraz za faktoryzację. Z tego powodu jest bardzo rzadko używaną metodą w obliczeniach na nowoczesnych komputerach dużej mocy. W drugiej metodzie zadanie faktoryzuje superwęzeł, który posiada i aktualizuje wszystkie współczynniki, do których ma udział dany superwęzeł. W metodzie Crouta zadanie odpowiedzialne jest za aktualizowanie i faktoryzację tylko

superwęzła, który posiada. Tylko druga i trzecia metoda można zastosować zarówno do rozkładów **LU** i $\mathbf{LL}^T$.



Rysunek 4.7: Macierze frontalne i przesyłanie danych w superwęzłowym multifrontalnym rozkładzie Choleskyego dla macierzy z rys. 4.2.

Istnieje także czwarta metoda zwana metodą multifrontalną, zawiera w sobie elementy obu metod - prawego-przeszukiwania i metody Crouta. W tej metodzie zadanie z danym superwęzeł oblicza swój wkład w aktualizację reszty macierzy (tak jak w metodzie prawego-przeszukiwania), ale nie wykonuje właściwych aktualizacji. Każde zadanie jest odpowiedzialne za zbieranie wszystkich odpowiednich aktualizacji i zastosowanie ich do swojego superwęzła (jak w metodzie Crouta), przed faktoryzacją superwęzła. Dane superwęzła i ich wkład do aktualizacji w metodzie multifrontalnej jest złożony w małe gęste macierze nazywane macierzami frontalnymi (ang. *frontal matrices*). Tablice z całkowitymi wartościami przechowują mapowanie między lokalnymi indeksami macierzy frontalnym, a globalnymi indeksami rzadkiej macierzy po faktoryzacji. Rysunek 4.7 przedstawia rozkład Choleskiego na superwęzłach metodą multifrontalną dla macierzy rzadkich z rys. 4.2. Z uwagi na to, że indeksy 6-8 tworzą superwęzeł to tworzą tylko jedno zadanie odpowiadające superwęzłowi w grafie zadań (drzewie eliminacji).

Kiedy zadanie jest gotowe do wykonania, najpierw konstruuje swoją macierz frontalną

poprzez łączenie udziałów z macierzy frontalnych swoich potomków i z macierzy współczynników. Wtedy faktoryzuje swój superwęzeł, który jest zacieniowaną częścią macierzy z rys. 4.7. Po faktoryzacji, niezacieniowana część (ta podmacierz macierzy frontalnej jest nazywana macierzą aktualizacji (ang. *update matrix*)) jest aktualizowana bazując na kroku aktualizacji z Kodu 4.1. Następnie tak skonstruowana macierz aktualizacji jest użyta przez rodzica zadania do stworzenia swojej macierzy frontalnej.

Rysunek 4.7 przedstawia symetryczną multifrontalną faktoryzację, dlatego wszystkie macierze są trójkątne. W ogólności dla dekompozycji **LU**, te macierzy byłyby kwadratowe lub prostokątne. W symetrycznej faktoryzacji multifrontalnej, aktualizacja macierzy potomka w drzewie eliminacji, ma udział tylko w macierzy frontalnej jego rodziców. Jednak graf zadań generalnie nie jest drzewem, ale skierowanym acyklicznym grafem (DAG) jak pokazano na rys. 4.3. Oprócz struktury macierzy frontalnych i aktualizacji, metoda multifrontalna dla niesymetrycznych macierzy, różni się jeszcze w dwóch aspektach. Po pierwsze macierz aktualizacji może być potrzebna dla więcej niż jednej macierzy frontalnej. Po drugie dane dla macierzy frontalnej mogą pochodzić niekoniecznie od bezpośrednich przodków superwęzła, ale również od dalszych przodków z grafu zadań.

Metoda multifrontalna jest częstym wyborem przy równoległych implementacjach faktoryzacji dla macierzy rzadkich. Głównie z powodu naturalnej lokalności danych (większość pracy przy faktoryzacji jest wykonywana na bardzo gęstych macierzach frontalnych) i dzięki temu łatwości synchronizacji. W ogólności każdy superwęzeł jest aktualizowany przez wiele innych superwęzłów i może on aktualizować wiele innych superwęzłów w procesie faktoryzacji. W przypadku implementacji naiwnej, wszystkie aktualizacje mogą powodować zakleszczenia i potrzebę synchronizacji na maszynach z pamięcią wspólną albo generować nadmierny ruch komunikatów na maszynach z pamięcią lokalną. Dodatkowo, w metodzie multifrontalnej aktualizacje są skumulowane i przesyłane na ścieżkach od liści do korzenia grafu zadań.

W zwykłej superwęzłowej rzadkiej faktoryzacji graf zadań jest skonstruowany w taki sposób, że zadanie zwiększa swoją wielkość wraz ze zbliżaniem się do korzenia, a liczba równoległych zadań się zmniejsza. Metoda multifrontalna jest dobrze dopasowana do obu rodzajów paralelizacji: zadaniowej (blisko liści) i na danych (blisko korzenia). Większe zadania, które wymagają dużych macierzy frontalnych blisko korzenia, mogą użyć wielu wątków lub procesów aby skorzystać z operacji równoległych na macierzach gęstych. Te operacje mają nie tylko dobrze opracowane równoległe algorytmy, ale również ich implementacje są bardzo dojrzałe, tzn. od bardzo wielu lat są testowane przez użytkowników, dzięki czemu większość błędów i problemów z wydajnością udało się usunąć.

4.1.8 Wybór elementu głównego w równoległych rzadkich faktoryzacjach

Do tej pory omówiono scenariusz, w którym wiersze i kolumny macierzy są przestawiane w fazie przenumerowania i to przenumerowanie pozostaje niezmienione w czasie numerycznej faktoryzacji. To założenie jest prawdziwe dla bardzo dużej klasy praktycznych problemów, ale są pewne zastosowania, które generują macierze, które posiadają zera albo bardzo małe wartości na przekątnej podczas faktoryzacji. To z kolei powoduje, że krok dzielenia w rozkładzie **LU** może być niemożliwy do wykonania albo prowadzić do numerycznej niestabilności.

Problem ten można rozwiązać poprzez zamianę wierszy i kolumn w macierzy, metoda ta jest znana w polskiej literaturze jako częściowy wybór elementu głównego (ang. *partial pivoting*). Kiedy napotyka się na element bardzo mały bądź bliski zeru w miejscu $\mathbf{A}[i,i]$ przed krokiem dzielenia, wtedy wiersz i jest zamieniany z innym wierszem j $(i < j \leqslant n)$, takim, że moduł $\mathbf{A}[j,i]$ (będzie on zajmował miejsce $\mathbf{A}[i,i]$ po zamianie) jest odpowiednio większy porównując do innych elementów $\mathbf{A}[k,i]$ $(i < k \leqslant n, k \neq i)$. Podobnie, zamiast wiersza i, kolumna i będzie zamieniona z odpowiednią kolumną j $(i < j \leqslant n)$. Zarówno wiersz i jak i kolumna i są zamieniane równocześnie z odpowiednią parą wiersz-kolumna aby zachować symetrię.

Do niedawna sądzono, że przez nieprzewidywalne zmiany w strukturze spowodowane częściowym wyborem elementu głównego, wykonanie na wstępie przenumerowania i symbolicznej faktoryzacji nie może być przeprowadzone oraz że te kroki powinny być połączone z numeryczną faktoryzacją. Z drugiej strony oddzielne procesy analizy i numerycznej faktoryzacji mają ogromny wpływ na wydajność i duże korzyści przy zrównolegleniu, które byłyby stracone gdyby te procesy połączyć. Nowoczesne algorytmy równoległe dla macierzy rzadkich potrafią przeprowadzić wybór elementu głównego z zachowaniem stabilności bez mieszania tych dwóch kroków. Metoda multifrontalna pozwala efektywnie zaimplementować równoległy częściowy wybór elementu głównego, a zarazem jego efekty są widoczne tak bardzo lokalnie, jak to jest możliwe.

Przed wykonaniem procesu przenumerowania, wiersze i kolumny są permutowane w ten sposób, że wartość bezwzględna iloczynu elementów na przekątnej jest maksymalna. Do tego można użyć specjalnych algorytmów do znajdowania skojarzeń w grafie. Ten krok daje pewność, że elementy na przekątnej mają relatywnie dużą wartość na początku faktoryzacji. Zaobserwowano, że jeśli wykona się ten krok, w większości przypadków, potrzebna bardzo mało zmian elementów, żeby utrzymać numeryczną stabilność, w procesie faktoryzacji. Wtedy faktoryzacja może być wykonana ze statycznym grafem zadań i statyczną strukturą superwęzłów dla $\mathbf{L}$ i $\mathbf{U}$ utworzoną w trakcie faktoryzacji symbolicznej. Kiedy zmiana jest jednak niezbędna, zmiany są rejestrowane w pewnych strukturach danych. Dzięki temu, że zmiany są rzadkie, jest mało wymian, a także nakład obliczeniowy nie jest duży.

W metodzie multifrontalnej częściowy wybór elementu głównego wykonuje się na poziomie macierzy frontalnej. Wymiana wierszy i kolumn w danym superwęźle jest lokalna i jeśli wszystkie wiersze i kolumny superwęzła mogą być sfaktoryzowane, to przez prostą zamianę kolejności, nic nie musi być zrobione poza superwęzłem. Czasami jednak nie można przeprowadzić tego w ten sposób. Zdarza się to, gdy wszyscy kandydaci z wierszy i kolumn do wymiany mają indeksy większe niż ostatnia para wiersz-kolumna w superwęźle. W tym przypadku stosowana jest technika opóźnionego wyboru elementu głównego (ang. *delayed pivoting*).

4.1.9 Algorytm zagnieżdżonego podziału

Ostatnim aspektem, który zostanie poruszony w tym wprowadzeniu, jest sposób przenumerowania wierszy macierzy, tak aby utworzona, w wyniku faktoryzacji, dekompozycja **LU** miała najmniejsze możliwe wypełnienie (ang. *fill-in*). Niestety proces ten jest NP-zupełny [118], a to powoduje że trzeba użyć metod heurystycznych, aby uzyskać dobre

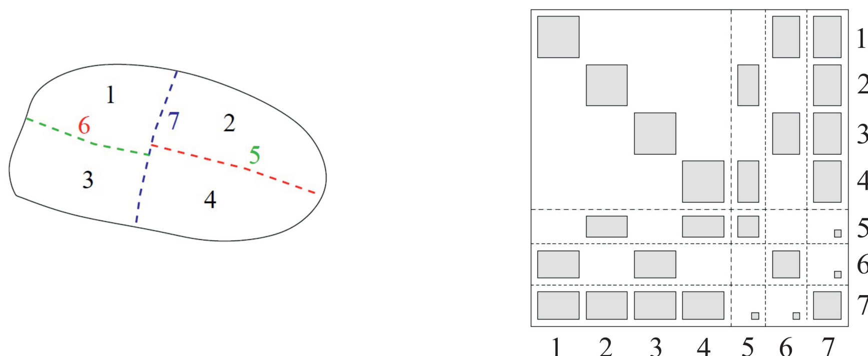
rozwiązanie w rozsądnym czasie. Najlepszy algorytmem przetestowanym przez autora w rozdz. 4.2.4 jest metoda bazująca na zagnieżdżonym podziale grafu (ang. *nested dissection (ND)*), w polskiej literaturze znana również jako metoda włożonych przekrojów [27]. Tutaj zostanie przedstawiona za [96].

Metoda zagnieżdżonego podziału zostanie opisana za pomocą rekursji oraz pojęcia *separatora*. Zbiór wierzchołków S w grafie jest nazywany separatorem, jeśli po usunięciu tego zbioru z grafu, graf zostanie podzielony na dwa rozłączne podgrafy. Wtedy metoda zagnieżdżonego podziału wygląda następująco:

Kod 4.3: Zagdnieżdżony podział grafu - ND(G,nmin)

```
1 if |V| <= nmin
2  Ponumeruj wierzcholki w V
3 else
4  Znajdz separator S dla V
5  Ponumeruj wierzcholki w S
6  Podziel V na dwa grafy R i L przez usuniecie S
7  ND(R,nmin)
8  ND(L,nmin)
9 end
```

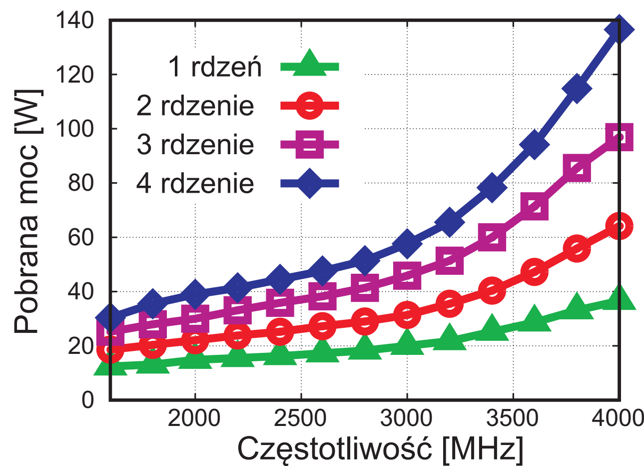
Główny krok w tej procedurze to podzielenie grafu na trzy części, z których dwie nie posiadają ze sobą wierzchołków wspólnych. Trzeci zbiór posiada wspólne wierzchołki z pozostałymi częściami i jest oznaczony jako separator. W dalszej kolejności wykonywane jest numerowanie wierzchołków w separatorze. Następnie proces jest powtarzany rekursywnie w częsciach, które nie mają wspólnych wierzchołków. Jeśli graf ma niewielką liczbę wierzchołków (w Kodzie 4.3 mniej niż *nmin*), wierzchołki są numerowane. Numerowanie wierzchołków wykonuje się po kolei, dzięki temu wierzchołki są ponumerowane w pewnym porządku, tak że numerowanie przed podziałem grafu zaczyna się od ostatnio ponumerowanego wierzchołka. Wierzchołki separatora zazwyczaj są numerowane od końca. Rysunek 4.8 przedstawia podział grafu i odpowiadające mu przenumerowanie.



Rysunek 4.8: Podział grafu i odpowiadająca mu przenumerowana macierz. Rysunek za [96, str. 95].

4.1.10 Inne aspekty optymalizacji faktoryzacji numerycznej

W pracy [18] zwrócono także uwagę, że czas i pamięć to nie jedyne zasoby, których wymaga algorytm faktoryzacji. Prąd to również zasób, który bardzo rzadko jest rozważany przez projektantów algorytmów. Praca [13] przedstawia równoległą faktoryzację Choleskiego opartą na superwęzłach, która ma za zadanie dbać o skalowalność napięcia i częstotliwości. Procesor uruchomiony przy niższej częstotliwości/napięciu jest wolniejszy, ale używa zdecydowanie mniej prądu, patrz rys 4.9. W ich metodzie procesor, który wykonuje obliczenia na superwęzłach po krytycznych ścieżkach, pracuje z maksymalną prędkością, ale procesory, które wykonują obliczenia poza krytycznymi ścieżkami mają obniżoną częstotliwość/napięcie. Spowolnienie tych procesorów, nie podnosi całkowitego czasu wykonania faktoryzacji, ale obniża zapotrzebowanie na prąd.



Rysunek 4.9: Pobrana moc procesora Intel Core i7-3770K, opracowanie własne na podstawie [116].

4.2 Porównanie różnych implementacji

4.2.1 Testowane implementacje

W tym podrozdziale porównano pięć różnych implementacji algorytmów do rozwiązywania rzadkich układów równań liniowych:

1. **PARDISO** Solwer PARDISO został opracowany przez grupę prof. Schenka z Uniwersytetu Szwajcarskiego w Lugano. Zgodnie z zapewnieniami autorów jest to wysoko wydajny, niezawodny, oszczędny pamięciowo i łatwy w użyciu kod do rozwiązywania rzadkich układów równań liniowych symetrycznych i niesymetrycznych na maszynach z pamięcią wspólną oraz na maszynach z pamięcią rozproszoną. Solwer używa kombinacji prawych i lewych przeszukiwań oraz operacji poziomu 3 (operacje macierz-macierz) biblioteki BLAS dla superwęzłów [98]. Równoległe metody wyboru elementu głównego (ang. *pivoting*) pozwalają na pełny wybór elementu głównego na superwęźle, który ma na celu zrównoważenie numerycznej stabilności i skalowalności faktoryzacji. Dla odpowiednio dużych problemów, testy numeryczne

pokazują, że skalowalność wykorzystanych algorytmów jest praktycznie niezależna od wykorzystanej architektury (pamięć wspólna, czy lokalna). Autorzy raportują przyspieszenie rzędu 7 dla maszyn z 8 rdzeniami. Solwer wykorzystuje dyrektywy OpenMP i MPI dla paralelizacji i był użyty przez społeczność obliczeniową na ogromnej liczbie systemów z pamięcią wspólną. Dodatkowo solwer ten ma unikalną własność, która powoduje, że wyniki obliczeń są dokładnie takie same z precyzją każdego bitu na maszynach wielordzeniowych i na klastrach komputerów.

2. **MKL PARDISO** Solwer PARDISO z biblioteki Math Kernel Library (MKL) firmy Intel [87] bazuje na pakiecie z poprzedniego punktu, ale na wersji z 2006 roku, więc wiele usprawnień wprowadzonych od tamtej pory nie zostało w nim ujętych. Z drugiej strony maszyny, na których zostały przeprowadzone wszystkie testy w niniejszej pracy, posiadają procesory firmy Intel, a ona miała pełen dostęp do wiedzy o architekturze tych procesorów i mógł stworzyć kod w pełni zoptymalizowany pod te procesory.

3. **HSL MA86** Harwell Subroutine Library (HSL) to biblioteka do wieloskalowych obliczeń naukowych, napisana i rozwijana przez Numerical Analysis Group w STFC Rutherford Appleton Laboratory. Kod źródłowy do celów akademickich udostępniany jest bez opłat . Solwer (HSL MA86 stanowi metodę pierwszego wyboru sugerowaną przez jego autorów dla macierzy symetrycznych nieokreślonych. Metoda ta dzieli fazę faktoryzacji na zadania, każde zajmuje się pojedynczym blokiem albo blokiem kolumn. Powstają trzy różne typy zadań: (1) faktoryzuj kolumnę, (2) aktualizuj wewnętrznie, (3) aktualizuj dane między superwęzłami (patrz rozdz. 4.3). Pełny opis można znaleźć w [43]. Gdy wszystkie dane, których zadanie potrzebuje, są dostępne, zadanie jest umieszczane w puli zadań do wykonania przez którykolwiek z dostępnych rdzeni. Cały proces faktoryzacji jest reprezentowany przez skierowany acykliczny graf, gdzie wierzchołki reprezentują zadania, a krawędzie reprezentują zależności między zadaniami.

4. **HSL MA97** To inna metoda z biblioteki HSL dla macierzy symetrycznych. Wykorzystuje ona technikę macierzy multifrontalnych (patrz rozdz. 4.1.7). Dodatkowo zapewnia bitową kompatybilność rozwiązania niezależnie od użytej liczby wątków. Wykorzystuje OpenMP do paralelizacji na dwóch płaszczyznach: (1) paralelizacja na poziomie drzewa (ang. *tree-level parallelism*) (2) paralelizacja na poziome węzła (ang. *node-level parallelism*) W pierwszym przypadku różne poddrzewa są faktoryzowane przez niezależne zadania i aby zapewnić bitową kompatybilność kolejność łączenia rozwiązań od dzieci jest stała w czasie. W drugim przypadku operacje, takie jak np. gęsta faktoryzacja, są rozbite na wiele zadań. Tutaj bitowa kompatybilność jest zapewniona dzięki paralelizacji danych (ang. *data parallelism*), tzn. każda suma wyliczana jest sekwencyjnie.

5. **WSMP** Watson Sparse Matrix Package jest rozwijany przez firmę IBM [37]. Do rozwiązywania układów symetrycznych używa zmodyfikowanej wersji metody macierzy multifrontalnych dla rzadkich rozkładów Choleskiego oraz wysoko skalowalny równoległy algorytm rozkładu Choleskiego dla macierzy rzadkich. W przypadku

wykonania równoległego, przypisuje on wszystkie wątki do korzenia drzewa eliminacji i rekursywnie przydziela wątki każdego rodzica do ich dzieci w drzewie, aby uzyskać zbalansowane obciążenie pracą. Gęste operacje na macierzach frontalnych w wierzchołkach drzewa są przydzielane więcej niż jednemu wątkowi i są również sparalelizowane. Wątki są zarządzane poprzez silnik wątków (ang. *task-parallel engine*), który dzięki wykorzystaniu metody „kradzieży pracy” (ang. *work-stealing*), uzyskuje małą ziarnistość obliczeń. Solwer używa również skalowalnej równoległej wersji algorytmu rozwiązywania rzadkich układów trójkątnych oraz ulepszoną i sparalelizowaną wersję algorytmu wielopoziomowego zagnieżdżonego dzielenia (ang. *multilevel nested dissection*), podobnego do wykorzystywanego w pakiecie METIS [74]. Dla macierzy nieokreślonych, solwer używa bloków 1×1 oraz 2×2, podobnie jak algorytm w [10].

Tabela 4.1 zestawia własności powyższych implementacji. Pierwsza kolumna tej tabeli to nazwa implementacji. Następne 3 kolumny opisują rodzaj faktoryzacji, które są dostępne: **LU**, Cholesky oraz $\mathbf{LDL}^T$ dla macierzy nieokreślonych i symetrycznych. Jeśli faktoryzacja $\mathbf{LDL}^T$ używa bloków 2×2 dla wyboru elementu głównego to wpisano „2”, w przeciwnym przypadku „1”. Kolejna kolumna określa czy solwer wspiera macierze o współczynnikach zespolonych. W następnych dwóch kolumnach oznaczono, czy dostępne są obecnie najpopularniejsze metody przenumerowania wierszy/kolumn macierzy, pierwsza minimalnego stopnia (MD) jest opisana w rozdz. 4.2.4, druga metoda zagnieżdzony podział (ND) została opisana w rozdz. 4.1.9. Kolejna kolumna opisuje typ pamięci, na której może pracować biblioteka („S” dla pamięci wspólnej, „D” dla pamięci rozproszonej). Ostatnia kolumna opisuje, jaką metodą przeprowadzana jest faktoryzacja.

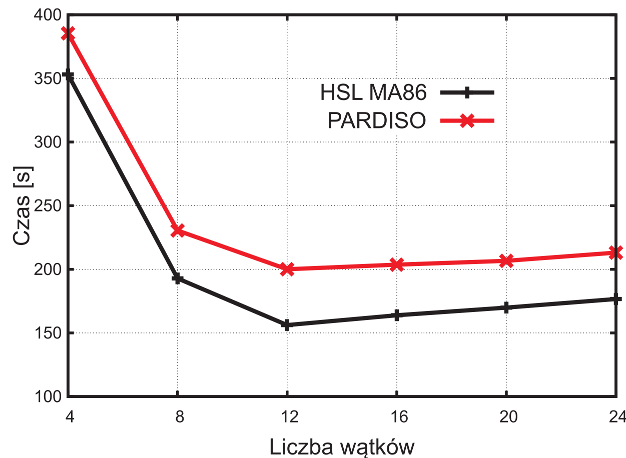
Testy zostały przeprowadzone na klastrze GRAFEN [29]. Wykorzystano jeden węzeł tego klastra, który zawiera 2 procesory Xeon X5650 2.66 GHz z 6 rdzeniami (łącznie 12 rdzeni) i 24GB DDR3 1333MHz pamięci. Do każdego solwera podłączono bibliotekę BLAS z biblioteki MKL [76] firmy Intel.

Tabela 4.1: Zestawienie własności testowanych implementacji.

Implementacja	LU	Cholesky	LDL^T	zespolone	MD	ND	typ pamięci	metoda faktoryzacji
PARDISO	X	X	2	X	X	X	SD	superwęzły z przeszukiwaniem lewym/prawym
MKL PARDISO	X	X	2	X	X	X	SD	superwęzły z przeszukiwaniem lewym/prawym
HSL MA86	-	X	2	X	X	X	S	superwęzły z przeszukiwaniem prawym
HSL MA97	-	X	2	X	X	X	S	multifrontalna
WSMP	X	X	1	X	X	X	SD	multifrontalna

4.2.2 Jednoczesna wielowątkowość

Przetestowano działanie wybranych solwerów pod względem wydajności, gdy uruchomiono jednoczesną wielowątkowość (SMT - ang. *simultaneous multithreading*), patrz rozdz. 2.6.3. Rysunek 4.10 przedstawia czas wykonania, dla różnej liczby wątków, w tym liczby większej od liczby fizycznych rdzeni. Widać z niego, że przy większej liczbie wątków niż liczba fizycznych rdzeni (ale nie logicznych rdzeni, patrz rozdz. 2.6.3), solwery rozwiązujące układy równań zwalniają. Dlatego w dalszych obliczeniach nie będzie używana większa liczba wątków niż liczba dostępnych fizycznie rdzeni.



Rysunek 4.10: Czas faktoryzacji dla różnej liczby wątków na maszynie z 12 rdzeniami fizycznymi. Test kostki $N = 64$.

4.2.3 Koligacja wątków

Przetestowano również różne ustawienia koligacji wątków (patrz rozdz. 2.6.4), które są dostępne dla procesorów firmy Intel poprzez interfejs KMP_AFFINITY. Interfejs ten posiada dwie główne opcje:

1. Rodzaj (ang. *type*) określa sposób rozmieszczania wątków na rdzeniach. Dostępne są dwa sposoby:
   (a) kompaktowy (ang. *compact*) - nowemu wątkowi przydziela rdzeń, który znajduje się najbliżej rdzenia, który został przydzielony ostatnio utworzonemu wątkowi,
   (b) rozproszony (ang. *scatter*) - przydziela wątek do rdzenia tak, żeby wątki były możliwie najbardziej równo rozdzielone na rdzenie i inne zasoby procesora (pamięci podręczne itp.). Jest przeciwnością rozmieszczenia typu kompaktowego.
2. Ziarnistość (ang. *granularity*) określa możliwość przerzucania wątku z jednego rdzenia logicznego na inny. Ta opcja dotyczy tylko procesorów z włączoną jednoczesną wielowątkowością (por. rozdz. 4.2.2). Dostępne są dwa sposoby:

(a) drobno (ang. *fine*) - wątki nie mogą być przerzucane między rdzenie logiczne,

(b) przy rdzeniu (ang. *core*) - wątki mogą być przerzucane między rdzenie logiczne w obrębie rdzenia fizycznego.

Tabela 4.2: Porównanie czasów względnych dla różnych koligacji wątków. Test kostki dla $N = 64$. Solwer HSL MA86, 12 wątków.

	COMPACT	SCATTER
FINE	2.13	1.00
CORE	2.14	1.01

Tabela 4.2 przedstawia wyniki dla wszystkich 4 kombinacji powyższych opcji. Testy wykonano dla solwera HSL MA86 i 12 wątków. Widać z niej, że największe korzyści przynosi opcja SCATTER oraz FINE i one będą używane w dalszej części pracy. Włączenie funkcji COMPACT powoduje, że dwa wątki trafiają na ten sam rdzeń fizyczny, co przekłada się na dwukrotnie większy czas wykonania.

4.2.4 Permutacja wierszy macierzy

Jak już wspomniano wcześniej bardzo dużym problemem przy dokładnym rozwiązywaniu układów równań jest efekt wypełnienia (ang. *fill-in*). Dlatego stosuje się różne algorytmy do przeprowadzenia permutacji wierszy w macierzy wejściowej, tak aby po wykonaniu faktoryzacji współczynnik wypełnienia był jak najmniejszy. To powoduje zmniejszenie kosztów obliczeniowych, a również w oczywisty sposób zmniejszy użycie pamięci.

Niestety proces ten jest NP-zupełny (patrz rozdz. 4.1.9), a to powoduje że trzeba użyć metod heurystycznych. Każda z implementacji omówionych w rozdz. 4.2.1 sugeruje użycie biblioteki METIS [74] do wykonania tego procesu. W bibliotece HSL zaimplementowano także 3 inne metody przenumerowania. Dodatkowo przez autora rozprawy został zaimplementowany interfejs do biblioteki mtMETIS [75], który jest wielowątkową wersją biblioteki METIS. W sumie daje to pięć następujących metod, które mogły być użyte z tym samym solwerem HSL MA86:

1. **Wybór poprzez kryterium Markowitza** - oznaczona jako cMark - Ta metoda wybiera diagonalne pivoty rzędu 1 lub 2 poprzez kryterium Markowitza. Metoda zastosowana w HSL jest opisana w [24].

2. **Minimalny stopień** - (ang. *minimum degree*) oznaczona jako MD. Jest to grafowy sposób znajdowania odpowiednich permutacji poprzez dzielenie grafu. Graf jest dzielony poprzez wybór wierzchołka z minimalnym stopieniem w aktualnym grafie. Kluczową decyzją jest wybór wierzchołka przy równych stopniach wielu wierzchołków. Dokładny opis metody zastosowanej w HSL znajduje się w [23].

3. **Przybliżony minimalny stopień** - (ang. *approximate minimum degree*) oznaczona jako AMD. Ta metoda różni się od poprzedniej tym, że przybliża minimalny stopień poprzez obliczaniem jego górnej granicy i na podstawie tej górnej granicy

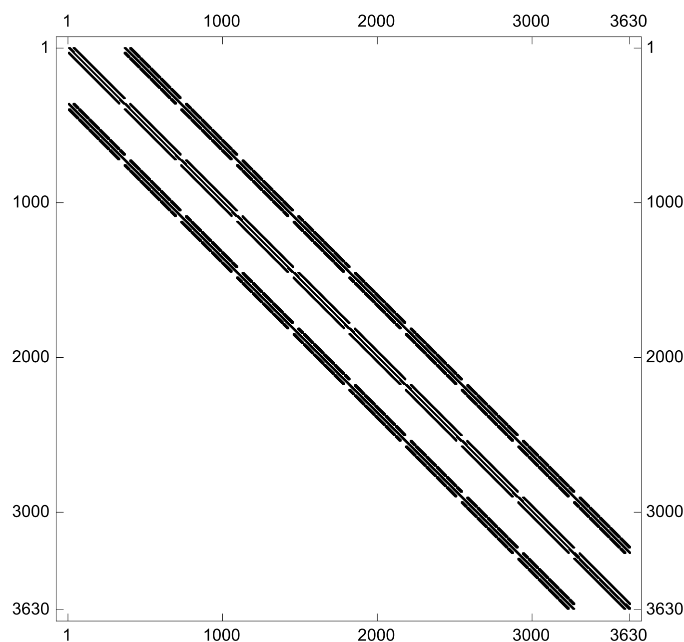
wybiera kolejne permutacje. Testy numeryczne wykazały, że ta metoda daje permutacje o jakości porównywalnej do najlepszej klasycznej metody MD [2]. Metoda zastosowana w HSL jest opisana w [21].

4. **METIS** - metoda korzysta z algorytmu zagnieżdżonego bipodziału (ang. *nested bisection*) opisanego w rozdz. 4.1.9. Do znajdywania separatora metoda wykorzystuje algorytm wielopoziomowego k-kierunkowego podziału grafów. Ze względu wykorzystanie tej metody również do dekompozycji obszaru, została ona opisana w dodatku B, szczegółowy jej opis znajduje się również w [59] i [60].

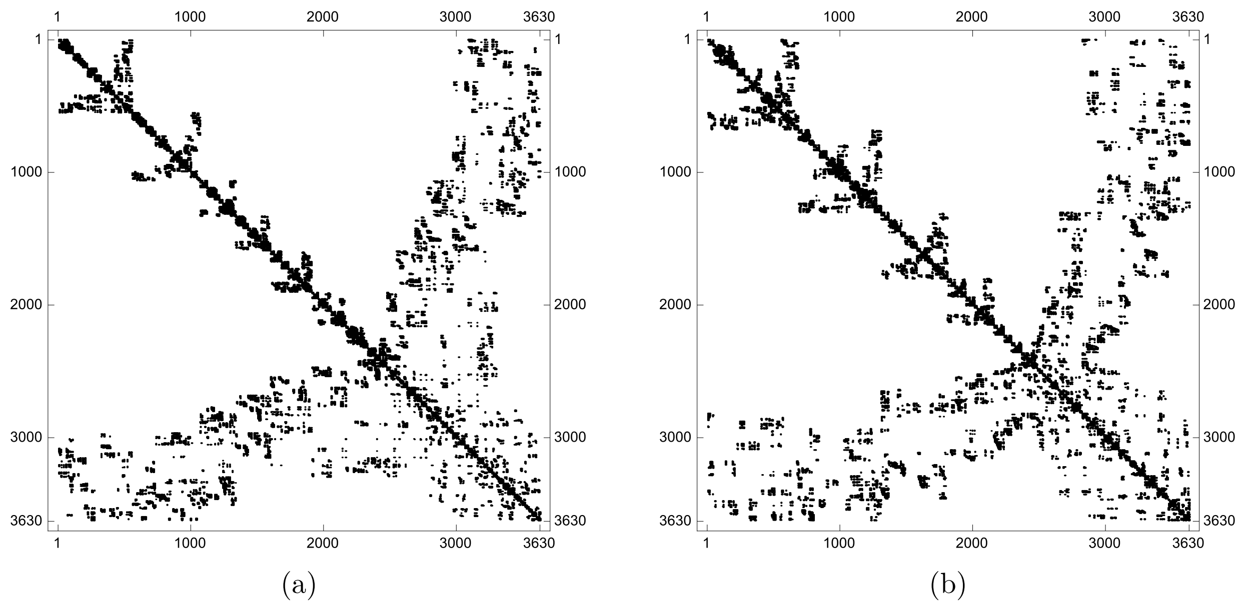
5. **mtMETIS** - metoda bazująca na zagnieżdzonym podziale grafu, wykorzystująca architekturę wielowątkową (METIS jest sekwencyjny).

Na rys. 4.11 przedstawiono struktura macierzy dla testu kostki o $N = 10$. Widać na nim trzy pasma wzdłuż diagonali. Rysunki 4.12a, 4.12b, 4.13a i 4.13b przedstawiają macierz po zastosowaniu poszczególnych metod permutacji macierzy. Można zauważyć, że każda z metod stara się stworzyć macierz „strzałkową” podczas swojego działania.

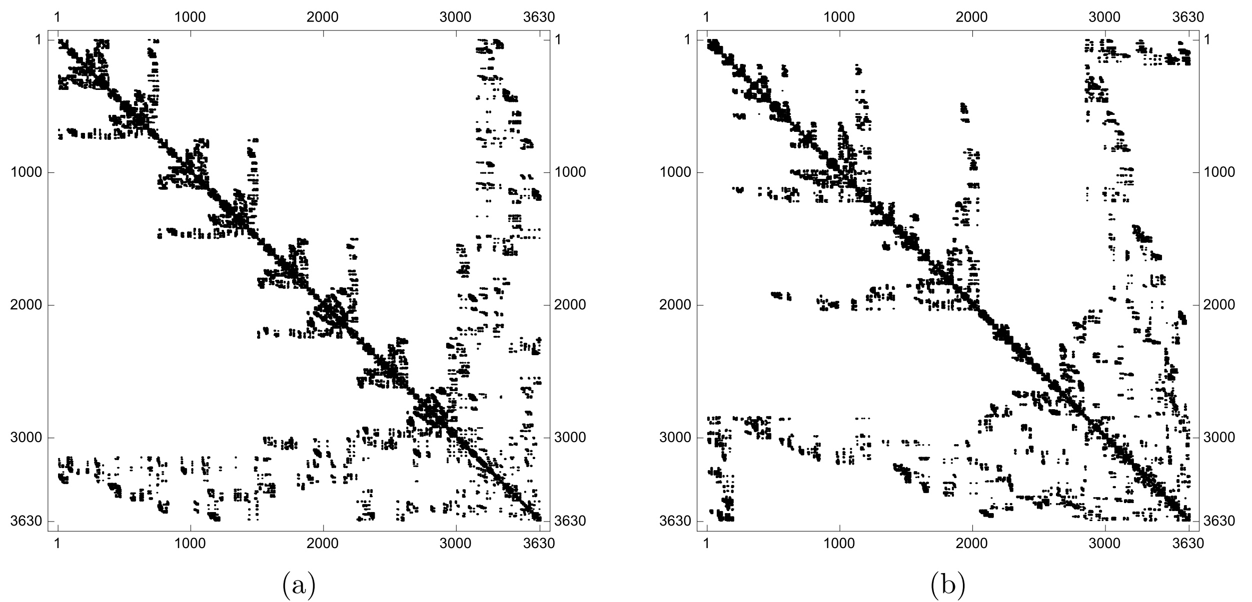
Na rys. 4.14 pokazano strukturę macierzy z rys. 4.11 po zastosowaniu **LU** dekompozycji. Z kolei rys. 4.15a, 4.15b, 4.16a i 4.16b, przedstawiają strukturę macierzy z rys. 4.12a, 4.12b, 4.13a i 4.13b po **LU** dekompozycji. Bez permutacji liczba elementów niezerowych to 2 609 288. Największe wypełnienie daje metoda MD, ok. 2 mln elementów niezerowych, z kolei najlepszy METIS to 1 647 935 elementów niezerowych, czyli o ok. 37 % mniej niż uzyskano dla macierzy niespermutowanej.



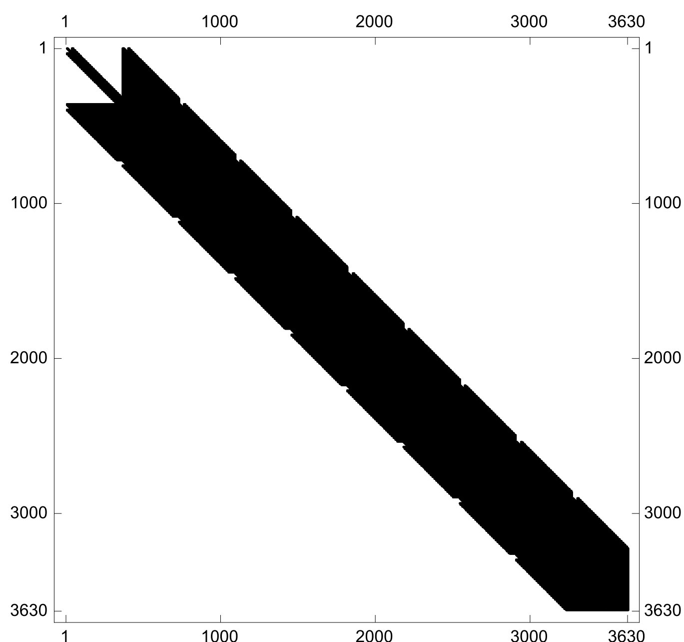
Rysunek 4.11: Struktura macierzy (bez stosowania permutacji). Test kostki $N = 10$.



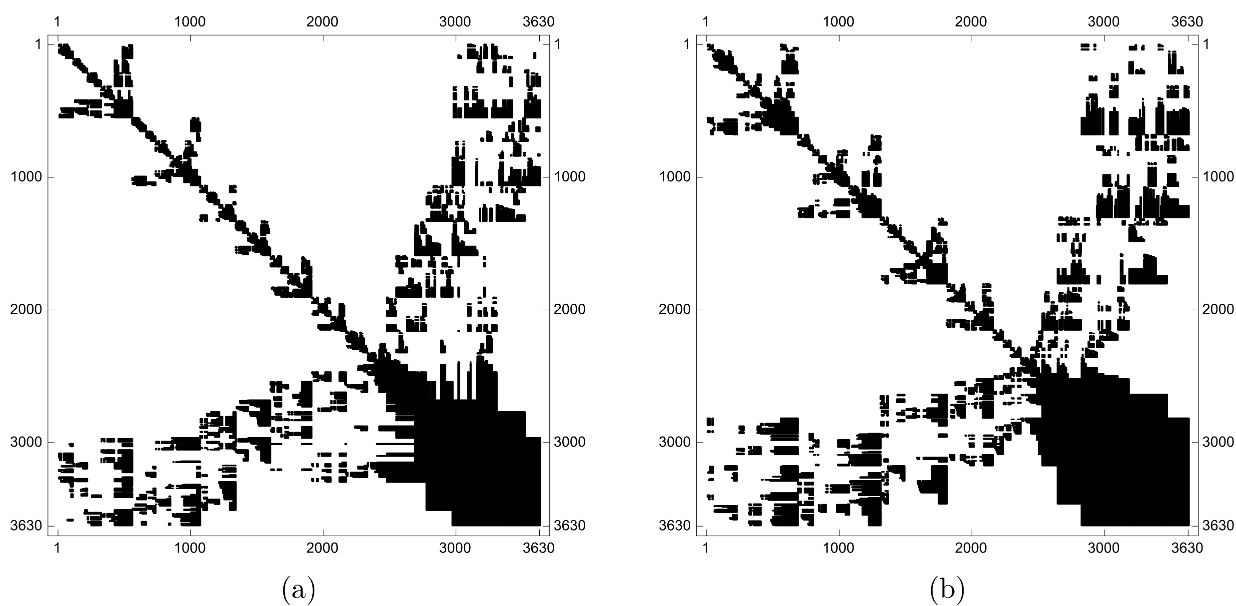
Rysunek 4.12: Struktura macierzy po zastosowaniu algorytmu permutacji macierzy: (a) cMark, i (b) MD. Test kostki $N = 10$.



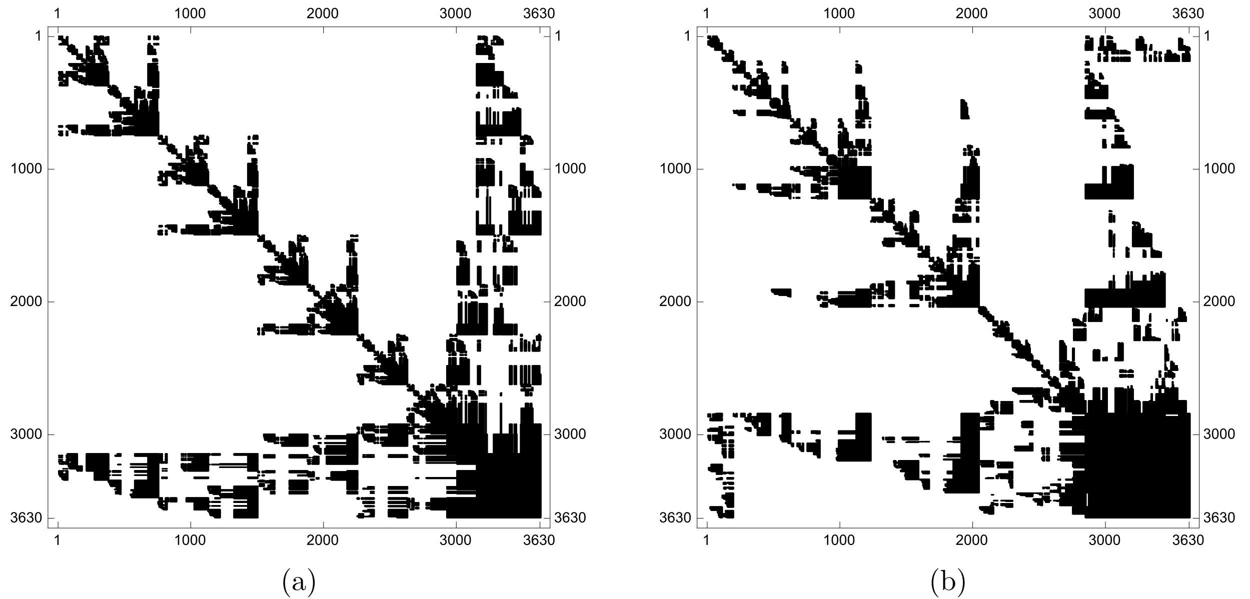
Rysunek 4.13: Struktura macierzy po zastosowaniu algorytmu permutacji macierzy: (a) METIS, i (b) AMD. Test kostki $N = 10$.



Rysunek 4.14: Struktura macierzy (bez stosowania permutacji) po **LU** dekompozycji. Test kostki $N = 10$.



Rysunek 4.15: Struktura macierzy po zastosowaniu algorytmu permutacji macierzy: (a) cMark, i (b) MD i po **LU** dekompozycji. Test kostki $N = 10$.



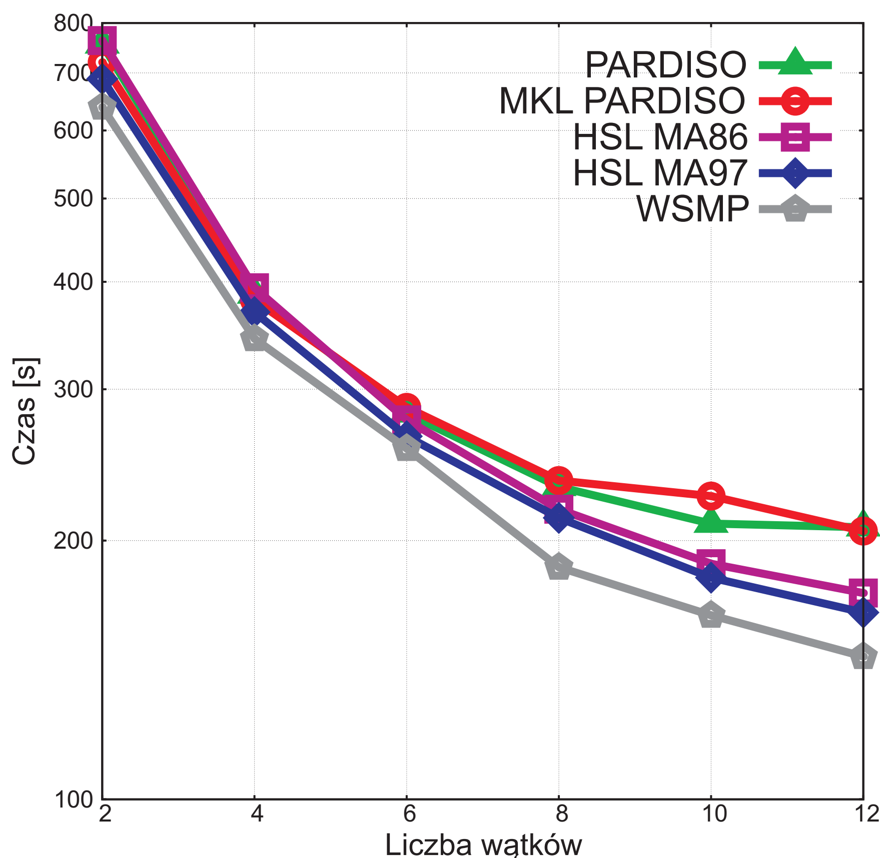
Rysunek 4.16: Struktura macierzy po zastosowaniu algorytmu permutacji macierzy: (a) METIS, i (b) AMD i po **LU** dekompozycji. Test kostki $N = 10$.

Tabela 4.3: Porównanie metod przenumerowania elementów w macierzy. Solwer HSL MA86. Test kostki dla $N = 32$.

Liczba wątków	Czas [sekundy]	cMark	MD	AMD	METIS	mtMETIS
1	permutacji	0.20	0.19	0.10	1.12	1.12
	faktoryzacji	133.93	106.32	69.38	25.83	25.83
	łączny	133.13	106.51	69.48	26.95	26.95
12	permutacji	0.20	0.20	0.10	1.12	0.72
	faktoryzacji	17.20	13.94	9.00	3.49	4.05
	łączny	17.40	14.13	9.10	4.61	4.77
Przyspieszenie						
	faktoryzacji	7.73	7.63	7.71	7.40	6.02
	łącznie	7.65	7.54	7.63	5.85	5.65

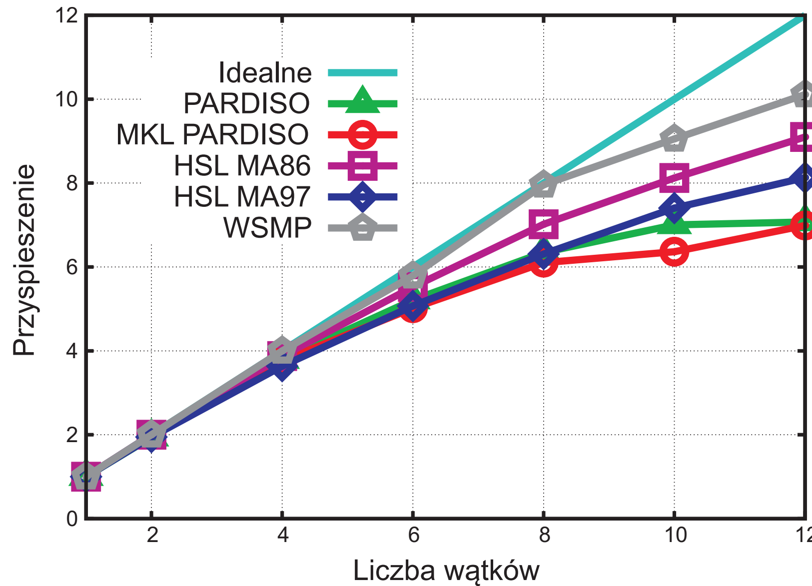
W tab. 4.3 porównano efekty metod przenumerowania elementów macierzy. Widać z niej, że metoda stosowana w pakiecie METIS daje jakościowo najlepsze przenumerowania, pod względem czasu faktoryzacji oraz czasu łącznego faktoryzacji i permutacji. Jest ona ponad 2 razy szybsza od kolejnej. Widać również, że zostało to osiągnięte pewnym kosztem - czas wykonania permutacji jest prawie 6 razy dłuższy niż dla najszybszej AMD. Jednak jakość wykonanego przenumerowania zdecydowanie rekompensuje czas, który został poświęcony na jego wykonanie. Dlatego w dalszych obliczeniach będzie stosowana biblioteka METIS do przenumerowywania wierszy w macierzy. Nie stosowano metody mtMETIS, ponieważ jest to biblioteka eksperymentalna i nie ma oficjalnej wersji. Została ona zamieszczona tylko dla porównania, dodatkowe wyniki można znaleźć w rozdz. 6.

4.2.5 Rezultaty testów



Rysunek 4.17: Czas wykonania fazy faktoryzacji przez różne solwery. Test kostki $N = 64$.

Na rys. 4.17 przedstawiono czas wykonania fazy faktoryzacji przez różne solwery przy zmianie liczby wątków. Dla 12 wątków najszybszy jest solwer WSMP, w porównaniu z drugim HSL MA97 jest szybszy o ok. 11%. Warto zauważyć, że solwery korzystające z metody macierzy frontalnych (WSMP, HSL MA97) są szybsze od pozostałych, co będzie miało wpływ na wykorzystanie pamięci (por. z tab. 4.4). Solwer HSL MA86 jest szybszy o ok. 15% od obu solwerów PARDISO. Oba solery PARDISO zachowują się bardzo podobnie z niewielką przewagą solwera MKL PARDISO. Solwer PARDISO jest szybszy od



Rysunek 4.18: Skalowalność etapu faktoryzacji. Test kostki $N = 64$.

solwera MKL PARDISO do 10 wątków, później jest nieznacznie wolniejszy o ok. 1%. Potwierdza to słowa autorów, że solwer PARDISO ma zaimplementowane pewne ulepszone algorytmy, ale widać też, że testowany był przez nich tylko do 8 wątków, dla którego deklarują przyspieszenie rzędu „do” 7 i takie jest osiągane, patrz rys. 4.18. Jednak przy wykorzystaniu pełnej liczby wątków mimo, że przyspieszenie mają lepsze, więc algorytmy zrównoleglające również, nie są w stanie wyprzedzić wspomaganego sprzętowo MKL PARDISO, różnice się zacierają pod względem czasu. Autorzy solwera HSL MA86 deklarują przyspieszenie „ponad” 6 dla 8 wątków i tutaj również osiągnięto podobny wynik rzędu 6.9 dla 8 wątków. Z rys. 4.18 widać, że HSL MA86 wyprzedza pozostałe implementacje, ma zdecydowanie lepsze przyspieszenie od pozostałych (z wyjątkiem WSMP) rzędu 8.5 razy dla 12 wątków.

Tabela 4.4: Wykorzystanie pamięci. Test kostki dla $N = 64$.

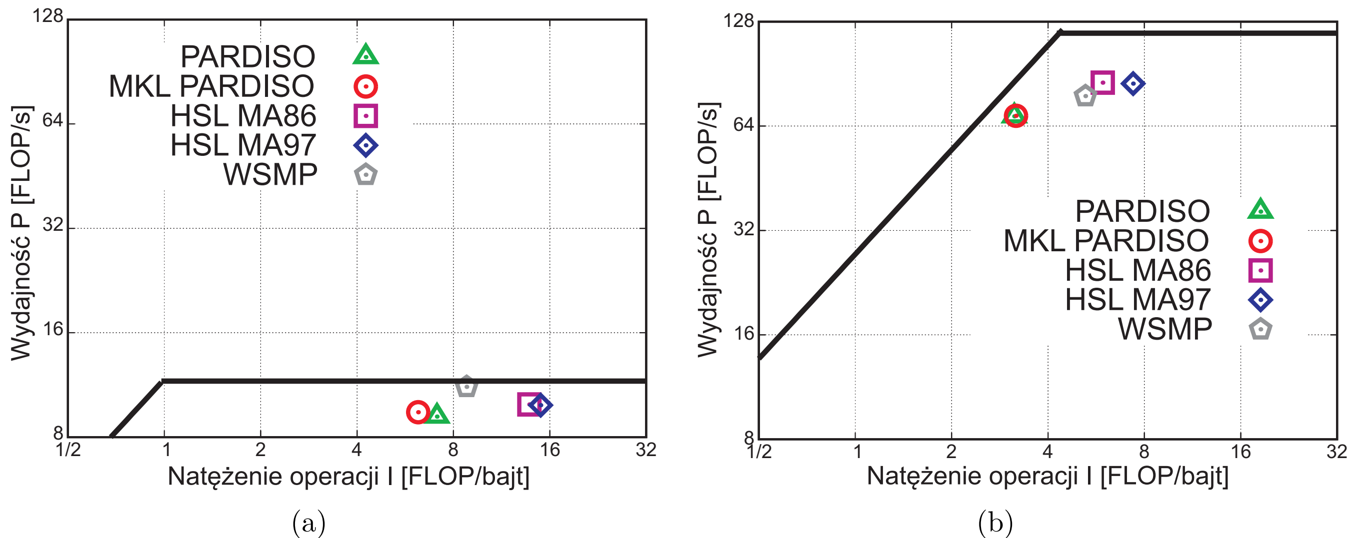
Solwer	Wykorzystanie pamięci [GB]
PARDISO	15.14
Intel MKL PARDISO	14.97
HSL MA86	15.45
HSL MA97	18.75
WSMP	20.47

W tab. 4.4 podano wykorzystanie pamięci przez poszczególne solwery. Widać, że najlepsza prędkość solwera WSMP wiąże się z największym wykorzystaniem pamięci. Można także zauważyć, że solwery korzystające z metody macierzy frontalnych (WSMP, HSL MA97) wykorzystują zdecydowanie więcej pamięci niż pozostałe solwery (oba PARDISO i HSL MA86).

Na rys. 4.19 przedstawiono model „Roofline” dla procesora Intel Xeon 5670 z zazna-

czonymi wykonaniami faktoryzcji dla poszczególnych solwerów, z wykonaniem sekwencyjnym (rys. 4.19a) oraz równoległym (rys. 4.19b).

Można zaobserwować, że dla wykonania sekwencyjnego faza faktoryzacji dla wszystkich solwerów jest ograniczona obliczeniowo a dla wykonania równoległego solwery PARDISO są ograniczone pamięciowo. Widać również, że wydajność solwerów jest bliska wydajności maksymalnej.



(a) (b)

Rysunek 4.19: Wykres „Roofline” dla fazy faktoryzacji: (a) dla 1 wątka i (b) dla 12 wątków.

Przetestowane solwery zdecydowanie przyspieszają rozwiązywanie układu równań liniowych, niestety faza faktoryzacji nie skaluje się idealnie (patrz rys. 4.18), a przedstawiona wydajność na modelu „Roofline” z rys.4.19b wskazuje, że wynika to z ograniczenia pamięciowego zastosowanych algorytmów. W następnym rozdziale zostaną opisane dokładniej techniki wykorzystywane w wybranym solwerze oraz przedstawione zostaną metody przyspieszenia obliczeń.

4.3 Paralelizacja w solwerze HSL MA86

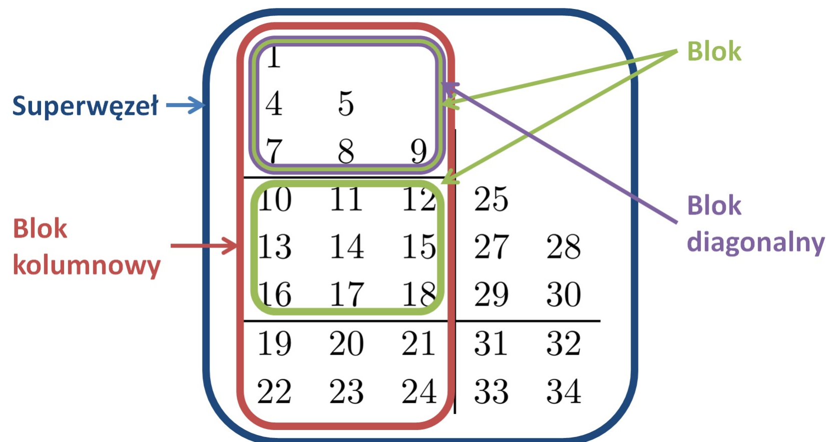
W tym rozdziale badano dlaczego faza faktoryzacji nie skaluje się odpowiednio w żadnym z przetestowanych solwerów, patrz rys. 4.18. Do badań wybrano solwer HSL MA86 ze względu na to, że jego kod źródłowy jest dostępny oraz wykazuje on większą skalowalność niż solwer HSL MA97. Na początku tego rozdziału przedstawiony zostanie sposób działania solwera HSL MA86, później opisany będzie czas i skalowalność poszczególnych kroków algorytmu. Na koniec wskazany będzie sposób przyspieszenia obliczeń, a zarazem poprawy skalowalności.

4.3.1 Sposób działania

Po początkowej fazie przenumerowania i analizy powstaje drzewo zadań, które musi wykonać solwer podczas fazy faktoryzacji. Zadania w fazie analizy zostały połączone

w superwęzły (patrz rozdz. 4.1.6), tak aby odpowiednie działania w fazie faktoryzacji odbywały się na macierzach gęstych.

W solwerze HSL MA86 superwęzeł to pewien ciągły zbiór kolumn macierzy, który trzeba sfaktoryzować. Każdy superwęzeł składa się z wielu bloków kolumnowych (to one podlegają faktoryzacji), a one z kolei złożone są z bloków o określonej wielkości (są one macierzami) oraz bloków diagonalnych (są one macierzami trójkątnymi dolnymi). Podział na bloki jest konieczny, aby maksymalnie wykorzystać możliwości biblioteki BLAS, gdy rozmiar samego superwęzła jest duży. Superwęzeł z podziałem na bloki kolumnowe o rozmiarze 3 został przedstawiony na rys. 4.20.



Rysunek 4.20: Superwęzeł i jego podział na bloki: zwykłe, kolumnowe i diagonalne.

Faza faktoryzacji macierzy wykonuje trzy rodzaje zadań:

Z1. faktoryzacja superwęzła,

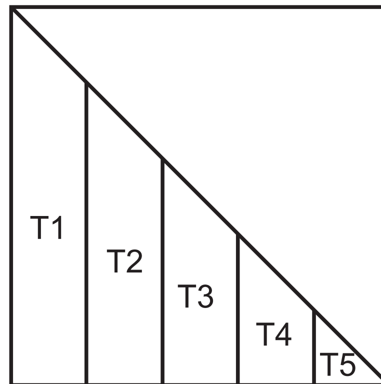
Z2. aktualizacja danych w superwęźle,

Z3. aktualizacja danych między superwęzłami.

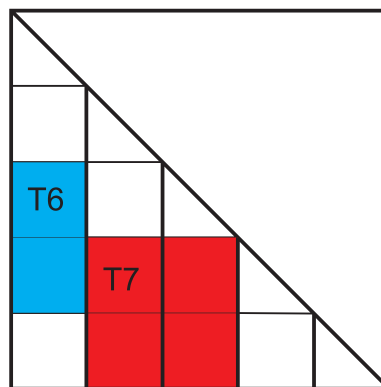
Dodatkowo istnieje jeszcze jeden typ zadania oznaczający koniec obliczeń. Zadanie faktoryzacji superwęzła sprowadza się do sfaktoryzowania bloku kolumnowego, a później dodania do wspólnej puli zadań, nowych zadań aktualizacji danych w superwęźle oraz między superwęzłami. Aktualizacja danych w superwęźle oraz między węzłami odpowiada aktualizacji odpowiednich bloków w superwęźle.

Początkowo wszystkie zadania to zadania faktoryzacji, które są stworzone z bloku kolumnowego, zostało to przedstawione na rys. 4.21, gdzie zadania T1, ..., T5 to zadania faktoryzacji, a równocześnie bloki kolumnowe o podobnym rozmiarze. Po wykonaniu zadania faktoryzacji może powstać potrzeba aktualizacji danych w superwęźle lub między superwęzłami, wtedy tworzone są odpowiednie zadania aktualizacji w superwęźle lub między superwęzłami, patrz rys. 4.22.

Zadania początkowe faktoryzacji, które są liśćmi w drzewie zadań, są dodawane do wspólnej puli zadań dla wszystkich wątków (oprócz wspólnej puli zadań, każdy wątek posiada własną, prywatną pulę zadań). Następnie rozpoczyna się wykonanie równoległe



Rysunek 4.21: Zadania faktoryzacji superwęzła na początku fazy faktoryzacji, bloki kolumnowe tworzą jedno zadanie.



Rysunek 4.22: Zadania aktualizacji w superwęźle (kolor niebieski) i między superwęzłami (kolor czerwony), zbiór bloków tworzy zadanie.

algorytmu, w którym każdy wątek wykonuje następujące czynności:

1. Pobranie zadania z prywatnej dla wątku puli zadań.
2. Jeśli w puli zadań wątku nie było zadań, pobranie zadania ze wspólnej puli (Z0).
3. Wykonanie odpowiedniego zadania (Z1 lub Z2 lub Z3).
4. Sprawdzenie, czy można dodać zadanie do puli zadań.

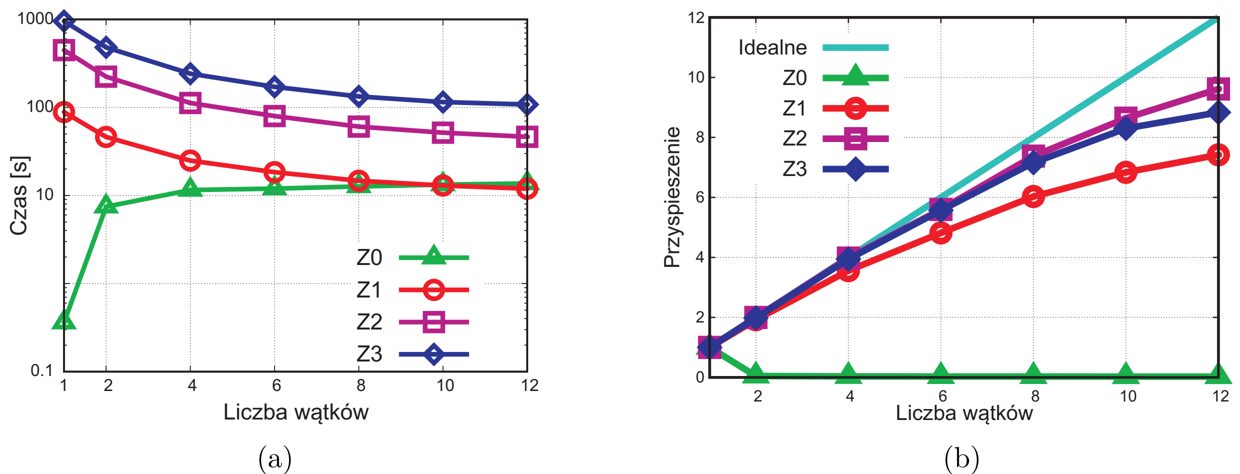
Wątek powtarza te czynności w nieskończonej pętli, aż pobrane zadanie będzie miało typ końca obliczeń. Punkt 1 nie wymaga żadnych działań synchronizacyjnych między wątkami, gdyż każdy wątek posiada swoją prywatną pulę zadań. Natomiast punkt 2 już tego wymaga. Operacje wykonywane w punkcie 2 oznaczono jako zadanie Z0.

Solwer HSL MA86 realizuje wspomnianą synchronizację w następujący sposób. Jeżeli dany wątek nie ma zadań, to blokuje wspólną pulę zadań i pobiera z niej połowę dostępnych zadań, tak aby nie musiał korzystać ze wspólnej puli zbyt często. W następnym podrozdziale zostanie pokazane, że ma to bardzo duży wpływ na czas wykonania tego

etapu. Punkt 4 polega na sprawdzeniu w drzewie zadań, czy dla rodzica zadania, które aktualnie zostało wykonane, wszystkie zadania dzieci zostały już wykonane, tzn. czy wszystkie potrzebne dane do wykonania zadania zostały już przygotowane. Dodanie zadania do puli zadań polega najpierw na sprawdzeniu, czy liczba zadań w puli prywatnej dla wątku nie przekracza maksymalnej; jeśli nie przekracza to zadanie zostaje dodane do puli wątku. Jeśli jednak maksymalna liczba zadań została przekroczona, wtedy zadanie zostanie dodane do wspólnej puli zadań. Domyślna maksymalna liczba zadań dla puli prywatnej wątku dla solwera HSL MA86 to 100.

4.3.2 Czas wykonania i skalowalność zadań w fazie faktoryzacji

Rysunek 4.23a przestawia czas sumaryczny dla poszczególnych zadań przy czym dla każdego zadania wybrano najdłuższy czas wykonania spośród wszystkich wątków. Widać, że zadania związane tylko z samą faktoryzacją (Z1-Z3) korzystają na wzroście liczby wątków. Natomiast zadanie Z0, polegające na pobraniu zadania z puli zadań wręcz przeciwnie, czas wzrasta ponad dziesięciokrotnie.



Rysunek 4.23: Czas sumaryczny (a) i przyspieszenie (b) poszczególnych zadań w procesie faktoryzacji. Solwer HSL MA86. Test kostki $N = 64$.

Podobnie sytuacja wygląda na rys. 4.23b. Zadania Z1-Z3 nie skalują się idealnie, ale przyspieszenie wzrasta wraz z liczbą wątków. Natomiast skalowalność zadania Z0 jest na poziomie 0, tzn. czas wykonania tego zadania bardzo szybko rośnie a nie spada wraz ze wzrostem liczby wątków. Ten wynik jest zrozumiały, ponieważ pobranie zadania, jeśli nie następuje z puli własnej wątku, musi być zsynchronizowane z pozostałymi wątkami.

Dodatkowo z tab. 4.5 widać, że czas trwania poszczególnych zadań jest bardzo krótki, w porównaniu z liczbą zadań do wykonania, więc pobranie zadania występuje bardzo często.

4.3.3 Przyspieszenie obliczeń i poprawa skalowalności

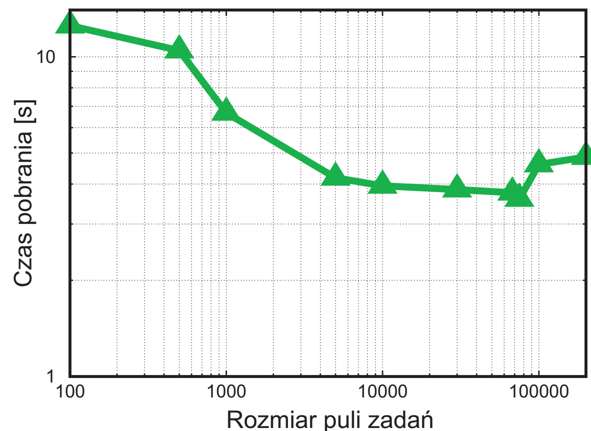
Poniższy pomysł na przyspieszenie obliczeń wynika z analizy wykonanej w poprzednim podrozdziale. Skoro synchronizacja podczas wydobywania zadania trwa tak długo, to

Tabela 4.5: Liczba różnych typów zadań i średni czas wykonania pojedynczego zadania. Test kostki $N = 64$.

Typ zadania	Liczba zadań	Czas wykonania jednego zadania [ms]
Z1	11098	1.08
Z2	152378	0.30
Z3	761065	0.14

trzeba ją robić jak najrzadziej, pamiętając, że synchronizacja jest potrzebna przy pobraniu zadania ze wspólnej puli zadań, a nie z puli prywatnej dla wątku.

Przy liczbie zadań ok. 900000, domyślna maksymalna liczba zadań w puli dla wątku jest za mała (tylko 100), więc wątek musi często odwoływać się do puli wspólnej po zadania. Dlatego wykonano testy, dla różnej maksymalnej liczby zadań w puli. Wyniki przedstawia rys. 4.24 widać z niego, że najmniejszy sumaryczny czas wykonania pobrania zadania (Z0) przez wątek jest wtedy, gdy maksymalna liczba zadań w puli własnej wątku zostanie ustawiona na 75 000.



Rysunek 4.24: Maksymalny sumaryczny czas pobrania zadania dla 12 wątków. Test kostki $N = 64$.

W tym przypadku liczba wszystkich zadań wynosi 924 541 i jeśli podzieli się tę liczbę przez liczbę wątków (12) to otrzyma się ok. 77 045. Liczba zadań nie jest znana przed rozpoczęciem fazy faktoryzacji, ale można posłużyć się liczbą równań do jej przybliżenia.

Dla kostki $N = 64$ liczba równań to 811 200 i po podzieleniu jej na 12 wątków otrzymano 67 600. Można przedstawić to wzorem $M = \frac{N_{eq}}{N_{tr}}$, gdzie M to maksymalna liczba zadań w puli własnej wątku, N_{eq} to liczba równań, a N_{tr} to liczba wątków.

W tab. 4.6 porównano czas i przyśpieszenie dla $M = 100$ i obliczonego wg zaproponowanego wzoru. Widać, że dzięki zastosowanej zmianie liczby zadań można przyspieszyć solwer o ok. 5%. Zaobserwowano także nieznaczny wzrost zapotrzebowania na pamięć o ok. 1%.

Tabela 4.6: Poprawa czasu faktoryzacji i przyspieszenia, dzięki zastosowaniu większej liczby maksymalnej zadań w puli wątku. 12 wątków. Test kostki $N = 64$.

M=100		$M = \frac{N_{eq}}{N_{tr}} = 67600$		Poprawa [%]	
Czas [s]	Przyspieszenie	Czas [s]	Przyspieszenie	czasu	przyspieszenia
174.29	8.05	165.66	8.47	4.95	5.21

4.3.4 Dostosowanie parametrów uporządkowania

W czasie badania kodu źródłowego solwera HSL MA86 okazało się, że podczas etapu analizy wykonuje on pewną optymalizację. Optymalizacja polega na przenumerowaniu wierszy macierzy w każdym superwęźle, w taki sposób aby zmaksymalizować lokalność pamięci podręcznej w czasie faktoryzacji. Wyłączenie tej opcji powoduje skrócenie czasu faktoryzacji o 3% z 174.29s do 169.65s.

Wykonano również analizę parametrów algorytmu METIS. Jednym z parametrów jest liczba wybieranych separatorów. Algorytm METIS może wybrać na każdym kroku określoną tym parametrem liczbę kandydatów na separatora i ostatecznie wybrać najlepszego z nich. Autorzy METIS zaznaczają, że sensowne wartości to od 1 do 5 separatorów, przy czym w przypadku 5, ich zdaniem, czas uporządkowania może wzrosnąć 3 razy. Czasy uporządkowania, czas faktoryzacji i wykorzystanie pamięci w zależności od liczby wybieranych separatorów podano w tab. 4.7.

Tabela 4.7: Porównanie czasu uporządkowania, faktoryzacji i pamięci w zależności od liczby separatorów w algorytmie METIS. 12 wątków. Test kostki $N = 64$.

Liczba separatorów	Czas uporządkowania [s]	Czas faktoryzacji [s]	Pamięć [GB]
1	5.75	174.29	15.93
2	5.78	175.41	16.03
3	7.79	174.82	15.98
4	**9.83**	**161.18**	**15.69**
5	11.98	161.98	15.71

Widać, że największe korzyści osiąga sie przy wyborze 4 separatorów. Uzyskano prawie dwukrotny wzrost czasu wykonania przenumerowania, ale jest on rekompensowany poprzez zmniejszony czas faktoryzacji oraz mniejsze wykorzystanie pamięci.

4.3.5 Dyskusja innych sposobów przyspieszenia

Z rys. 4.23b wynika, że istnieje jeszcze pewne pole do poprawy skalowalności faktoryzacji i zmniejszenia czasu całego procesu. Przebadano trzy typy zadań (Z1-Z3), aby sprawdzić, czy nie posiadają punktów synchronizacyjnych, które powodowałyby utratę skalowalności. Niestety takich punktów nie znaleziono. Natomiast zauważono, że występujące działania na macierzach gęstych wykonują się dłużej, gdy uruchomi się większą liczbę wątków, co powoduje spadek skalowalności. Wydaje się, że może być to spowodowane zbliżeniem się do maksymalnej przepustowości pamięci (ang. *memory bandwith*),

tzn. osiągnięto sprzętową barierę dalszych optymalizacji, ale nie na poziomie procesora lecz na poziomie pamięci, co zaobserwowano w modelu „Roofline" w rozdz. 4.2.5.

Przebadano również, czy występują efekty związane z fałszywym współdzieleniem (ang. *false sharing*). Zastosowano wszystkie techniki eliminujące fałszywe współdzielenie proponowane przez twórców kompilatora Intel. Między innymi:

1. inicjalizacja danych przez wątek,
2. alokacja tablic przez wątek,
3. minimalizacja wspólnych tablic.

Niestety nie uzyskano poprawy czasów wykonania.

Warto również zauważyć, że faktoryzacja zaczyna wyraźnie zmniejszać swoją skalowaność przy 6 wątkach, patrz rys. 4.18. Z uwagi na to, że komputer wykorzystywany do testów posiada dwa 6-rdzeniowe procesory, zbadano czy gdyby podzielić zadanie faktoryzacji na dwa mniejsze, to uzyskano by lepsze przyspieszenie, tzn. czy każdy procesor pracowałby ze skalowalnością dla 6 wątków, co mogłoby dać wyższą skalowalności niż dla 12 wątków. Z wykorzystaniem interfejsu KMP_AFFINITY (patrz rozdz. 2.6.4), obliczano test kostki dla $N = 64$ i dla $N = 50$, tzn. dla o połowę mniejszego zadania (o połowę mniej niewiadomych). Interfejs KMP_AFFINITY służył do uruchomienia: (a) 6 wątków tylko na jednym procesorze (6×1) za pomocą parametru COMPACT, lub (b) 3 na jednym i 3 na drugim (3×2) za pomocą parametru SCATTER; ten parametr był wykorzystany we wszystkich testach do tej pory.

Z tab. 4.8 widać, że wariant (b) jest zdecydowanie lepszy niż (a). Oznacza to, że gdyby stworzono solwer, który dzieliłby zadanie na połowę i przydzielił każdemu procesorowi jedną część i ta część byłaby sfaktoryzowana przez solwer HSL MA86, to skalowalność tej operacji, byłaby mniejsza niż skalowalność dla 12 wątków na 2 procesorach, patrz rys. 4.18. Dlatego zaniechano tej próby.

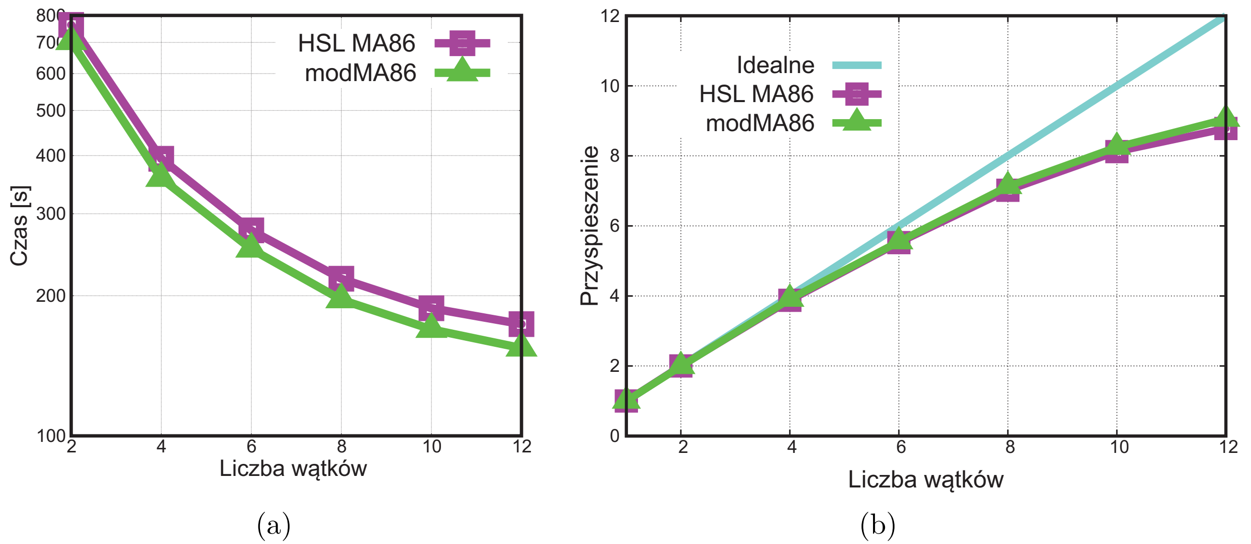
Tabela 4.8: Przyspieszenie faktoryzacji przy różnych koligacjach wątków. Solwer HSL MA86. Test kostki.

N	Przyspieszenie		
	6×1	3×2	12
50	4.62	5.49	8.91
64	4.62	5.59	8.66

4.3.6 Podsumowanie przyspieszenia solwera HSL MA86

W rozdz. 4.3.3 i rozdz. 4.3.4 zaproponowano trzy poprawki do solwera HSL MA86:

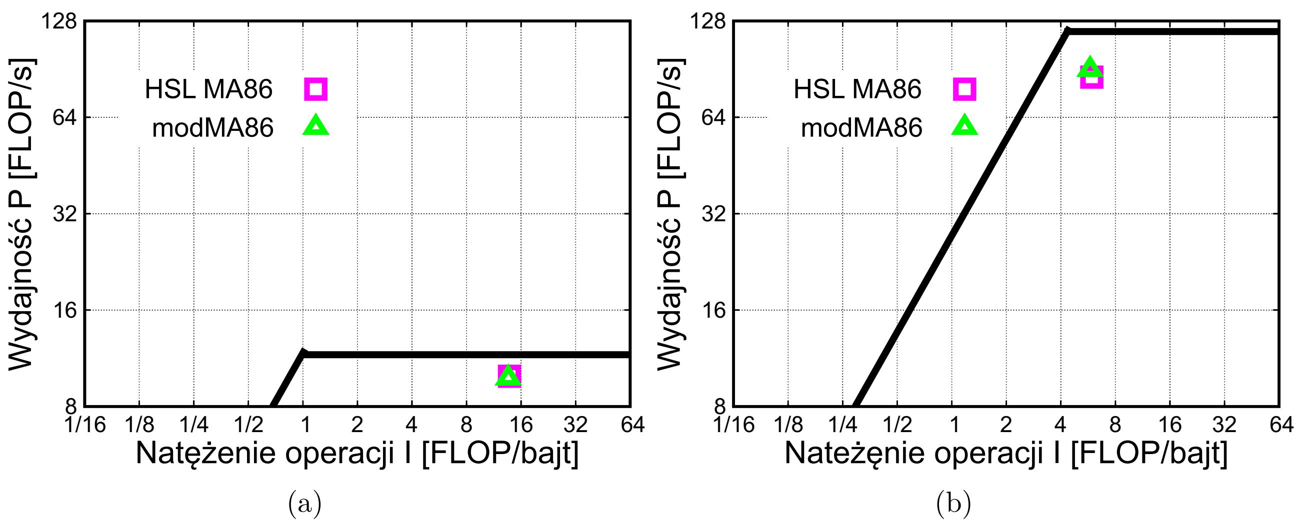
1. dynamiczne określenie wielkości prywatnej puli zadań dla wątku,
2. wyłączenie opcji maksymalizacji lokalności pamięci podręcznej w fazie analizy,
3. poszukiwanie optymalnej liczby separatorów.



Rysunek 4.25: Czas wykonania (a) i przyspieszenie (b) faktoryzacji. Test kostki $N = 64$.

Rysunek 4.25 przedstawia czasy wykonania i skalowalność faktoryzacji dla solwera HSL MA86 oraz solwera modMA86 z powyższymi poprawkami. Spowodowały one przyspieszenie solwera HSL MA86 o 11.5%, z 174.29s do 154.37s jeśli chodzi o czas faktoryzacji. Poprawiła się również skalowalność solwera o 7% z 8.5 do 9.1 razy.

Na rys. 4.26 przedstawiono model „Roofline" dla procesora Intel Xeon 5670 z zaznaczonymi wykonaniami faktoryzcji dla solwera HSL MA86 oraz dla solwera z poprawkami modMA86, dla wykonania sekwencyjnego (rys. 4.26a) oraz równoległego (rys. 4.26b). Można zaobserwować, że dla wykonania sekwencyjnego nie ma wzrostu wydajności, ale dla wykonania równoległego solwer z zaproponowanymi poprawkami (modMA86) ma wydajność większą o 5%.



Rysunek 4.26: Wykres „Roofline" dla fazy faktoryzacji: (a) dla 1 wątka i (b) dla 12 wątków.

4.4 Solwer z mieszaną precyzją

Do tej pory w tym rozdziale przedstawiono wydajność solwerów wykorzystujących obliczenia z podwójną precyzją. W tym podrozdziale przedstawiony zostanie modMA86mix, który wykorzystuje pojedynczą precyzję dla najbardziej czasochłonnych operacji, ale za pomocą metody iteracyjnego poprawiania zostanie uzyskany wynik w podwójnej precyzji. W połączeniu z wynikami z rozdz. 4.3 uzyskano o wiele szybszy i bardziej skalowalny solwer do rozwiązywania układów równań liniowych.

4.4.1 Iteracyjne poprawianie rozwiązania

Iteracyjne poprawianie rozwiązania (ang. *iterative refinement*) po raz pierwszy zostało zaproponowane przez Wilkinsona [112] w 1963. W niniejszej pracy będzie przedstawione wyprowadzenie tej metody za tą pracą. Dla równania:

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{4.2}$$

zostanie skonstruowany ciąg wektorów $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ...$, który w pewnych warunkach jest zbieżny do prawdziwego rozwiązania $\mathbf{x}$. Pierwsze przybliżenie $\mathbf{x}^{(1)}$ zostanie otrzymane poprzez wykonanie faktoryzacji za pomocą solwera HSL MA86 w pojedynczej precyzji. W ten sposób powstaną macierze: trójkątna dolna $\mathbf{L}$ i trójkątna górna $\mathbf{U}$, takie dla których zachodzi równość:

$$\mathbf{L}\mathbf{U} = \mathbf{A} + \mathbf{E}, \tag{4.3}$$

gdzie macierz $\mathbf{E}$ to macierz błędów powstałych w wyniku błędów zaokrągleń przy wykonywaniu faktoryzacji. Niech wektory $\mathbf{r}^{(s)}$ i $\mathbf{x}^{(s)}$ będą określone za pomocą zależności:

$$\mathbf{r}^{(s)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(s)}, \quad \mathbf{x}^{(s+1)} = \mathbf{x}^{(s)} + (\mathbf{L}\mathbf{U})^{-1}\mathbf{r}^{(s)} \tag{4.4}$$

a więc $\mathbf{r}^{(s)}$ jest wektorem reszt odpowiadającym $\mathbf{x}^{(s)}$. Gdyby można było konstruować ciąg $\mathbf{x}^{(s)}$ bez błędów zaokrągleń, to jego wyrazy spełniałyby zależności

$$\mathbf{x}^{(s+1)} = \mathbf{x}^{(s)} + (\mathbf{L}\mathbf{U})^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}^{(s)}) = \mathbf{x}^{(s)} + (\mathbf{L}\mathbf{U})^{-1}(\mathbf{A}\mathbf{x} - \mathbf{A}\mathbf{x}^{(s)}). \tag{4.5}$$

Teraz można odjąć obustronnie wektor rozwiązania $\mathbf{x}$:

$$\mathbf{x}^{(s+1)} - \mathbf{x} = \mathbf{x}^{(s)} - \mathbf{x} + (\mathbf{L}\mathbf{U})^{-1}\mathbf{A}(\mathbf{x} - \mathbf{x}^{(s)}) = [\mathbf{I} - (\mathbf{L}\mathbf{U})^{-1}\mathbf{A}](\mathbf{x}^{(s)} - \mathbf{x}). \tag{4.6}$$

Wykonując takie podstawienie s razy uzyska się,

$$\mathbf{x}^{(s+1)} - \mathbf{x} = [\mathbf{I} - (\mathbf{L}\mathbf{U})^{-1}\mathbf{A}]^{s}(\mathbf{x}^{(1)} - \mathbf{x}). \tag{4.7}$$

Przemnażając lewostronnie przez $\mathbf{A}$, otrzymamy

$$\begin{aligned} \mathbf{r}^{(s+1)} &= \mathbf{A}(\mathbf{x} - \mathbf{x}^{(s+1)}) = \mathbf{A}[\mathbf{I} - (\mathbf{L}\mathbf{U})^{-1}\mathbf{A}]^{s}(\mathbf{x} - \mathbf{x}^{(1)}) \\ &= [\mathbf{A} - \mathbf{A}(\mathbf{L}\mathbf{U})^{-1}\mathbf{A}][\mathbf{I} - (\mathbf{L}\mathbf{U})^{-1}\mathbf{A}]^{s-1}(\mathbf{x} - \mathbf{x}^{(1)}) \\ &= [\mathbf{I} - \mathbf{A}(\mathbf{L}\mathbf{U})^{-1}]\mathbf{A}(\mathbf{x} - \mathbf{x}^{(s)}), \end{aligned} \tag{4.8}$$

$$\mathbf{r}^{(s+1)} = \left[\mathbf{I} - \mathbf{A}(\mathbf{L}\mathbf{U})^{-1}\right]\mathbf{r}^{(s)}. \tag{4.9}$$

Jeśli $\mathbf{E} = \mathbf{0}$, to w jednym kroku iteracyjnym $\mathbf{x}^{(s)}$ stanie się równe $\mathbf{x}$, zaś $\mathbf{r}^{(s)}$ będzie wektorem zerowym. Równania (4.7) oraz (4.8) pokazują, że zbieżność badanego ciągu jest pewna tylko wtedy, gdy zachodzi

$$[\mathbf{I} - (\mathbf{A} + \mathbf{E})^{-1}\mathbf{A}]^s \to \mathbf{0}, \quad s \to \infty. \tag{4.10}$$

W pracy [112] pokazano, że ciąg wektorów $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ jest zbieżny jeśli macierz $\mathbf{A}$ nie jest bardzo źle uwarunkowana. Widać również, że przedstawiony ciąg wektorów to nic innego, jak kolejne wektory metody Newtona dla funkcji $\mathbf{F}(\mathbf{x}) = \mathbf{b} - \mathbf{A}\mathbf{x}$. Metoda Newtona oblicza zera funkcji $\mathbf{F}(\mathbf{x})$ za pomocą następującej zależności:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - (\mathbf{F}'(\mathbf{x}_n))^{-1}\mathbf{F}(\mathbf{x}_n). \tag{4.11}$$

Procedura wykonania iteracyjnego poprawiania rozwiązania wygląda następująco:

1. Wykonaj faktoryzację macierzy $\mathbf{A}$, tj. $\mathbf{A} = \mathbf{LU}$,
2. Rozwiąż układ $\mathbf{LUx} = \mathbf{b}$, rozwiązanie będzie pierwszym przybliżeniem ciągu wektorów, tj. $\mathbf{x}^{(0)} = \mathbf{x}$,
3. Oblicz wektor $\mathbf{r}^{(s)}$, tj. $\mathbf{r}^{(s)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(s)}$,
4. Korzystając z faktoryzacji z punktu 1, rozwiąż równanie $\mathbf{LUx} = \mathbf{r}^{(s)}$,
5. Popraw rozwiązanie $\mathbf{x}^{(s+1)} = \mathbf{x}^{(s)} + \mathbf{x}$,
6. Sprawdź zbieżność, jeśli brak to idź do punktu 3, w przeciwnym wypadku zakończ.

Zbieżność jest badana za pomocą nierówności:

$$\beta \geqslant \frac{\|\mathbf{r}^{(s)}\|_\infty}{\|\mathbf{A}\|_\infty \|\mathbf{x}^{(s)}\|_\infty + \|\mathbf{b}\|_\infty}, \tag{4.12}$$

gdzie β to oczekiwana dokładność, a normy są zdefiniowane następująco:

$$\|\mathbf{A}\|_\infty = \max_i \sum_j |a_{ij}|, \qquad \|\mathbf{v}\|_\infty = \max_i |v_i|. \tag{4.13}$$

4.4.2 Iteracyjne poprawianie rozwiązania dla solwera z mieszaną precyzją

Mimo że, metoda z poprzedniego podrozdziału znana jest już od ponad 50 lat, to dopiero niedawno badano ją w kontekście nie tylko poprawy dokładności rozwiązania, ale również w kontekście wydajności całego procesu, tzn. o ile można skrócić czas wykonywania faktoryzacji używając pojedynczej precyzji dla kroków najdroższych czasowo, tzn. 1,2 i 4, a podwójnej precyzji dla pozostałych kroków. W pracy [57, s. 232] udowodniono, że takie postępowanie powoduje, że ciąg wektorów $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ będzie dążył do rozwiązania w podwójnej precyzji.

Praca [67] badała mieszaną precyzję dla macierzy gęstych zaimplementowaną w pakiecie LAPACK. Przebadano wiele typów procesorów m.in Opteron, Itanium, Cray, PowerPC oraz Cell. W późniejszej pracy tej samej grupy [11] zastosowano mieszaną precyzję dla

macierzy rzadkich. Iteracyjne poprawianie stosowano z sukcesem również łącząc zwykłe procesory z jednostkami graficznymi np. w [50].

Warto zauważyć, że w bibliotece HSL istnieje solwer do rozwiązywania równań liniowych z mieszaną precyzją - MA79. Do paralelizacji korzysta on z solwera MA77 (opisany w pracy [41]), który z kolei korzysta z metody multifrontalnej, a do rozwiązywania macierzy multifrontalnych korzysta z solwera MA64, który służy do rozwiązywania równań liniowych dla macierzy gęstych. Dopiero solwer MA64 korzysta z paralelizacji na wątkach, czyli występuje tutaj paralelizacją dopiero na poziomie macierzy multifrontalnych. Implementacja z niniejszej pracy wykorzystuje paralelizację na wyższym poziomie przy podziale superwęzłów między wątki, zastosowaną w MA86.

Iteracyjne poprawianie rozwiązania zostało zaimplementowane w niniejszej pracy w programie FEAP [25] używając do tego solwera modMA86; otrzymany solwer z mieszaną precyzją oznaczono jako modMA86mix.

Strategia rozwiązywania układu równań liniowych jest następująca: pojedyncza precyzja zostanie wykorzystana aby przeprowadzić faktoryzację, a podwójna precyzja aby iteracyjnie poprawić rozwiązanie. Jeśli nie osiągnie się wystarczającej dokładności, to rozwiązanie zostanie poprawione bardziej wymagającą metodą FGMRES (patrz dodatek F). Jeśli dalej nie otrzyma się wystarczającej dokładności, to rozwiązanie zostanie obliczone w podwójnej precyzji.

W pracy [3] pokazano, że jeśli iteracyjne poprawianie zawiedzie to FGMRES często prowadzi do rozwiązania i poradzi sobie lepiej niż zwykły GMRES. Z kolei w pracy [4] udowodniono zbieżność rozwiązania, gdy użyje się algorytmu FGMRES do odtworzenia rozwiązania w podwójnej precyzji z faktoryzacji w pojedynczej precyzji. To podstawowa motywacja do użycia metody FGMRES gdy zawiedzie iteracyjne poprawianie.

Algorytm rozwiązania w mieszanej precyzji wygląda następująco:

1. **Ustaw:** Oczekiwaną dokładność γ oraz $prec = single$
2. **Pętla:**
   (a) Faktoryzuj $\mathbf{A}$ z precyzją $prec$
   (b) Rozwiąż $\mathbf{Ax} = \mathbf{b}$ i oblicz β za pomocą równ. (4.12)
   (c) Jeśli $\beta \leqslant \gamma$ to kończ, wpp przeprowadź iteracyjne poprawianie.
   (d) Jeśli $\beta \leqslant \gamma$ to kończ, wpp przeprowadź procedurę FGMRES.
   (e) Jeśli $\beta \leqslant \gamma$ to kończ.
   (f) Jeśli $prec = single$ to $prec = double$ i powtórz pętlę, wpp zakończ z błędem.

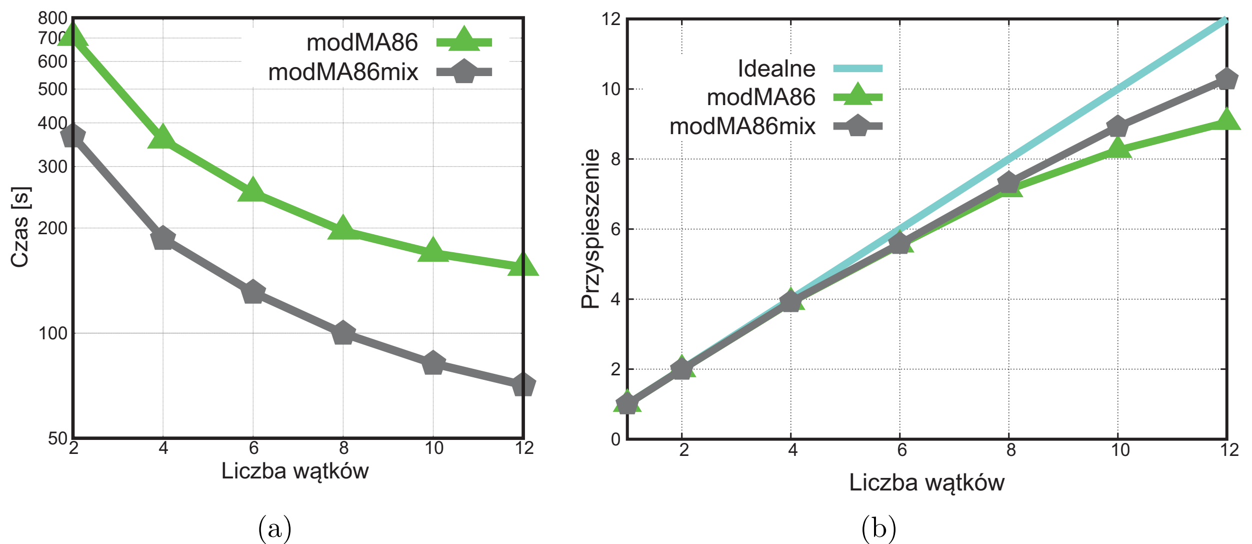
W tab. 4.9 porównano błędy i czasy wykonania dwóch metod: (1) iteracyjnego poprawiania i (2) FGMRES. Widać z niej, że FGMRES potrzebuje więcej czasu na dojście do odpowiedniego rozwiązania, dlatego metoda ta jest używana dopiero wtedy gdy zawiedzie metoda iteracyjnego poprawiania.

Wartości przemieszczeń otrzymane używając podwójnej precyzji (Double) i mieszanej precyzji z iteracyjnym poprawianiem rozwiązania (Mixed) były identyczne.

Tabela 4.9: Porównanie metod poprawiania rozwiązania. Test kostki $N = 64$.

Metoda	Liczba iteracji	Czas wykonania [s]	Błąd
Iteracyjne poprawianie	2	2.61	1.3E-16
FGMRES	4	5.01	3.2E-17

Wyniki wydajnościowe fazy faktoryzacji numerycznej zostały przedstawione w na rys. 4.27 i widać, że faktoryzacja w pojedynczej precyzji jest ok. 2.17 razy szybsza niż w podwójnej precyzji. Przyspieszenie w pojedynczej precyzji także jest większe i wynosi 10.28 dla 12 wątków. Warto zauważyć, że iloraz czasu obliczeń na 1 wątku w podwójnej precyzji do czasu obliczeń na 12 wątkach w pojedynczej precyzji dla faktoryzacji wynosi 19.68 na 12 rdzeniach.



(a) (b)

Rysunek 4.27: Czas wykonania (a) i przyspieszenie (b) faktoryzacji. Test kostki $N = 64$.

Czas wykonania samej fazy rozwiązania jest przedstawiony w tab. 4.10 i widać, że dla mieszanej precyzji wzrósł on o 54%, ale ponieważ wartość wzrostu to mniej nż 1s, więc jest ona rekompensowana skróconym czasem faktoryzacji numerycznej.

Dodatkowo warto zauważyć, że dzięki wykorzystaniu pojedynczej precyzji w fazie faktoryzacji, można obliczać większe zadania. Dla testu kostki $N = 64$, wykorzystanie pamięci wynosi 8.78GB, czyli jest zmniejszone o 43% w porównaniu do tej dla podwójnej precyzji, patrz tab. 4.4. Największym zadaniem, które można było przeliczyć na jednym węźle obliczeniowym, było zadanie dla $N = 78$, co daje ok. 1.5 mln niewiadomych, czyli prawie dwa razy więcej niż największe zadanie w podwójnej precyzji dla $N = 64$.

Tabela 4.10: Czas fazy rozwiązania (z iteracyjnym poprawianiem).

Liczba wątków	Czas [s]	
	Double	Mixed
12	1.70	2.61

5 Rozproszone rozwiązywanie układów równań liniowych

5.1 Wprowadzenie

Algorytmy na maszyny z pamięcią rozproszoną wykorzystują przesyłanie komunikatów i np. dla faktoryzacji **LU** na takich maszynach zaproponowano szereg różnych algorytmów [20]. W pracy [15] przedstawiono algorytm z częściowym wyborem elementu głównego oraz równoważeniem obciążenia za pomocą metod zorientowanych wierszowo z bezpośrednim równoległym algorytmem rozwiązywania macierzy trójkątnych. W późniejszej pracy [66] pokazano, że podobną wydajność można otrzymać za pomocą algorytmów zorientowanych kolumnowo.

Następnie szukano dobrego zrównoważenia między czasem obliczeń oraz czasem komunikacji. W pracy [93] pokazano, że metoda podziału kolumnowego jest wysoce efektywna. Blokowy podział macierzy został wprowadzony w [14] i spowodował znaczną redukcję komunikacji. Nowoczesne solwery na wiele węzłów takie jak WSMP [37], MUMPS [78], czy SuperLU_Dist [68], wykorzystują podział na dwuwymiarowe bloki, które mogą być ułożone w cykl. Taki podział zapewnia, że większość (jeśli nie wszystkie) procesory, mogą uczestniczyć w aktualizacji na każdym kroku blokowej eliminacji, a także zapewnia, że komunikacja między procesami jest ograniczona do zbiorów wierszy lub kolumn tych procesów [69].

Z analitycznego punktu widzenia jest bardzo trudno wyprowadzić wyrażenie na liczbę arytmetycznych operacji w trakcie fazy faktoryzacji lub wyrażenie na liczbę niezerowych elementów macierzy po faktoryzacji macierzy rzadkiej [35]. Dzieje się tak, ponieważ obliczenia oraz efekt wypełnienia (ang. *fill-in*) w trakcie faktoryzacji macierzy rzadkiej, to funkcja liczby oraz pozycji elementów niezerowych macierzy.

Algorytmy równoległe dla faktoryzacji Choleskiego zorientowane kolumnowo, tzn. takie gdzie przekazywane są kolumny faktoryzowanej macierzy, można podzielić na dwa rodzaje: (1) algorytmy *fan-out* - metoda, w której przekazywane między procesorami są tylko sfaktoryzowane kolumny, (2) algorytmy *fan-in* - metoda, w której przekazywane między procesorami są tylko kolumny, które trzeba zaktualizować. Prosty algorytm *fan-out* dla macierzy $N \times N$ na p procesorach ma objętość komunikacyjną rzędu $O(Np \log N)$. Można to poprawić za pomocą sprytnych mapowań między kolumnami a procesorami, co powoduje zmniejszenie objętości komunikacyjnej do $O(Np)$, patrz np. [30].

Z drugiej strony całkowity koszty obliczeń jest rzędu $O(N^{1.5})$, więc stosunek kosztu komunikacji do kosztu obliczeń w algorytmach bazujących na kolumnowym podziale jest stosunkowo wysoki. W konsekwencji prowadzi to do tego, że algorytmy te bardzo słabo się skalują, gdy wzrasta liczba procesów. W pracy [5] zaproponowano rodzinę algorytmów faktoryzacji Choleskiego, które posiadają łączną objętość komunikacji rzędu $\Theta(N\sqrt{p}\log N)$. Niestety, mimo że objętość komunikacji jest mniejsza od innych algorytmów zorientowanych kolumnowo, funkcja stałej wydajności (patrz rozdz. 2.2.2) jest rzędu $\Theta(p^3)$, ze względu na to, że algorytm ten nie jest w stanie efektywnie użyć więcej niż $O(\sqrt{N})$ procesów. Później zaproponowane algorytmy (np. [31], [61], [72]) ograniczyły objętość komunikacyjną do $O(N\sqrt{p}\log p)$.

Należy nie zapominać o tym, że o ile objętość komunikacyjną można wyznaczyć

analitycznie, to jest to tylko dolne ograniczenie całkowitego narzutu komunikacyjnego (ang. *communication overhead*). A właśnie całkowity narzut komunikacyjny ma istotny wpływ na osiąganą wydajność oraz przyspieszenie. W pracy [6] pokazano nawet, że algorytm *fan-in*, który ma mniejszą objętość komunikacyjną niż rozproszony algorytm multifrontalny, posiada większy narzut komunikacyjny i dlatego ma mniejszą wydajność.

Dlatego najbardziej popularne rozproszone solwery korzystają z metod multifrontalnych. W pracy [36] pokazano, że WSMP radzi sobie lepiej od MUMPS oraz SuperLU_Dist, w związku z tym w dalszej części niniejszej pracy opisano solwer WSMP, przetestowano go, a później porównano z własnym solwerem na klaster. Przedtem jednak opisane zostanie podstawowe narzędzie dekompozycji obszaru (ang. *domain decomposition*) - uzupełnienie Schura oraz sposób jego wyznaczania użyty w niniejszej pracy.

5.2 Uzupełnienie Schura

5.2.1 Wprowadzenie

Mając dane cztery macierze $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ o wymiarach $\dim \mathbf{A} = p \times p, \dim \mathbf{B} = p \times q, \dim \mathbf{C} = q \times p, \dim \mathbf{D} = q \times q$ oraz zakładając, że $\mathbf{D}$ jest odwracalna, można zdefiniować macierz $\mathbf{M}$, która jest macierzą blokową o wymiarach $(p+q) \times (p+q)$:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}. \tag{5.1}$$

Uzupełnieniem Schura bloku $\mathbf{D}$ macierzy $\mathbf{M}$ jest macierz o wymiarach $p \times p$ zdefiniowana następująco:

$$\mathbf{S} = \mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C}. \tag{5.2}$$

Uzupełnienie Schura można wykorzystać do rozwiązywania układów równań liniowych w następujący sposób. Dany jest układ równań liniowych:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}, \tag{5.3}$$

gdzie $\mathbf{x}$ oraz $\mathbf{a}$ są wektorami o wymiarach p, natomiast $\mathbf{y}$,$\mathbf{b}$ są wektorami o wymiarach q. Jeśli z drugiego wiersza wyznaczy się $\mathbf{y}$ i wstawi do pierwszego wiersza, to otrzyma się

$$\underbrace{(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})}_{\mathbf{S}}\mathbf{x} = \underbrace{\mathbf{a} - \mathbf{B}\mathbf{D}^{-1}\mathbf{b}}_{\mathbf{s}}, \tag{5.4}$$

czyli:

$$\mathbf{S}\mathbf{x} = \mathbf{s}. \tag{5.5}$$

Oznacza to, że jeżeli można odwrócić macierz $\mathbf{D}$ i $\mathbf{S}$ (uzupełnienie Schura), to można znaleźć $\mathbf{x}$ i używając drugiego wiersza równ. (5.3) wyznaczyć $\mathbf{y}$. W ten sposób problem odwracania macierzy $(p+q) \times (p+q)$ został zredukowany do problemu odwracania macierzy $p \times p$ i $q \times q$.

5.2.2 Uzupełnienie Schura przy równoległym rozwiązywaniu układu równań liniowych dla MES

Stosując MES należy rozwiązać symetryczny układ równań liniowych $\mathbf{Ku} = \mathbf{f}$ wyznaczony dla całej domeny. Gdy korzysta się z metody dekompozycji obszaru (ang. *domain decomposition*), należy podzielić domenę na wiele mniejszych domen (tzw. subdomen). Wtedy dla każdej subdomeny $i \quad (i = 1, ..., n)$ można utworzyć układ równań liniowych na innym węźle obliczeniowym klastra:

$$\begin{bmatrix} \mathbf{K}_i & \mathbf{B}_i^T \\ \mathbf{B}_i & \mathbf{C}_i \end{bmatrix} \begin{bmatrix} \mathbf{u}_i \\ \mathbf{u}_i^I \end{bmatrix} = \begin{bmatrix} \mathbf{f}_i \\ \mathbf{f}_i^I \end{bmatrix}. \tag{5.6}$$

gdzie $\mathbf{u}_i^I$ to niewiadome interfejsowe i $\mathbf{u}_i$ to niewiadome wewnętrzne dla subdomeny i. Dla każdej subdomeny można wyznaczyć uzupełnienie Schura dla macierzy $\mathbf{K}_i$, tak jak opisano w poprzednim podrozdziale:

$$\mathbf{S}_i = \mathbf{C}_i - \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{B}_i^T, \tag{5.7}$$

$$\mathbf{s}_i = \mathbf{f}_i^I - \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{f}_i. \tag{5.8}$$

Po wyeliminowaniu niewiadomych wewnętrznych otrzyma się:

$$\mathbf{S}_i \mathbf{u}_i^I = \mathbf{s}_i. \tag{5.9}$$

Po agregacji dla wszystkich subdomen, otrzymamy układ równań dla interfejsów,

$$\mathbf{S}\mathbf{u}^I = \mathbf{s}, \tag{5.10}$$

gdzie:

$$\mathbf{S} = \sum_{i=1}^{n} \mathbf{S}_i, \qquad \mathbf{s} = \sum_{i=1}^{n} \mathbf{s}_i, \qquad \mathbf{u}^I = \sum_{i=1}^{n} \mathbf{u}_i^I. \tag{5.11}$$

Po rozwiązaniu równ. (5.10), należy wyznaczyć $\mathbf{u}_i$ z układów równań:

$$\mathbf{K}_i \mathbf{u}_i = \mathbf{f}_i - \mathbf{B}_i^T \mathbf{u}_i^I, \tag{5.12}$$

każdy rozwiązując na innym węźle obliczeniowym. Kroki z równ. (5.7), (5.8) oraz (5.12) można wykonać równolegle na różnych węzłach obliczeniowych klastra.

Kluczowe w całym algorytmie jest wyliczenie uzupełnienia Schura $\mathbf{S}_i$ wg równ. (5.7), podczas którego trzeba najpierw sfaktoryzować macierz $\mathbf{K}_i$ a później rozwiązać układ równań z wieloma prawymi stronami tak, aby otrzymać $\mathbf{K}_i^{-1}\mathbf{B}_i^T$. Obliczenie $\mathbf{K}_i^{-1}\mathbf{B}_i^T$ jest bardzo czasochłonne, co pokaże poniższy przykład.

Dla testu kostki $N = 64$ liczba równań wynosi 811 200, a po podzieleniu domeny na dwie identyczne subdomeny (za pomoca płaszczyzny równoległej do ściany kostki) macierz interfejsowa ma wymiar 12 480. Tabele 5.1 oraz 5.2 pokazują jak skaluje się czas wykonania fazy rozwiązania tzn. obliczenia 1, 12 i 24 kolumn z $\mathbf{K}_i^{-1}\mathbf{B}_i^T$ dla dwóch solwerów HSL MA86 i MKL PARDISO. Widzimy, że faza ta słabo się skaluje, tzn. ok. 3 razy dla 12 wątków. Podobne rezultaty uzyskano w pracy [42]. Jest to spowodowane

Tabela 5.1: Skalowalność fazy rozwiązania dla solwerów HSL MA86 oraz MKL PARDISO z wypełnionymi prawymi stronami. Test kostki $N = 64$. T - czas [s], S - przyspieszenie.

	Liczba prawych stron											
	1				12				24			
Liczba wątków	HSL MA86		MKL PARDISO		HSL MA86		MKL PARDISO		HSL MA86		MKL PARDISO	
	T	S	T	S	T	S	T	S	T	S	T	S
1	3.57	1.00	3.71	1.00	14.41	1.00	14.24	1.00	22.24	1.00	22.85	1.00
6	1.77	2.01	1.60	2.32	5.23	2.75	5.15	2.77	8.09	2.75	8.03	2.85
12	1.69	2.12	1.48	2.51	4.89	2.95	6.93	2.05	7.55	2.94	6.79	3.36

Tabela 5.2: Skalowalność fazy rozwiązania dla solwerów HSL MA86 oraz MKL PARDISO z rzadkimi prawymi stronami (0.006%). Test kostki dla $N = 64$. T - czas [s], S - przyspieszenie.

	Liczba prawych stron											
	1				12				24			
Liczba wątków	HSL MA86		MKL PARDISO		HSL MA86		MKL PARDISO		HSL MA86		MKL PARDISO	
	T	S	T	S	T	S	T	S	T	S	T	S
1	3.71	1.00	3.76	1.00	14.37	1.00	14.12	1.00	22.36	1.00	22.49	1.00
6	1.62	2.30	1.52	2.47	5.10	2.82	5.15	2.74	8.10	2.76	7.78	2.85
12	1.84	2.01	1.53	2.46	4.86	2.96	7.70	1.83	7.45	3.00	7.63	2.95

tym, że większość czasu, w trakcie pracy solwera zajmuje faza faktoryzacji i to ta faza była poddana intensywnym badaniom numeryków, a nie faza rozwiązania.

Dla przypadku $N = 64$ faktoryzacja zajmuje ok. 3 minut, więc czas rozwiązania rzędu kilku sekund jest mało istotny w przypadku jednokrotnego wykonania. Jednak przy obliczaniu uzupełnienia Schura, fazę rozwiązania układów trójkątnych (czwarta faza rozwiązywania rzadkich układów równań, patrz rozdz. 4.1.4) trzeba wykonać 12 480 razy.

Można wykonywać fazę rozwiązania pojedynczo albo w niedużych pakietach po 12 lub 24 wektorów prawych stron, natomiast nie można przekazać od razu wszystkich prawych stron, ponieważ to spowodowałoby bardzo duże zapotrzebowanie na pamięć (rozwiązanie jest gęste, a wymiar wektora prawej strony jest rzędu 400 tysięcy). Z tab. 5.1 oraz 5.2 widać, że wyliczenie uzupełnienia Schura dla każdej kolumny z osobna zajęłoby ok. 5 godzin. Jeśli obliczenia zrobić w pakietach po 12 prawych stron, to czas zmniejszy się do 1.5 godziny. Można również, zamiast korzystać z wykonania równoległego fazy rozwiązania, przydzielić każdemu rdzeniowi do wykonania jedno rozwiązanie, wtedy czas policzenia uzupełnienia Schura wyniósłby ok. 1 godziny, czyli także zbyt długo. Z tab. 5.1 oraz 5.2 widać także, że wpływ stopnia wypełnienia prawych stron jest mało znaczący.

W następnym podrozdziale zostanie przedstawiona metoda przyspieszenia obliczeń uzupełnienia Schura, która w rozdz. 5.3 zostanie wykorzystana do rozwiązywania układu równań na wielu węzłach obliczeniwoych. Dodatkowo metoda ta będzie wykorzystana w rozdz. 5.2.5 do obniżenia zapotrzebowania na pamięć podczas fazy faktoryzacji.

5.2.3 Cześciowa faktoryzacja do obliczania uzupełnienia Schura

W poprzednim podrozdziale pokazano, że obliczanie uzupełnienia Schura poprzez wielokrotne wykonywanie fazy rozwiązania może prowadzić do bardzo długich czasów wykonania oraz będzie słabo skalowalne na maszynie wielordzeniowej ze względu na człon $\mathbf{K}_i^{-1}\mathbf{B}_i^T$. Z drugiej strony wiadomo, że faza faktoryzacji numerycznej osiąga akceptowalne przyspieszenia na maszynach wielordzeniowych. Dlatego warto skorzystać z metody częściowej faktoryzacji (CF) do obliczeń uzupełnienia Schura. Została ona przedstawiona w pracy [89] i polega na obliczeniu dekompozycji **LU** dla macierzy $\bar{\mathbf{K}}_i$ z równ. (5.6):

$$\bar{\mathbf{K}}_i := \begin{bmatrix} \mathbf{K}_i & \mathbf{B}_i^T \\ \mathbf{B}_i & \mathbf{C}_i \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ \mathbf{0} & \mathbf{U}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11}\mathbf{U}_{11} & \mathbf{L}_{11}\mathbf{U}_{12} \\ \mathbf{L}_{21}\mathbf{U}_{11} & \mathbf{L}_{21}\mathbf{U}_{12} + \mathbf{L}_{22}\mathbf{U}_{22} \end{bmatrix}, \tag{5.13}$$

gdzie $\mathbf{L}_{ij}$ i $\mathbf{U}_{ij}$ to macierze trójkątne odpowiednio dolne i górne. Porównując odpowiednie bloki macierzy $\bar{\mathbf{K}}_i$ otrzymuje się $\mathbf{K}_i = \mathbf{L}_{11}\mathbf{U}_{11}$, $\mathbf{B}_i^T = \mathbf{L}_{11}\mathbf{U}_{12}$, $\mathbf{B}_i = \mathbf{L}_{21}\mathbf{U}_{11}$ oraz $\mathbf{C}_i = \mathbf{L}_{21}\mathbf{U}_{12} + \mathbf{L}_{22}\mathbf{U}_{22}$. Wyznaczając $\mathbf{U}_{12}$ oraz $\mathbf{L}_{21}$ z drugiego i trzeciego równania uzyska się:

$$\mathbf{L}_{22}\mathbf{U}_{22} = \mathbf{C}_i - \mathbf{B}_i\mathbf{U}_{11}^{-1}\mathbf{L}_{11}^{-1}\mathbf{B}_i^T = \mathbf{C}_i - \mathbf{B}_i\mathbf{K}_i^{-1}\mathbf{B}_i^T = \mathbf{S}_i, \tag{5.14}$$

gdzie $\mathbf{S}_i$ to uzupełnienie Schura z równ. (5.7). Faktoryzacja jest nazwana częściową, ponieważ nie oblicza się $\mathbf{L}_{22}$ i $\mathbf{U}_{22}$. Ponadto, macierze $\mathbf{L}_{11}$ i $\mathbf{U}_{11}$ są rozkładem **LU** dla macierzy $\mathbf{K}_i$, co można wykorzystać w dalszych etapach rozwiązywania układu. Założono niejawnie, że $\mathbf{K}_i$ i $\bar{\mathbf{K}}_i$ są odwracalne.

Trzeba pamiętać również, że przed fazą faktoryzacji jest wykonywana faza przenumerowania wierszy i kolumn macierzy, w celu zredukowania efektu wypełnienia. Dla macierzy $\bar{\mathbf{K}}_i$ dochodzi dodatkowe ograniczenie - wiersze i kolumny macierzy $\mathbf{C}_i$ nie mogą być przestawiane. Z tym ograniczeniem można wykonać przenumerowanie na dwa sposoby. Albo znajdowane jest takie przenumerowanie, że żaden z wierszy i żadna kolumna macierzy $\mathbf{C}_i$ nie jest przestawiana, ale struktura (wypełnienie) wierszy i kolumn macierzy $\mathbf{B}_i$ oraz $\mathbf{B}_i^T$ mogą wpływać na przenumerowanie wierszy i kolumn macierzy $\mathbf{K}_i$. Albo znajdowane jest przenumerowanie wierszy i kolumn macierzy $\mathbf{K}_i$, a do pozostałych wierszy i kolumn macierzy $\bar{\mathbf{K}}_i$ nie wykonuje się przenumerowania. Nie będzie to równie dobre jak przenumerowanie wszystkich wierszy i kolumn macierzy $\bar{\mathbf{K}}_i$, ale w tym momencie nie ma dobrych algorytmów dla pierwszego sposobu, dlatego w rozprawie zastosowano drugi sposób.

5.2.4 Implementacja częściowej faktoryzacji w HSL MA86

Powyższy algorytm został zaimplementowany przez autora niniejszej rozprawy w solwerze HSL MA86; zmodyfikowany solwer oznaczono modMA86. Metoda częściowej faktoryzacji jest także zaimplementowana np. w solwerze PARDISO [89].

Implementacja częściowej faktoryzacji macierzy $\bar{\mathbf{K}}_i$ w modMA86 polegała na wykonaniu dwóch czynności:

1. Podziale zadań faktoryzacji na dwie grupy, patrz Kod 5.1. Zadania faktoryzacji działające na bloku z macierzą $\mathbf{K}_i$ należą do pierwszej grupy i będą wykonane normalnie. Zadania faktoryzacji działające na blokach poza macierzą $\mathbf{K}_i$ należą do drugiej grupy i ich wykonanie będzie zmodyfikowane.

2. Implementacji modyfikacji zadania faktoryzacji. Modyfikacja polega na niewykonywaniu faktycznej faktoryzacji, a tylko na redukcji zależności między superwęzłami w drzewie eliminacji, patrz Kod 5.2. Dzięki redukcji zależności uzyskano pewność, że nieskończona równoległa pętla po zadaniach będzie zakończona (por. rozdz. 4.3.1).

Kod 5.1: Podział zadań faktoryzacji na dwie grupy. Procedura MA86_analyse_double

```
1   if(control%schur.ge.sptr(node) &
2   .and.control%schur.lt.(sptr(node+1)-1)) then
3    l_nb = control%schur - sptr(node) + 1
4    if(l_nb.eq.control%schur) then
5     keep%nodes(node)%nb = l_nb
6    else
7     sz = 1; cur_l_nb = l_nb
8     do i = 2, l_nb-1
9      do while ( MOD(cur_l_nb, i).eq.0)
10      cur_l_nb = cur_l_nb / i
11      if(sz*i.gt.MIN(nb_default,control%schur)) exit
12      sz = i*sz
13     enddo
14    enddo
15    l_nb = sz; keep%nodes(node)%nb = l_nb
16   endif
17  endif
```

Kod 5.2: Modyfikacja zadania faktoryzacji. Procedura task_dispatch

```
1  if(control%schur.lt.lfact(bcol)%sa) then
2   lfact(bcol)%nelim = blkn
3   do i = 1,blkn
4    map(i) = i
5   end do
6  else
7   call factor_solve_block(blkm, blkn, blkn,  &
8    delay_col, colwork, blkm*blkn, work, lfact(bcol)%d, &
9    map, lfact(bcol)%nelim, tstats, control, flag)
10 endif
11 if(control%schur.lt.lfact(bcol)%sa) then
12  call add_updates_new_incomp(stack, map, nodes, &
13   blocks, dblk, blocks(dblk)%node, control, info, st)
14 else
15  call add_updates_new(stack, map, nodes, blocks, &
16   dblk, blocks(dblk)%node, control, info, st)
17 endif
```

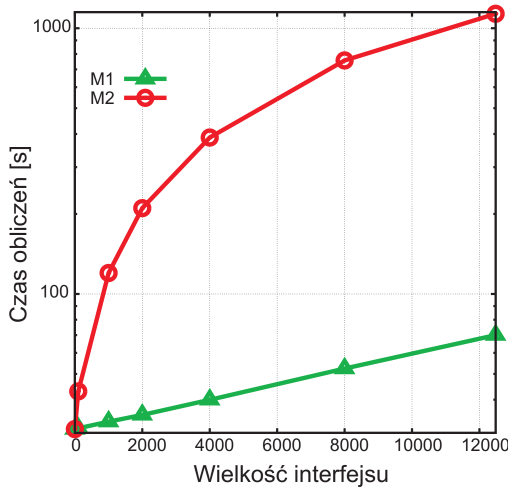
Porównanie czasu obliczeń uzupełnienia Schura dla podziału na zadania na dwie części znajduje się w tab. 5.3, gdzie M1 oznacza rozwiązywanie układu w pakietach po 12 prawych stron, a M2 oznacza częściową faktoryzację macierzy; rozróżniono czas faktoryzacji i tworzenia uzupełnienia Schura dla M1, a w przypadku M2 oblicza się jednocześnie faktoryzację i uzupełnienie Schura. Używając M2, obliczenie uzupełnienia Schura a pomocą modMA86 jest szybsze niż za pomocą PARDISO o ok. 18%. Wynika to z faktu, że dzięki modyfikacjom wprowadzonym do solwera modMA86 i opisanym w rozdz. 4.3 uzyskano skrócenie czasu faktoryzacji.

Tabela 5.3: Czas obliczenia uzupełnienia Schura dla podziału zadania na dwie części. Test kostki N=32 (104 544 niewiadomych, interfejs - 3 168 niewiadomych, subdomena i interfejs - 50 688 niewiadomych)

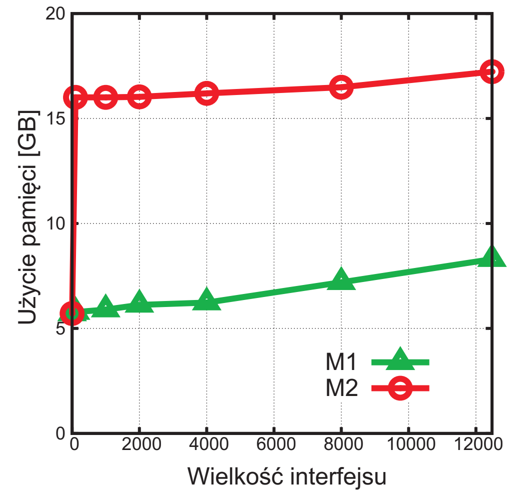
Metoda	Solver	Faktoryzacja [s]	Uzupeł. Schura [s]	Łącznie [s]	M1/M2
M1	PARDISO	0.82	17.27	18.09	
	modMA86	0.79	19.20	19.99	
M2	PARDISO	1.99	-	1.99	9.09
	modMA86	1.64	-	1.64	12.19

Rysunek 5.1a przedstawia czas wykonania obliczeń uzupełnienia Schura w zależności od rozmiaru interfejsu. Dla interfejsu o wielkości 12480 metoda M2 jest szybsza od M1 o 94%. Na rys. 5.1b porównano użycie pamięci, przez metodę M1 i M2. Również tutaj można zaobserwować zalety metody M2. Z uwagi na fakt, że z reguły uzupełnienie Schura jest macierzą o wiele bardziej gęstą niż macierz, z której jest ono tworzone, trzeba zaalokować pełny wektor prawych stron tyle razy ile wynosi rozmiar interfejsu. Ponieważ wektor prawych stron jest nieporównywalnie większy niż interfejs (ok. 10 razy), skok

zapotrzebowania na pamięć na początku jest największy w przypadku metody M1. W przypadku metody M2 nie trzeba alokować dodatkowych wektorów, dlatego zapotrzebowanie na pamięć rośnie stabilnie wraz ze wzrostem wielkości interfejsu.



(a) Czas obliczeń.



(b) Wykorzystanie pamięci.

Rysunek 5.1: Porównanie metod obliczenia uzupełnienia Schura, dla podziału zadania na dwie części. Test kostki N = 64. Subdomena - 386 880 niewiadomych, solwer modMA86, prawa dolna podmacierz macierzy rozszerzonej zerowa.

5.2.5 Sekwencyjna częściowa faktoryzacja na pojedynczym węźle dla dwóch subdomen

Wykorzystując rozumowanie przedstawione w rozdz. 5.2.2 dla dwóch subdomen oraz zaprezentowaną metodę M2 do obliczenia uzupełnienia Schura przez solwer można stworzyć algorytm faktoryzujący macierz, którego zapotrzebowanie na pamięć będzie zdecydowanie niższe od standardowego. Następujący algorytm wykonuje metodę przedstawioną w rozdz. 5.2.2 sekwencyjnie:

1. Wygeneruj macierz $\bar{\mathbf{K}}_1$,
2. Częściowo faktoryzuj macierz $\bar{\mathbf{K}}_1$ i wylicz uzupełnienie Schura,
3. Wygeneruj macierz $\bar{\mathbf{K}}_2$,
4. Częściowo faktoryzuj macierz $\bar{\mathbf{K}}_2$ i wylicz uzupełnienie Schura,
5. Rozwiąż równanie dla interfejsów z równ. (5.10),
6. Popraw prawą stronę dla $i = 2$ w równ. (5.12),
7. Rozwiąż równ. (5.12) dla $i = 2$,
8. Popraw prawą stronę dla $i = 1$ w równ. (5.12),

9. Faktoryzuj macierz $\mathbf{K}_1$,

10. Rozwiąż równ. (5.12) dla $i = 1$.

Dzięki temu, że wypełnienie macierzy $\bar{\mathbf{K}}_i$ podczas częściowej faktoryzacji, będzie blisko o połowę mniejsze od wypełnienia macierzy $\mathbf{K}$, uzyskano redukcję pamięci w całym procesie. Warto zwrócić uwagę, że po częściowej faktoryzacji macierzy $\bar{\mathbf{K}}_1$, trzeba zwolnić pamięć, co powoduje, że do rozwiązywania równ. (5.12) potrzebne jest powtórzenie faktoryzacji dla macierzy $\mathbf{K}_1$ (punkt 9). Macierzy $\mathbf{K}_2$ nie trzeba ponownie faktoryzować, ze względu na fakt wspomniany w poprzednim podrozdziale, tzn. podczas częściowej faktoryzacji macierzy $\bar{\mathbf{K}}_2$, obliczono również faktoryzację macierzy $\mathbf{K}_2$, co wykorzystano przy wyliczeniu równ. (5.12) dla $i = 2$ w punktach 6 oraz 7.

Powyższy sekwencyjny algorytm został zaimplementowany w dwóch wersjach: w pierwszej do wykonania częściowej faktoryzacji i rozwiązania równania dla interfejsów wykorzystano solwer PARDISO a w drugiej solwer modMA86; tę drugą wersję oznaczono modMA86mem. Z tab. 5.4 widać, że metoda sekwencyjna jest o ok. 40% wolniejsza od standardowego wykonania w podwójnej precyzji, a z tab. 5.5 wynika, że daje ona oszczędność pamięci 32% porównując do standardowej metody. W zestawieniu podano także rezultaty dla sekwencyjnej metody w pojedynczej precyzji (S) (bez poprawiania wyników); jest ona o 12% szybsza i daje ona oszczędność pamięci o 55%, porównując do standardowej metody w podwójnej precyzji.

Dla porównania, dodatkowo badano metodę z mieszaną precyzją (M) (solver modMA86mix) i jest ona 2.2 razy szybsza i daje oszczędność pamięci o 46% , porównując do standardowej metody w podwójnej precyzji. Widzimy, że metoda z mieszaną precyzją jest znacznie szybsza niż metoda sekwencyjna (nawet w pojedynczej precyzji) jednak redukcja pamięci jest nieco niższa.

Tabela 5.4: Redukcja wykorzystanej pamięci. Porównanie czasu. Test kostki $N = 64$.

	Czas obliczeń [s]				Zmiana [%]		
	Metoda				Metoda		
	Standardowa	Mieszana	Sekwencyjna		Mieszana	Sekwencyjna	
Solwer	(D)	(M)	(D)	(S)	(M)	(D)	(S)
PARDISO	210.98	-	288.90	-	-	+37	-
modMA86mem	156.07	71.03	218.83	138.09	-54	+40	-12

Tabela 5.5: Redukcja wykorzystanej pamięci. Porównanie pamięci. Test kostki $N = 64$.

	Pamięć [GB]				Zmiana [%]		
	Metoda				Metoda		
	Standardowa	Mieszana	Sekwencyjna		Mieszana	Sekwencyjna	
Solwer	(D)	(M)	(D)	(S)	(M)	(D)	(S)
PARDISO	14.16	-	8.71	-	-	-38	-
modMA86mem	14.47	7.78	9.91	6.46	-46	-32	-55

5.3 Rozwiązywanie bezpośrednie układów równań na wielu węzłach

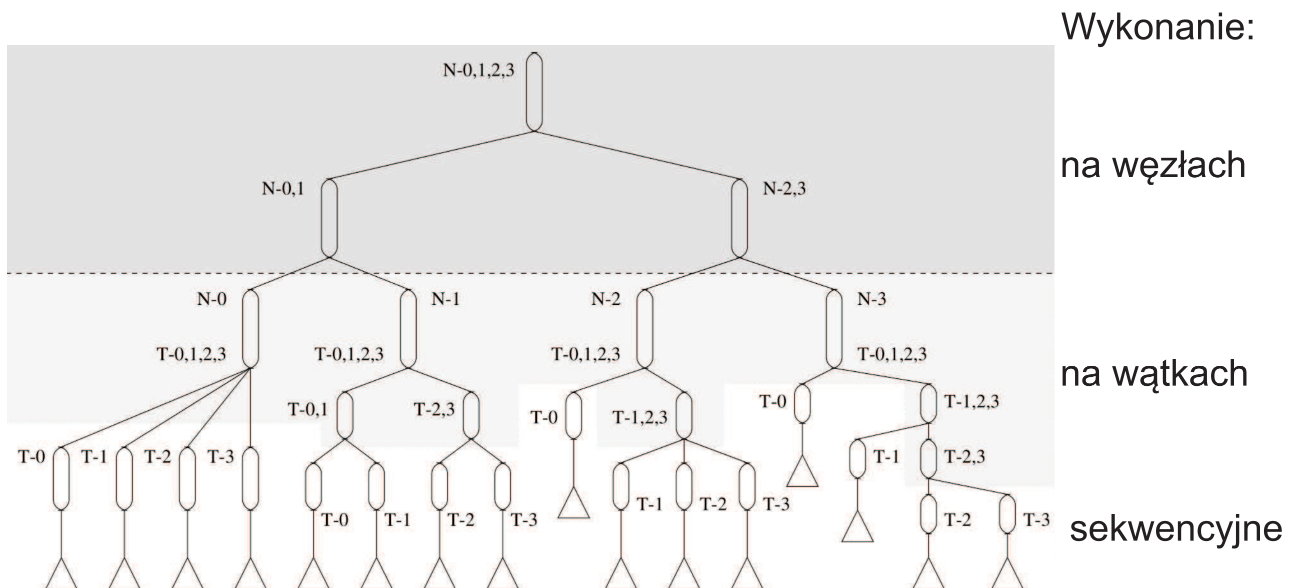
5.3.1 Testy WSMP

Solwer WSMP (Watson Sparse Matrix Package) firmy IBM [37], służący do rozwiązywania symetrycznych i rzadkich układów równań liniowych, posiada również, oprócz wersji wielowątkowej opisanej w rozdz. 4.2.1, wersję rozproszoną wykorzystującą przesyłanie komunikatów (MPI).

Podstawową trybem pracy rozproszonego WSMP jest tryb „0-master”. Praca w tym trybie oznacza, że wszystkie dane, włączając macierz i wektor prawych stron, początkowo są przechowywane na węźle oznaczonym numerem „0”. W tym rozdziale węzeł oznacza węzeł obliczeniowy klastra. Wersja rozproszona, podobnie jak wielowątkowa, wykonuje cztery podstawowe fazy dla równoległych solwerów: (1) przenumerowanie, (2) symboliczna faktoryzacja, (3) numeryczna faktoryzacja, (4) rozwiązanie układów trójkątnych. Wszystkie fazy są wykonywane równolegle oprócz symbolicznej faktoryzacji. Faza ta jest najmniej czasochłonna i jest wykonywana na węźle „0”. Węzeł „0” jest wykorzystywany niewspółmiernie mocniej w trakcie fazy przenumerowania niż inne węzły.

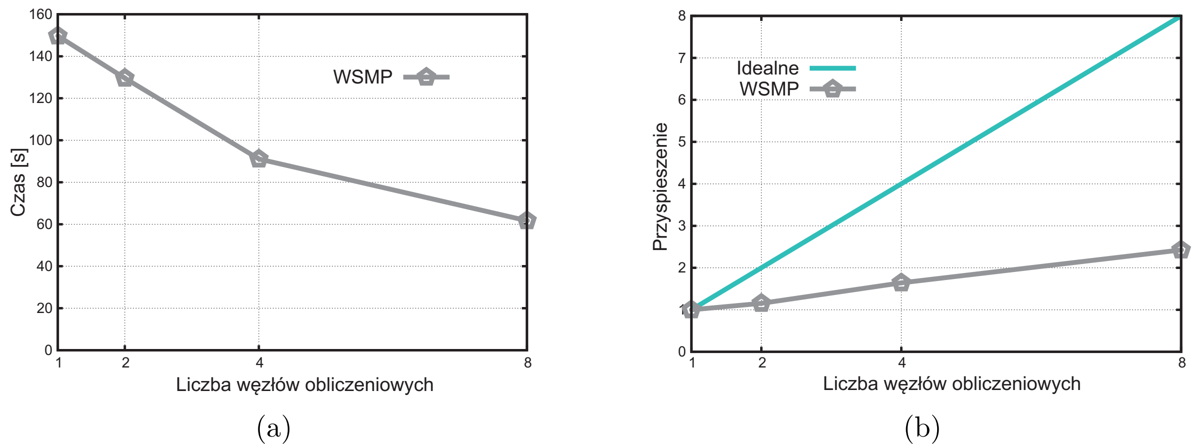
Autorzy WSMP ostrzegają, że czas przenumerowania i analizy może wzrastać (w porównaniu do czasu faktoryzacji) wraz ze wzrostem liczby węzłów, gdyż to faza faktoryzacji posiada wysoką skalowalność. Wersja rozproszona może być uruchomiona na dowolnej liczbie węzłów ale najlepszą wydajność osiąga się gdy liczba węzłów to potęgi 2, dlatego w testach stosowano tylko takie liczby węzłów.

Faza faktoryzacji w WSMP bazuje na algorytmie multifrontalnym (patrz rozdz. 4.1.7), a do zrównoleglenia wykorzystuje się drzewo eliminacji. Przykładowy podział drzewa eliminacji między węzłami został przedstawiony na rys. 5.2.



Rysunek 5.2: Podział drzewa eliminacji do wykonania na klastrze z 4 węzłami (N) i 4 wątkami każda (T).

Przetestowano solwer rozproszony WSMP wykorzystując test kostki dla $N = 64$. Do wszystkich testów w tym i kolejnych podrozdziałach wykorzystano te same węzły obliczeniowe klastra GRAFEN [29] oraz 12 wątków na każdym węźle. Na rys. 5.3a pokazano czas wykonania fazy faktoryzacji w zależności od liczby węzłów. Czas dla 8 węzłów jest ponad dwukrotnie krótszy niż w przypadku jednego węzła. Rysunek 5.3b przedstawia skalowalność WSMP, i widać, że uzyskane przyspieszenie znacznie odbiega od idealnego przyspieszenia. Stanowi to główną motywacją do podjęcia dalszych prac w tym podrozdziale. Tabela 5.6 zawiera wykorzystanie pamięci przez WSMP. Było ono mierzone na każdym węźle, a w tabeli zamieszczono wartość maksymalną.



Rysunek 5.3: Czas wykonania (a) i przyspieszenie (b) fazy faktoryzacji, solwer WSMP na klaster. Test kostki $N = 64$..

Tabela 5.6: Maksymalne użycie pamięci na jednym weźle podczas faktoryzacji, solwer WSMP na klaster. Test kostki $N = 64$.

Liczba węzłów	Wykorzystanie pamięci[GB]
1	20.47
2	16.61
4	9.65
8	5.72

5.3.2 Własny solwer na klaster

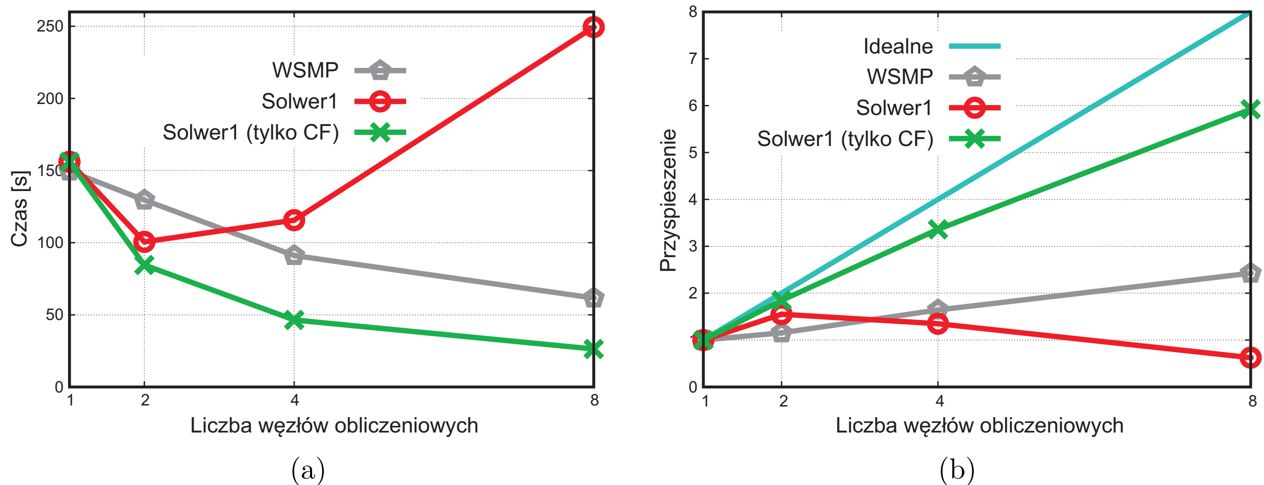
W rozdz. 5.2.2 przedstawiono sposób w jaki można wykorzystać uzupełnienie Schura do rozwiązania układu równań liniowych na wielu węzłach. Pokazano również w rozdz. 5.2.4, że dzięki częściowej faktoryzacji, można wyznaczać uzupełnienie Schura w efektywny sposób. Jeżeli teraz obie metody zostaną połączone to otrzyma się algorytm na rozwiązywanie układu równań liniowych na wielu węzłach.

Algorytm wygląda następująco:

1. Każdy węzeł otrzymuje od węzła „0” macierz $\bar{\mathbf{K}}_i$,

2. Każdy węzeł częściowo faktoryzuje macierz $\bar{\mathbf{K}}_i$,
3. Każdy węzeł wylicza uzupełnienie Schura dla prawej strony,
4. Węzeł „0" otrzymuje uzupełnienia Schura od wszystkich węzłów,
5. Węzeł „0" rozwiązuje równanie dla interfejsów z równ. (5.10),
6. Każdy węzeł otrzymuje od węzła „0" rozwiązanie równania interfejsów,
7. Każdy węzeł poprawia prawą stronę,
8. Każdy węzeł rozwiązuje równ. (5.12).

Podczas wyznaczania uzupełnienia Schura za pomocą częściowej faktoryzacji (krok 2), faktoryzowane są poszczególne macierze $\mathbf{K}_i$, co wykorzystano w fazie rozwiązania (krok 8). Powyższy algorytm został zaimplementowany z wykorzystaniem solwera modMA86 i oznaczony jako Solwerze1. Jeśli zastosuje się podejście naiwne i krok 5 będzie rozwiązany za pomocą tego samego solwera co krok 2, to uzyska się czasy wykonania faktoryzacji pokazane na rys. 5.4a. Widać, że czas częściowej faktoryzacji zaimplementowanej w Solwer1 skaluje się zadowalająco (patrz rys. 5.4b). Jednak łączny czas faktoryzacji (częściowa faktoryzacja i faktoryzacja macierzy interfejsów - kroki 2 i 5), nie skaluje się dobrze (patrz rys. 5.4b) i uzyskano czasy gorsze niż przy wykonaniu na jednym węźle.



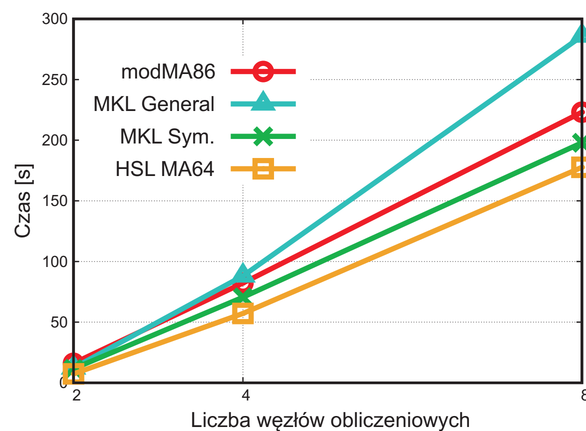
Rysunek 5.4: Czas wykonania (a) i przyspieszenie (b) częściowej faktoryzacji i faktoryzacji macierzy interfejsów. Test kostki $N = 64$.

Dlatego zbadano strukturę macierzy dla interfejsów, która zostaje poddana faktoryzacji w kroku 5. W tab. 5.7 podano charakterystkę macierzy dla interfejsów, w zależności od liczby domen, na którą podzielono całą macierz. Liczba elementów niezerowych jest podana dla części trójkątnej macierzy (dolnej lub górnej, ponieważ macierz jest symetryczna). Widać, że jest ona macierzą gęstą; fakt ten zauważono także w [63]). Dlatego do faktoryzacji macierzy dla interfejsów przetestowano także solwery dla macierzy gęstych. Rysunek 5.5 przedstawia czasy faktoryzacji dla trzech solwerów dla macierzy gęstych

(MKL General - procedura DGESV z biblioteki MKL, MKL Sym. - procedura DSYSV z biblioteki MKL i HSL MA64) oraz czas faktoryzacji dla solwera rzadkiego (modMA86). (Na tym rysunku nie ma wyników dla jednego węzła, ponieważ dla jednego węzła jest tylko jedna subdomena, więc nie ma macierzy interfejsowej.) Najszybciej działa solwer MA64, jednak łączny czas wynosi 204s i faktoryzacja macierzy dla interfejsów wciąż pozostaje barierą dla skalowalnego rozwiązywania układów równań na wielu węzłach.

Tabela 5.7: Charakterystyka macierzy dla interfejsów w zależności od liczby subdomen. Test kostki $N = 64$.

Liczba domen	Liczba niewiadomych	Liczba elementów niezerowych [mln]	Wypełnienie [%]
2	12 480	77.89	100.0
4	25 059	256.21	81.6
8	37 254	295.25	42.5



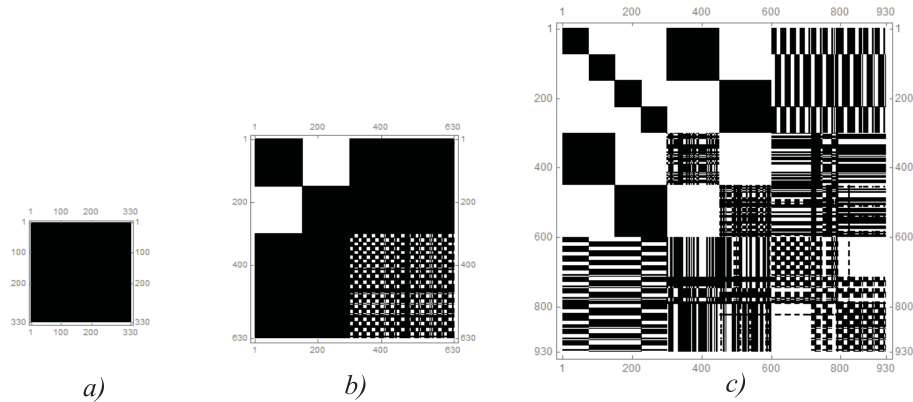
Rysunek 5.5: Czas wykonania faktoryzacji macierzy interfejsów używając solwerów gęstych i solwera modMA86. Liczba węzłów = liczba subdomen. Test kostki $N = 64$.

Naturalnym kolejnym pomysłem, jest wykorzystanie wszystkich węzłów do faktoryzacji macierzy dla interfejsów, a nie tylko węzła „0”. Pierwszy sposób, który się narzuca, to podzielenie macierzy dla interfejsów na tyle węzłów ile jest dostępnych i wykonanie tych samych kroków co dla całej macierzy. Z tab. 5.8 można wywnioskować jednak, że takie działanie jest skazane na niepowodzenie. Przy podziale macierzy dla interfejsów na tyle domen, na ile podzielono macierz początkową, uzyska się wielkość interfejsu rzędu wielkości macierzy dla interfejsów. To oznacza, że trzeba byłoby faktoryzować macierz podobną do tej, której czas faktoryzacji chciano skrócić. Prowadzi to do wniosku, że to podejście nie pozwoli skrócić czasu wykonania faktoryzacji.

Rysunek 5.6 przedstawia strukturę macierzy dla interfejsów w zależności od podziału na liczbę domen (Test kostki $N = 10$). Można zauważyć, że macierz dla interfejsów jest naturalnie podzielona na dwa razy mniej domen, niż została podzielona macierz wejściowa. Można wykorzystać ten fakt i podzielić macierz dla interfejsów między węzły właśnie wg tego naturalnego podziału, wynikającego z podziału macierzy wejściowej.

Tabela 5.8: Podział macierzy dla interfejsów. Test kostki $N = 64$.

Liczba domen	Wielkość macierzy dla interfejsów	Wielkość domeny minimalna	Wielkość domeny maksymalna	Wielkość interfejsu
2	12 480	78	102	12 300
4	25 059	51	58	24 840
8	37 254	37	44	36 933



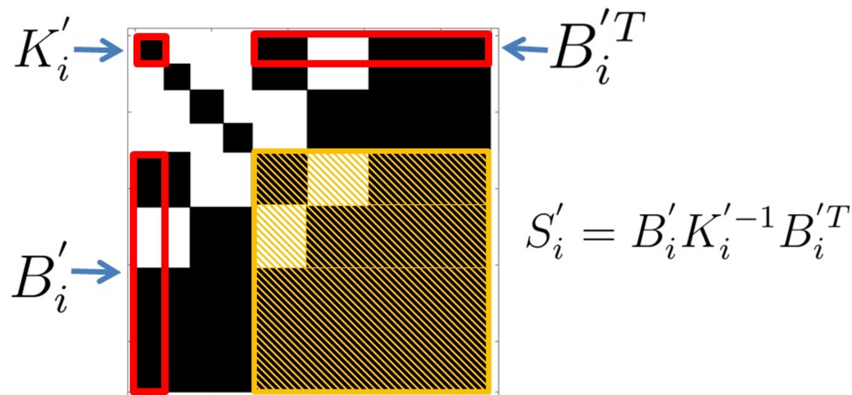
Rysunek 5.6: Struktura macierzy interfejsowej dla podziału na: a) 2 domeny, b) 4 domeny i c) 8 domen. Test kostki $N = 10$.

Algorytm faktoryzacji na wielu węzłach przedstawia się następująco:

1. Każdy węzeł otrzymuje od węzła „0" macierz $\bar{\mathbf{K}}_i$,
2. Każdy węzeł wykonuje częściową faktoryzację macierzy $\bar{\mathbf{K}}_i$,
3. Każdy węzęł wylicza uzupełnienie Schura dla prawej strony,
4. Węzeł „0" otrzymuje uzupełnienia Schura,
5. Faktoryzuj macierz dla interfejsów z równ. (5.10):
   (a) Połowa węzłów otrzymuje od węzła „0" część macierzy dla interfejsów,
   (b) Połowa węzłów wykonuje częściową faktoryzację macierzy,
   (c) Połowa węzłów wylicza uzupełnienie Schura dla nowych prawych stron,
   (d) Węzeł „0" otrzymuje uzupełnienia Schura,
   (e) Węzeł „0" faktoryzuje nową macierz dla interefejsów,
   (f) Węzeł „0" wylicza nowe rozwiązanie równania interfejsów,
   (g) Połowa węzłów otrzymuje od węzła „0" nowe rozwiązanie równania interfejsów,
   (h) Połowa węzłów poprawia prawą stronę,
   (i) Połowa węzłów wylicza rozwiązanie równania interfejsów,

   (j) Węzeł „0” zbiera rozwiązanie równania interfejsów,

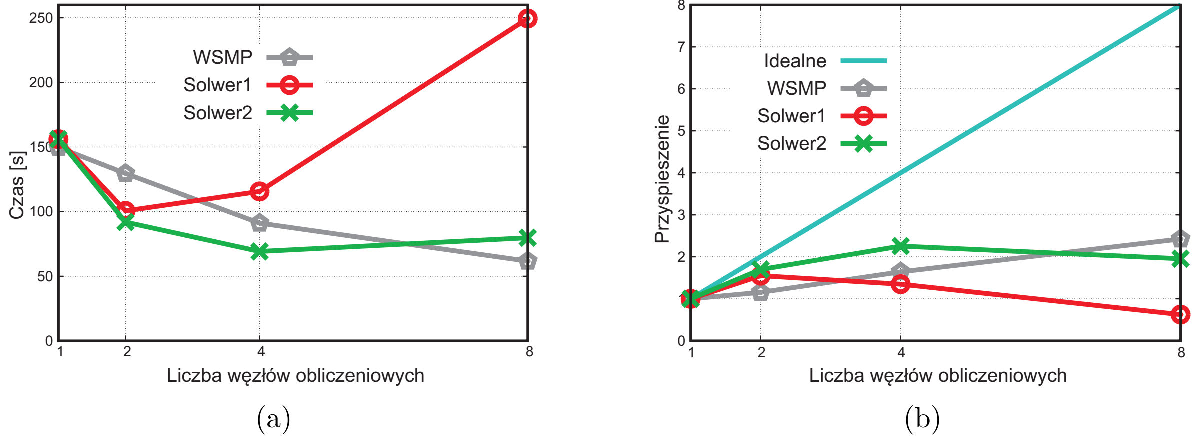
6. Każdy węzeł otrzymuje od węzła „0” rozwiązanie równania interfejsów,
7. Każdy węzeł poprawia prawą stronę,
8. Każdy węzeł rozwiązuje równ. (5.12).



Rysunek 5.7: Faktoryzacja macierzy dla interfejsów. Kolor czerwony część macierzy biorących udział w częściowej faktoryzacji. Kolorem żółtym zaznaczono część macierzy, która zostanie zaktualizowana.

Punkt 5 powyższego algorytmu został przedstawiony na rys. 5.7 i widać, że faktoryzacja macierzy dla interfejsów polega na wykonaniu tych samych czynności, co przy faktoryzacji całej macierzy. Części macierzy dla interfejsów zostały oznaczone symbolem prim ($'$).

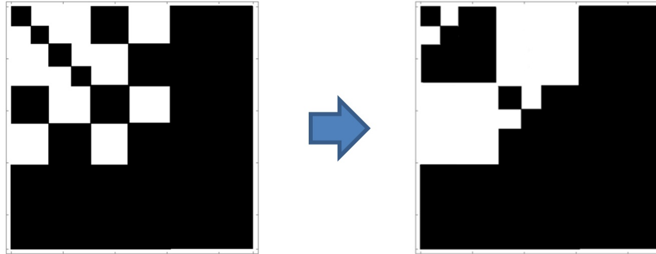
Czas wykonania faktoryzacji dla powyższego algorytmu został przedstawiony na rys. 5.8a i oznaczony jako Solwer2. Dla porównania zamieszczono również czasy dla solwera WSMP oraz poprzedni algorytm Solwer1. Częściowa faktoryzacja oraz faktoryzacja nowej macierzy dla interfejsów w algorytmie Solwer2 została wykonana za pomocą solwera HSL MA64. Z rys. 5.8a wynika, że czas faktoryzacji dla 2 i 4 węzłów jest krótszy dla Solwer2, ale dla 8 węzłów czas faktoryzacji jest dłuższy niż dla WSMP. Dlatego dla 8 węzłów zastosowano inne podejście, tzw. hierarchiczną faktoryzację i zostanie ono opisane w następnym podrozdziale.



Rysunek 5.8: Porównanie czasu faktoryzacji (a) i przyspieszenia (b) dla WSMP i dwóch implementacji własnych. Test kostki $N = 64$.

5.3.3 Hierarchiczna faktoryzacja macierzy dla interfejsów

Przedstawiony poniżej sposób został użyty w pracy [34] dla solwera wielowątkowego, ale wzory zostały podane bez wyprowadzenia. W niniejszej pracy zostaną one jawnie wyprowadzone.



Rysunek 5.9: Przenumerowanie uzupełnienia Schura dla 8 domen.

Hierarchiczna faktoryzacja bazuje na spostrzeżeniu, że macierz dla interfejsów $\mathbf{S}$ dla 8 domen można tak przenumerować (patrz rys. 5.9), aby miała ona następującą strukturę:

$$\mathbf{S} = \begin{bmatrix} \mathbf{K}'_1 & \mathbf{0} & \mathbf{B}'^T_3 \\ \mathbf{0} & \mathbf{K}'_2 & \mathbf{B}'^T_4 \\ \mathbf{B}'_3 & \mathbf{B}'_4 & \mathbf{C}'_3 \end{bmatrix} = \left[\begin{array}{ccc|ccc|c} \mathbf{K}'_{11} & \mathbf{0} & \mathbf{B}'^T_{11} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}'^T_{31} \\ \mathbf{0} & \mathbf{K}'_{12} & \mathbf{B}'^T_{12} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}'^T_{32} \\ \mathbf{B}'_{11} & \mathbf{B}'_{12} & \mathbf{C}'_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & B'^T_{33} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{K}'_{21} & \mathbf{0} & \mathbf{B}'^T_{21} & \mathbf{B}'^T_{41} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{K}'_{22} & \mathbf{B}'^T_{22} & \mathbf{B}'^T_{42} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}'_{21} & B'_{22} & \mathbf{C}'_2 & \mathbf{B}'^T_{43} \\ \hline \mathbf{B}'_{31} & \mathbf{B}'_{32} & \mathbf{B}'_{33} & \mathbf{B}'_{41} & \mathbf{B}'_{42} & \mathbf{B}'_{43} & \mathbf{C}'_3 \end{array}\right]. \tag{5.15}$$

Uzupełnienie Schura dla macierzy $\mathbf{K}'_1$ oraz $\mathbf{K}'_2$ wynosi:

$$\begin{aligned} \mathbf{S}'_1 &= \mathbf{C}'_1 - \mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11} - \mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}, \\ \mathbf{S}'_2 &= \mathbf{C}'_2 - \mathbf{B}'_{21}\mathbf{K}'^{-1}_{21}\mathbf{B}'^T_{21} - \mathbf{B}'_{22}\mathbf{K}'^{-1}_{22}\mathbf{B}'^T_{22}. \end{aligned} \tag{5.16}$$

Z kolei macierz dla interfejsów dla całej macierzy $\mathbf{S}$ wynosi:

$$\mathbf{S}' = \mathbf{C}'_3 - \mathbf{B}'_1\mathbf{K}'^{-1}_1\mathbf{B}'^T_1 - \mathbf{B}'_2\mathbf{K}'^{-1}_2\mathbf{B}'^T_2. \tag{5.17}$$

Aby wyznaczyć macierze $\mathbf{K}'^{-1}_1$ oraz $\mathbf{K}'^{-1}_2$ będzie wykorzystany następujący wzór na odwrotność macierzy blokowej:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1} \\ -\mathbf{S}^{-1}\mathbf{C}\mathbf{A}^{-1} & \mathbf{S}^{-1} \end{bmatrix}, \tag{5.18}$$

gdzie $\mathbf{S} = \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}$. Dla macierzy diagonalnej:

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{D}^{-1} \end{bmatrix}. \tag{5.19}$$

Po zastosowaniu równ. (5.18) do macierzy $\mathbf{K}'_1$

$$\begin{aligned}
(\mathbf{K}'_1)^{-1} &= \begin{bmatrix} \begin{bmatrix} \mathbf{K}'_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'_{12} \end{bmatrix} & \begin{bmatrix} \mathbf{B}'^T_{11} \\ \mathbf{B}'^T_{12} \end{bmatrix} \\ \begin{bmatrix} \mathbf{B}'_{11} & \mathbf{B}'_{12} \end{bmatrix} & C'_1 \end{bmatrix}^{-1} \\
&= \begin{bmatrix} \begin{bmatrix} \mathbf{K}'_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'_{12} \end{bmatrix}^{-1} + \begin{bmatrix} \mathbf{K}'_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'_{12} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{B}'^T_{11} \\ \mathbf{B}'^T_{12} \end{bmatrix} \mathbf{S}'^{-1}_1 \begin{bmatrix} \mathbf{B}'_{11} & \mathbf{B}'_{12} \end{bmatrix} \begin{bmatrix} \mathbf{K}'_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'_{12} \end{bmatrix}^{-1} & -\begin{bmatrix} \mathbf{K}'_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'_{12} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{B}'^T_{11} \\ \mathbf{B}'^T_{12} \end{bmatrix} \mathbf{S}'^{-1} \\ -\mathbf{S}'^{-1} \begin{bmatrix} \mathbf{B}'_{11} & \mathbf{B}'_{12} \end{bmatrix} \begin{bmatrix} \mathbf{K}'_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'_{12} \end{bmatrix}^{-1} & \mathbf{S}'^{-1} \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} \mathbf{K}'^{-1}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'^{-1}_{12} \end{bmatrix} + \begin{bmatrix} \mathbf{K}'^{-1}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'^{-1}_{12} \end{bmatrix} \begin{bmatrix} \mathbf{B}'^T_{11} \\ \mathbf{B}'^T_{12} \end{bmatrix} \mathbf{S}'^{-1}_1 \begin{bmatrix} \mathbf{B}'_{11} & \mathbf{B}'_{12} \end{bmatrix} \begin{bmatrix} \mathbf{K}'^{-1}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'^{-1}_{12} \end{bmatrix} & -\begin{bmatrix} \mathbf{K}'^{-1}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'^{-1}_{12} \end{bmatrix} \begin{bmatrix} \mathbf{B}'^T_{11} \\ \mathbf{B}'^T_{12} \end{bmatrix} \mathbf{S}'^{-1}_1 \\ -\mathbf{S}'^{-1}_1 \begin{bmatrix} \mathbf{B}'_{11} & \mathbf{B}'_{12} \end{bmatrix} \begin{bmatrix} \mathbf{K}'^{-1}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'^{-1}_{12} \end{bmatrix} & \mathbf{S}'^{-1}_1 \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} \mathbf{K}'^{-1}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'^{-1}_{12} \end{bmatrix} + \begin{bmatrix} \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11} \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12} \end{bmatrix} \mathbf{S}'^{-1}_1 \begin{bmatrix} \mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \end{bmatrix} & -\begin{bmatrix} \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11} \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12} \end{bmatrix} \mathbf{S}'^{-1}_1 \\ -\mathbf{S}'^{-1}_1 \begin{bmatrix} \mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \end{bmatrix} & \mathbf{S}'^{-1}_1 \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} \mathbf{K}'^{-1}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'^{-1}_{12} \end{bmatrix} + \begin{bmatrix} \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11} \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12} \end{bmatrix} \begin{bmatrix} \mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \end{bmatrix} & -\begin{bmatrix} \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1 \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1 \end{bmatrix} \\ -\begin{bmatrix} \mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \end{bmatrix} & \mathbf{S}'^{-1}_1 \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} \mathbf{K}'^{-1}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}'^{-1}_{12} \end{bmatrix} + \begin{bmatrix} \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \end{bmatrix} & -\begin{bmatrix} \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1 \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1 \end{bmatrix} \\ -\begin{bmatrix} \mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \end{bmatrix} & \mathbf{S}'^{-1}_1 \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} \mathbf{K}'^{-1}_{11} + \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{K}'^{-1}_{12} + \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \end{bmatrix} & -\begin{bmatrix} \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1 \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1 \end{bmatrix} \\ -\begin{bmatrix} \mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} \end{bmatrix} & \mathbf{S}'^{-1}_1 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{K}'^{-1}_{11} + \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} & -\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_1 \\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{K}'^{-1}_{12} + \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} & -\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_1 \\ -\mathbf{S}'^{-1}_1\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & -\mathbf{S}'^{-1}_1\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} & \mathbf{S}'^{-1}_1 \end{bmatrix}.
\end{aligned} \tag{5.20}$$

Przemnażając ostatnią macierz z równ. (5.20) obustronnie, tak jak podano poniżej

$$\begin{bmatrix}\mathbf{B}'_{31} & \mathbf{B}'_{32} & \mathbf{B}'_{33}\end{bmatrix}\left[\quad\cdot\quad\right]\begin{bmatrix}\mathbf{B}'^T_{31}\\ \mathbf{B}'^T_{32}\\ \mathbf{B}'^T_{33}\end{bmatrix}, \tag{5.21}$$

otrzymamy

$$\begin{aligned}
&\begin{bmatrix}\mathbf{B}'_{31} & \mathbf{B}'_{32} & \mathbf{B}'_{33}\end{bmatrix}\begin{bmatrix}\mathbf{K}'^{-1}_{11}+\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} & -\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & \mathbf{K}'^{-1}_{12}+\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} & -\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\\ -\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11} & -\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12} & \mathbf{S}'^{-1}_{1}\end{bmatrix}\begin{bmatrix}\mathbf{B}'^T_{31}\\ \mathbf{B}'^T_{32}\\ \mathbf{B}'^T_{33}\end{bmatrix}\\
&=\begin{bmatrix}\mathbf{B}'_{31} & \mathbf{B}'_{32} & \mathbf{B}'_{33}\end{bmatrix}\begin{bmatrix}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}+-\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{B}'^T_{33}\\ \mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}+\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}+-\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{B}'^T_{33}\\ -\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}-\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}+\mathbf{S}'^{-1}_{1}\mathbf{B}'^T_{33}\end{bmatrix}\\
&=\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}\\
&\quad-\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{B}'^T_{33}+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}\\
&\quad+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}-\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{B}'^T_{33}\\
&\quad-\mathbf{B}'_{33}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}-\mathbf{B}'_{33}\mathbf{S}'^{-1}_{1}\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}+\mathbf{B}'_{33}\mathbf{S}'^{-1}_{1}\mathbf{B}'^T_{33}.
\end{aligned} \tag{5.22}$$

Teraz można przegrupować ostatnie równanie i wyłączyć przed nawias wspólne czynniki:

$$\begin{aligned}
&\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}\\
&\quad+\qquad\mathbf{B}'_{33}\mathbf{S}'^{-1}_{1}(-\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}-\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}+\mathbf{B}'^T_{33})\\
&\quad+\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}(\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}-\mathbf{B}'^T_{33})\\
&\quad+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}(\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}-\mathbf{B}'^T_{33})\\
&=\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}\\
&\quad+\qquad\mathbf{B}'_{33}\mathbf{S}'^{-1}_{1}(\mathbf{B}'^T_{33}-\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}-\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32})\\
&\quad-\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}(\mathbf{B}'^T_{33}-\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}-\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32})\\
&\quad-\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}(\mathbf{B}'^T_{33}-\mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}-\mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}).
\end{aligned} \tag{5.23}$$

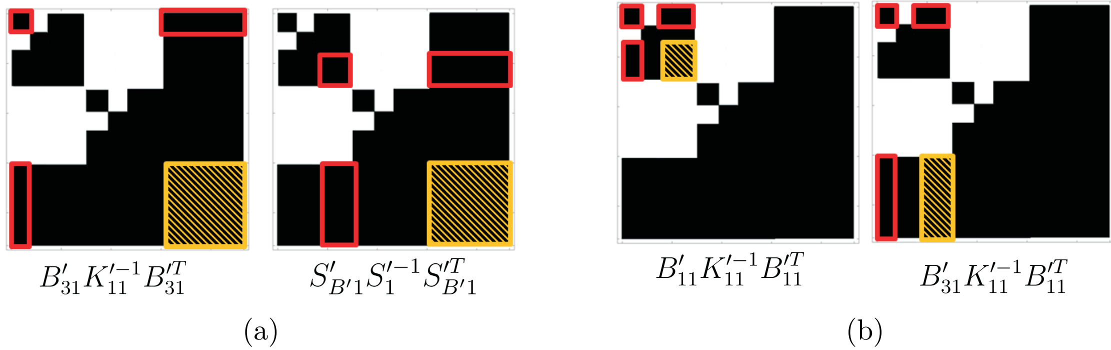
Następnie można zauważyć, że w każdym nawiasie jest ten sam czynnik, który został oznaczony poprzez $\mathbf{S}'^T_{B'1}$ i znów można wyłączyć wspólny czynnik przed nawias:

$$\begin{aligned}
&\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}\\
&\quad+\mathbf{B}'_{33}\mathbf{S}'^{-1}_{1}\mathbf{S}'^T_{\mathbf{B}'1}-\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}\mathbf{S}'^{-1}_{1}\mathbf{S}'^T_{\mathbf{B}'1}-\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12}\mathbf{S}'^{-1}_{1}\mathbf{S}'^T_{\mathbf{B}'1}\\
&=\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}+(\mathbf{B}'_{33}-\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{11}-B'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{12})\mathbf{S}'^{-1}_{1}\mathbf{S}'^T_{B'1}\\
&=\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}+\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}+\mathbf{S}'_{B'1}\mathbf{S}'^{-1}_{1}\mathbf{S}'^T_{B'1}.
\end{aligned} \tag{5.24}$$

Gdy analogiczne rozumowanie zostanie powtórzone dla macierzy $\mathbf{K}'^{-1}_2$, to równ. (5.17) na macierz dla interfejsów dla macierzy $\mathbf{K}'$ można przekształcić do postaci:

$$\begin{aligned}
\mathbf{S}'&=\mathbf{C}'_3-\mathbf{B}'_1\mathbf{K}'^{-1}_1\mathbf{B}'^T_1-\mathbf{B}'_2\mathbf{K}'^{-1}_2\mathbf{B}'^T_2\\
&=\mathbf{C}'_3-\mathbf{B}'_{31}\mathbf{K}'^{-1}_{11}\mathbf{B}'^T_{31}-\mathbf{B}'_{32}\mathbf{K}'^{-1}_{12}\mathbf{B}'^T_{32}-\mathbf{S}'_{B'1}\mathbf{S}'^{-1}_{1}\mathbf{S}'^T_{B'1}\\
&\quad-\mathbf{B}'_{41}\mathbf{K}'^{-1}_{21}\mathbf{B}'^T_{41}-\mathbf{B}'_{42}\mathbf{K}'^{-1}_{22}\mathbf{B}'^T_{42}-\mathbf{S}'_{B'2}\mathbf{S}'^{-1}_{2}\mathbf{S}'^T_{B'2}.
\end{aligned} \tag{5.25}$$

Rysunki 5.10a oraz 5.10b przedstawiają odpowiednie części równ. (5.25), aby zobrazować, które części macierzy dla interfejsów odpowiadają poszczególnym wyrażeniom z równ. (5.25).



Rysunek 5.10: Wizualizacja macierzy dla interfejsów oraz części uzupełnienia Schura i ich odpowiedników: (a) z równ. (5.25); (b) z równ. (5.23) i (5.24).

Aby otrzymać rozwiązanie zostanie wykonane następujące rozumowanie. Trzeba rozwiązać równanie:

$$\mathbf{S}\mathbf{x} = \mathbf{b}, \quad \text{gdzie:} \quad \mathbf{x} = [\mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3]^T, \quad \mathbf{b}' = [\mathbf{b}'_1 \quad \mathbf{b}'_2 \quad \mathbf{b}'_3]^T, \tag{5.26}$$

a $\mathbf{S}$ zdefiniowane jest w równ. (5.15). Macierz interfejsów dla macierzy $\mathbf{S}$ została wyznaczona i wynosi $\mathbf{S}'$ wg równ. (5.17). Aby otrzymać równanie dla interfejsów dla równ. (5.26), trzeba wyznaczyć prawą stronę równania dla interfejsów, która ma postać:

$$\mathbf{b}'_3 - \mathbf{B}'_3\mathbf{K}'^{-1}_1\mathbf{b}'_1 - \mathbf{B}'_4\mathbf{K}'^{-1}_1\mathbf{b}'_2. \tag{5.27}$$

Dla tego wzoru trzeba wyznaczyć $\mathbf{K}'^{-1}_1\mathbf{b}'_1 = \mathbf{z}_1$ i w tym celu należy rozwiązać równanie:

$$\mathbf{K}'_1\mathbf{z}_1 = \mathbf{b}'_1, \tag{5.28}$$

gdzie

$$\mathbf{K}'_1 = \begin{bmatrix} \mathbf{K}'_{11} & \mathbf{0} & \mathbf{B}'^T_{11} \\ \mathbf{0} & \mathbf{K}'_{12} & \mathbf{B}'^T_{12} \\ \mathbf{B}'_{11} & \mathbf{B}'_{12} & \mathbf{C}'_1 \end{bmatrix}, \qquad \mathbf{z}_1 = \begin{bmatrix} \mathbf{z}_{11} \\ \mathbf{z}_{12} \\ \mathbf{z}_{13} \end{bmatrix}, \qquad \mathbf{b}'_1 = \begin{bmatrix} \mathbf{b}'_{11} \\ \mathbf{b}'_{12} \\ \mathbf{b}'_{13} \end{bmatrix}. \tag{5.29}$$

Dla macierzy $\mathbf{K}'_1$ wyznaczono $\mathbf{S}'_1$ w równ. (5.17). Aby wyznaczyć prawe strony równania interfejsów dla macierzy $\mathbf{K}'_1$ należy rozwiązać równanie:

$$\mathbf{S}'_1\mathbf{z}_{13} = \mathbf{b}'_{13} - \mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{b}'_{11} - \mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{b}'_{11}. \tag{5.30}$$

Następnie trzeba rozwiązać równania:

$$\mathbf{K}'_{11}\mathbf{z}_{11} = \mathbf{b}'_{11} - \mathbf{B}'^T_{11}\mathbf{z}_{13}, \qquad \mathbf{K}'_{12}\mathbf{z}_{12} = \mathbf{b}'_{12} - \mathbf{B}'^T_{12}\mathbf{z}_{13}. \tag{5.31}$$

Stąd $\mathbf{z}_{11}, \mathbf{z}_{12}, \mathbf{z}_{13}$, co daje $\mathbf{z}_1$. Analogicznie można wyznaczyć $\mathbf{z}_2$, a mając $\mathbf{z}_1$ oraz $\mathbf{z}_2$ można wyznaczyć prawą stronę dla równania interfejsów z równ. (5.27). Należy pamiętać,

że macierze w równ. (5.30) oraz w równ. (5.31) zostały sfaktoryzowane na wcześniejszym etapie i teraz wystarczy wykonywać podstawienie wstecz/wprzód. Można przystąpić do rozwiązania równania dla interfejsów:

$$\mathbf{S}'\mathbf{x}_3 = \mathbf{b}_3 - \mathbf{B}'_3\mathbf{z}_1 - \mathbf{B}'_4\mathbf{z}_2. \tag{5.32}$$

Mając $\mathbf{x}_3$ można rozwiązać równanie:

$$\mathbf{K}'_1\mathbf{x}_1 = \mathbf{b}_1 - \mathbf{B}'^T_3\mathbf{x}_3 = \mathbf{c}_1, \tag{5.33}$$

gdzie: $\mathbf{c}_1 = [\mathbf{c}_{11} \quad \mathbf{c}_{12} \quad \mathbf{c}_{13}]^T$, $\mathbf{x}_1 = [\mathbf{x}_{11} \quad \mathbf{x}_{12} \quad \mathbf{x}_{13}]^T$. Ponownie uzupełnienie Schura dla macierzy $\mathbf{K}'_1$ otrzyma się z równ. (5.17) i jest równe $\mathbf{S}'_1$. Następnie należy wyznaczyć prawą stronę oraz rozwiązać równanie dla interfejsów:

$$\mathbf{S}'_1\mathbf{x}_{13} = \mathbf{c}_{13} - \mathbf{B}'_{11}\mathbf{K}'^{-1}_{11}\mathbf{c}_{11} - \mathbf{B}'_{12}\mathbf{K}'^{-1}_{12}\mathbf{c}_{12}. \tag{5.34}$$

Znając $\mathbf{x}_{13}$ można wyznaczyć $\mathbf{x}_{11}$ oraz $\mathbf{x}_{12}$ z równań:

$$\mathbf{K}'_{11}\mathbf{x}_{11} = \mathbf{c}_{11} - \mathbf{B}'^T_{11}\mathbf{x}_{13}, \qquad \mathbf{K}'_{12}\mathbf{x}_{12} = \mathbf{c}_{12} - \mathbf{B}'^T_{12}\mathbf{x}_{13}. \tag{5.35}$$

Znając $\mathbf{x}_{11}, \mathbf{x}_{12}$ oraz $\mathbf{x}_{13}$ wyznacza się $\mathbf{x}_1$. Analogicznie można wyznaczyć $\mathbf{x}_2$. Stąd będzie znane $\mathbf{x}_1$, $\mathbf{x}_2$ oraz $\mathbf{x}_3$, czyli $\mathbf{x}$ dla równ. (5.26).

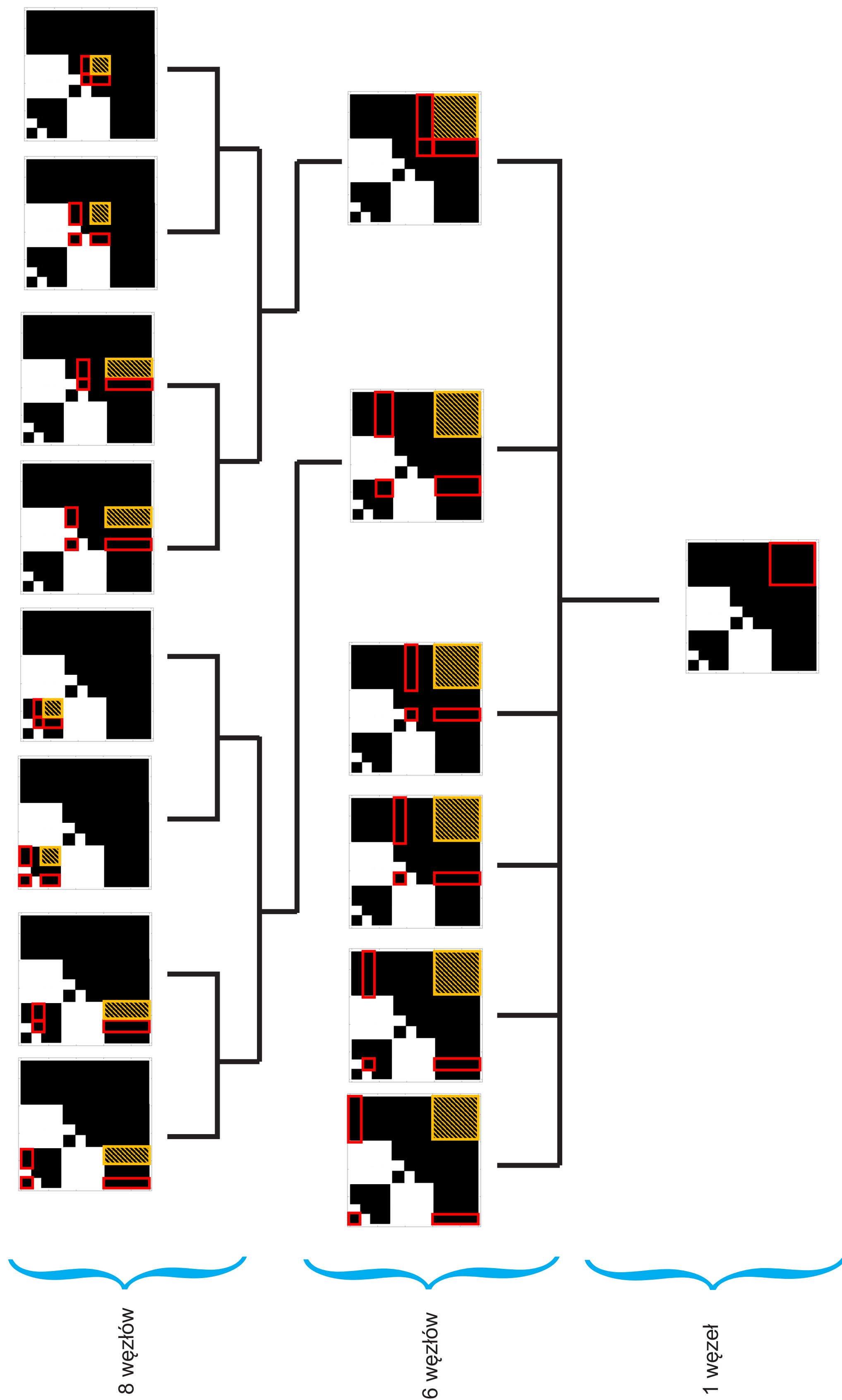
5.3.4 Hierarchiczna faktoryzacja uzupełnienia Schura na wielu węzłach

W poprzednim podrozdziale wyprowadzono odpowiednie wzory, które mogą posłużyć do faktoryzacji uzupełnienia Schura. W tym podrozdziale zostanie omówiony podział odpowiednich zadań na węzły, tak aby zrównoważyć pracę każdego z nich.

Na podstawie równ. (5.25) widać, że trzeba wykonać cztery częściowe faktoryzacje związane z członami $\mathbf{B}'_{2+ij}\mathbf{K}'^{-1}_{ij}\mathbf{B}'^T_{2+ij}$ oraz dwie częściowe faktoryzacje związane z członami $\mathbf{S}'_{B'i}\mathbf{S}'^{-1}_i\mathbf{S}'^T_{B'i}$. Ale żeby móc wykonać częściową faktoryzację członu $\mathbf{S}'_{B'i}\mathbf{S}'^{-1}_i\mathbf{S}'^T_{B'i}$, z równ. (5.16) oraz równ. (5.24) należy wykonać jeszcze cztery częściowe faktoryzacje członów $\mathbf{B}'_{ij}\mathbf{K}'^{-1}_{ij}\mathbf{B}'^T_{ij}$ oraz cztery wyznaczenia członów $\mathbf{B}'_{2+ij}\mathbf{K}'^{-1}_{ij}\mathbf{B}'^T_{ij}$. Ostatnie cztery człony będą wyznaczone w tradycyjny sposób tzn. metodą M1 (patrz rozdz. 5.2.2), a nie za pomocą częściowej faktoryzacji. Trzeba tak uczynić, ze względu na fakt, że człon $\mathbf{B}'_{2+ij}\mathbf{K}'^{-1}_{ij}\mathbf{B}'^T_{ij}$ nie tworzy macierzy symetrycznej.

Powstają, więc cztery rodzaje zadań:

1. częściowa faktoryzacja członów $\mathbf{B}'_{2+ij}\mathbf{K}'^{-1}_{ij}\mathbf{B}'^T_{2+ij}$,
2. częściowa faktoryzacja członów $\mathbf{B}'_{ij}\mathbf{K}'^{-1}_{ij}\mathbf{B}'^T_{ij}$,
3. wyliczenie członów $\mathbf{B}'_{2+ij}\mathbf{K}'^{-1}_{ij}\mathbf{B}'^T_{ij}$,
4. częściowa faktoryzacja członów $\mathbf{S}'_{B'i}\mathbf{S}'^{-1}_i\mathbf{S}'^T_{B'i}$.

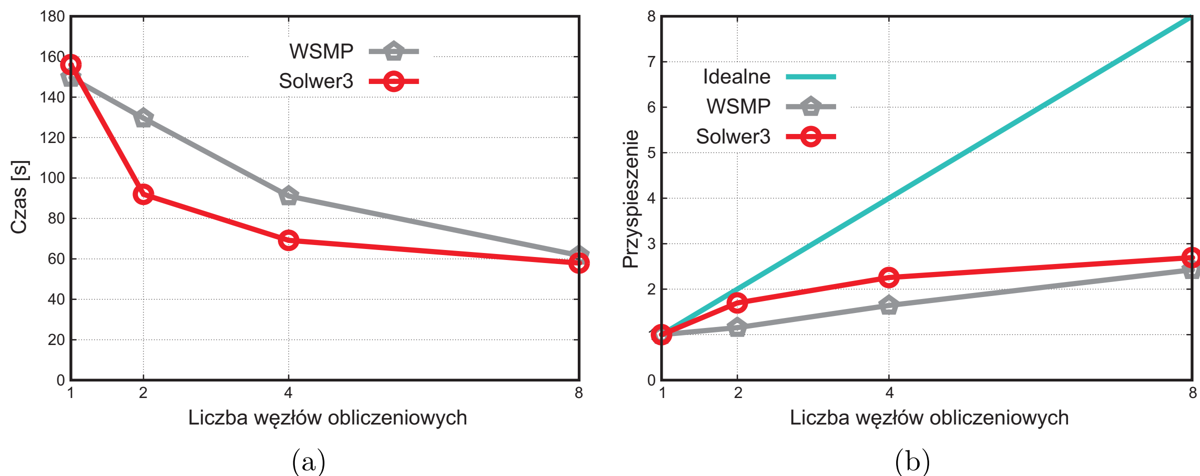


Rysunek 5.11: Schematyczne przedstawienie obliczeń za pomocą hierarchicznej faktoryzacji.

Kroki 1 oraz 3 są potrzebne do wykonania kroku 4, ale mają podobny koszt obliczeniowy, więc zostaną wykonane jako pierwsze na ośmiu węzłach. Kroki 3 oraz 4 również mogą być wykonane niezależnie, dodatkowo krok 4 jest droższy obliczeniowo, więc koszt kroku 3 może być ukryty w trakcie wykonywania kroku 4, co zostanie zrobione na 6 węzłach. Na końcu jeden węzeł bedzie mógł wykonać faktoryzację macierzy dla interfejsów $\mathbf{S}'$ dla macierzy interfejsowej $\mathbf{S}$. Rysunek 5.11 przedstawia schematycznie powyższy algorytm.

Na rys. 5.12a pokazano czas wykonania faktoryzacji w zależności od liczby węzłów. Czas dla własnego solwera oznaczonego jako Solwer3 jest krótszy niż czas WSMP. Rysunek 5.12b przedstawia skalowalność i można zauważyć, że Solwer3 wykazuje trochę lepszą skalowalność niż WSMP, wynosi ona ok. 2.69 razy dla 8 węzłów.

Tabela 5.9 pokazuje wykorzystanie pamięci przez oba solwery oraz różnicę procentową między zapotrzebowaniem na pamięć solwera WSMP a Solwera3. Widać, że Solwer3 potrzebuje mniej pamięci niż WSMP; dla 8 węzłów o 14% mniej.



Rysunek 5.12: Czas wykonania (a) i przyspieszenie (b) etapu faktoryzacji. Test kostki $N = 64$.

Tabela 5.9: Maksymalne użycie pamięci na jednym weźle podczas faktoryzacji. Test kostki $N = 64$.

Liczba węzłów	Wykorzystanie pamięci[GB] WSMP	Solwer3	Różnica [%]
1	20.47	15.45	32.5
2	16.61	10.55	57.4
4	9.65	6.23	54.9
8	5.72	5.03	13.7

5.3.5 Solwer hierarchiczny z mieszaną precyzją na wielu węzłach

W poprzednim podrozdziale pokazano, że pomimo zastosowania różnych technik, faktoryzacja macierzy dla interfejsów nadal jest barierą dla skalowalych obliczeń na wielu węzłach. Problem ten dostrzeżono także w innych pracach np. w [117], gdzie próbowano go ominąć poprzez stosowanie metody iteracyjnej do rozwiązania układu dla interfejsów. Ze względu na fakt, że w metodach iteracyjnych wykonuje się tylko mnożenia macierzy przez wektor, nie trzeba tworzyć jawnie całej macierzy interfejsowej. Można więc wykonać mnożenie wektora przez lokalne uzupełnienia Schura, dla każdej subdomeny z osobna, a na końcu zsumować wyniki. Takie podejście opisano także w rozdz. 5.4 niniejszej pracy.

W rozdz. 4.4.2 pokazano, że użycie mieszanej precyzji i iteracyjnego poprawiania przyspiesza działania solwera na jednym węźle i zachowuje odpowiednią dokładność. W pracach [32] oraz [28] mieszaną precyzję stosowano do przyspieszania obliczeń dla dekompozycji obszaru, jednak zastosowano tam metodę iteracyjną, gdzie preconditioner był w podwójnej precyzji a pozostałe kroki iteracji w pojedynczej precyzji. W niniejszej pracy stosuje się mieszaną precyzję inaczej; po zbudowaniu macierzy dla interfejsów w podwójnej precyzji, zostaje ona sfaktoryzowana w pojedynczej precyzji i aby uzyskać odpowiednią dokładność wykonane zostanie iteracyjne poprawianie w podwójnej precyzji.

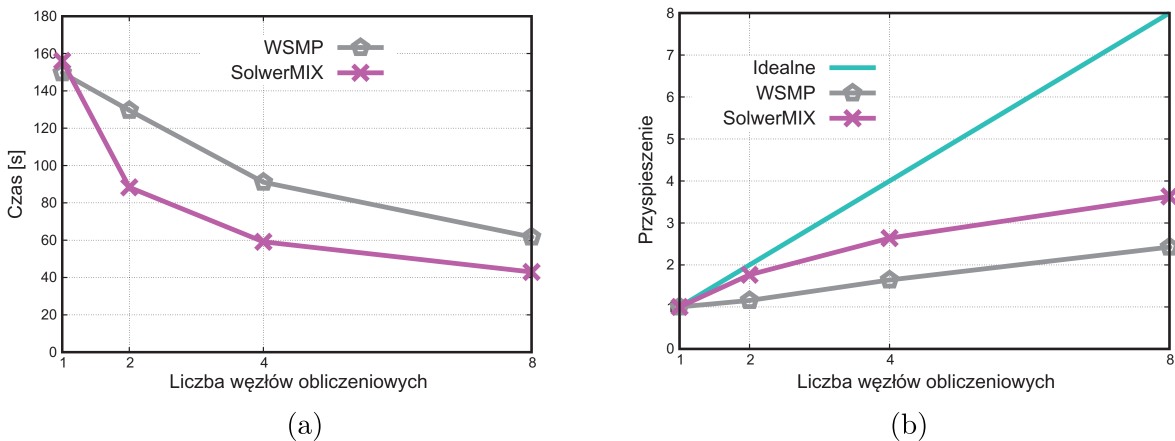
Dodatkową motywacją do zastosowania iteracyjnego poprawiania dla układu równań dla interfejsów, jest fakt, że uzupełnienie Schura ma mniejszy wskaźnik uwarunkowania niż cała macierz [9], a iteracyjne poprawianie dobrze działa dla macierzy, których wskaźnik uwarunkowania nie jest zbyt duży, patrz [77]. Całe podejście przedstawia następujący algorytm (oznaczony jako SolwerMIX):

1. Każdy węzeł otrzymuje od węzła „0” macierz $\bar{\mathbf{K}}_i$,
2. Każdy węzeł częściowo faktoryzuje macierz $\bar{\mathbf{K}}_i$,
3. Każdy węzeł wylicza uzupełnienie Schura dla prawej strony,
4. Węzeł „0” otrzymuje uzupełnienia Schura od wszystkich węzłów,
5. Węzeł „0” rozwiązuje układ równań dla interfejsów z równ. (5.10):
   (a) faktoryzacja jest wykonana w pojedynczej precyzji z wykorzystaniem Solwera3,
   (b) rozwiązanie jest w podwójnej precyzji z wykorzystaniem iteracyjnego poprawiania, patrz rozdz. 4.4.2 ,
6. Każdy węzeł otrzymuje od węzła „0” rozwiązanie układu równań dla interfejsów,
7. Każdy węzeł poprawia prawą stronę równ. (5.12),
8. Każdy węzeł rozwiązuje równ. (5.12).

W kroku 5 wykorzystano Solwer3, który stosuje hierarchiczną faktoryzację opisaną w rozdz. 5.3.3.

Porównano własny solwer na klaster oznaczony jako SolwerMIX z rozproszonym solwerem WSMP wykorzystując dwa testy: test kostki (patrz rozdz. 3.4.1) oraz test dla powłoki (patrz rozdz. 3.4.2).

Test kostki Na rys. 5.13a pokazano czas wykonania faktoryzacji w zależności od liczby węzłów. Czas dla SolweraMIX jest krótszy o ok. 46% niż czas WSMP dla 8 węzłów. Rysunek 5.13b przedstawia przyspieszenie i można zauważyć, że SolwerMIX wykazuje lepszą skalowalność niż WSMP, przyspieszenie wynosi ok. 3.63 razy dla 8 węzłów; oba odbiegają od idealnego przyspieszenia.



(a) (b)

Rysunek 5.13: Czas wykonania (a) i przyspieszenie (b) fazy faktoryzacji. Test kostki $N = 64$.

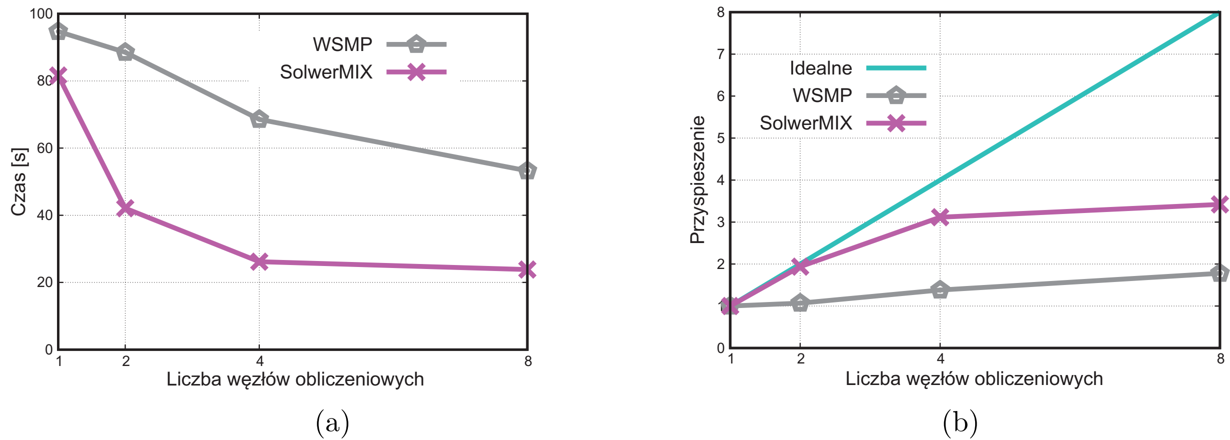
Tabela 5.10 pokazuje wykorzystanie pamięci przez oba solwery oraz różnicę względną zapotrzebowania na pamięć WSMP i SolwerMIX. Widać, że SolwerMIX potrzebuje mniej pamięci niż WSMP; dla 8 węzłów o ok. 39% mniej.

Tabela 5.10: Maksymalne użycie pamięci na jednym weźle podczas faktoryzacji. Test kostki dla $N = 64$.

Liczba	Wykorzystanie pamięci[GB]		Różnica
węzłów	WSMP	SolwerMIX	względna [%]
1	20.47	15.45	32.5
2	16.61	10.05	65.3
4	9.65	5.64	71.1
8	5.72	4.13	38.5

Test dla powłoki $N = 212$. W tym przypadku obliczono test z rodz. 3.4.2 dla siatki $N \times N \times 10$, gdzie $N = 212$, co daje ok. 1.5 miliona niewiadomych. Na rys. 5.14a pokazano czas wykonania faktoryzacji w zależności od liczby węzłów. Czas dla SolwerMIX jest krótszy ok. 2 razy niż czas WSMP dla 8 węzłów. Rysunek 5.14b przedstawia skalowalność i można zauważyć, że SolwerMIX wykazuje lepszą skalowalność niż WSMP; uzyskano przyspieszenie ok. 3.42 razy dla 8 węzłów.

Tabela 5.11 pokazuje wykorzystanie pamięci przez oba solwery oraz różnicę względną zapotrzebowania na pamięć WSMP i SolwerMIX. Widać, że SolwerMIX potrzebuje mniej pamięci niż WSMP; dla 8 węzłów o ok. 38% mniej.



(a) (b)

Rysunek 5.14: Czas wykonania (a) i przyspieszenie (b) fazy faktoryzacji. Test powłoki $N = 212$.

Tabela 5.11: Maksymalne użycie pamięci na jednym weźle podczas faktoryzacji. Test powłoki $N = 212$.

Liczba węzłów	Wykorzystanie pamięci[GB] WSMP	SolwerMIX	Różnica względna [%]
1	22.42	18.35	22.2
2	14.38	11.86	21.2
4	8.84	6.69	32.1
8	5.92	4.28	38.3

5.4 Iteracyjne rozwiązywanie układów równań na wielu węzłach

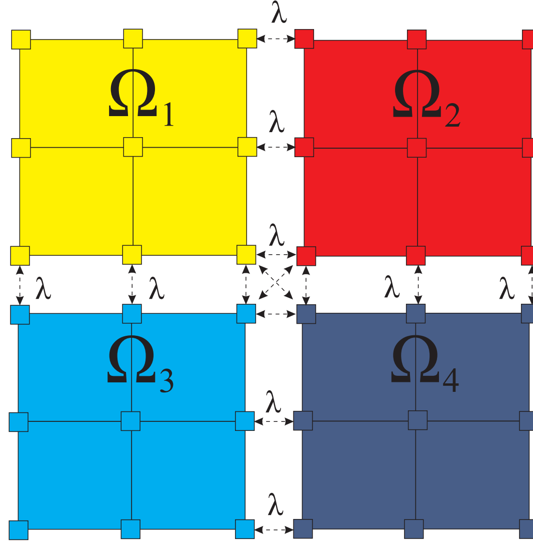
Głównym problemem przy rozwiązywaniu układów równań na wielu węzłach obliczeniowych, jest długi czas wykonywania faktoryzacji macierzy interfejsowej. Dlatego powstały metody mieszane, które faktoryzują macierze subdomenowe za pomocą solwera bezpośredniego, a układ równań dla interfejsów rozwiązują za pomocą metody iteracyjnej.

W tym rozdziale zostanie przedstawiona metoda FETI (ang. *Finite Element Tearing and Interconnecting*), której rozszerzenie FETI-DP (ang. *Dual Primal*) jest właściwą metodą dla konstrukcji powłokowych, które są w kręgu zainteresowań autora niniejszej rozprawy. Po przedstawieniu głównej idei metody FETI, zostaną przedstawione wnioski, które autor pracy zebrał w czasie implementacji tej metody.

5.4.1 Wprowadzenie do FETI

Metoda FETI jest bardzo podobna do metody z wykorzystaniem uzupełniena Schura, ale zamiast wyrugowania zmiennych dla subdomen, aby uzyskać macierz interfejsową, stosuje się mnożniki Lagrange'a aby powiązać zmienne interfejsowe. Po raz pierwszy metoda została zaprezentowana w artykule [46]. Główna idea metody FETI została pokazana na

rys. 5.15. Nie będą teraz rozpatrywane tzw. „pływające” subdomeny, zostaną one opisane w dalszej części tego rozdziału. Przyjęto więc założenie, że dla każdej subdomeny i stworzona dla niej macierz sztywności $\mathbf{K}_i$ jest odwracalna. Przez $\mathbf{u}_i$ będą rozumiane zmienne dla całej i-tej subdomeny, włączając w to zmienne interfejsowe. Niech macierz



Rysunek 5.15: Schematyczne przedstawienie metody FETI, interfejsy połączone za pomocą mnożników Lagrange'a.

$\mathbf{B}_i$ będzie macierzą boolowską, która wyznacza, jakie zmienne z wektora $\mathbf{u}_i$ są zmiennymi interfejsowymi z odpowiednim znakiem:

$$|\mathbf{B}_i|\mathbf{u}_i = \mathbf{u}^I, \tag{5.36}$$

gdzie przez $\mathbf{u}^I$ oznaczono zmienne interfejsowe. Dzięki temu można zapisać warunek ciągłości na interfejsach w następujący sposób:

$$\sum_{i=1}^{n} \mathbf{B}_i\mathbf{u}_i = \mathbf{0}, \tag{5.37}$$

a układ równań dla subdomen następująco:

$$\mathbf{K}_i\mathbf{u}_i = \mathbf{f}_i + \mathbf{B}_i^T\boldsymbol{\lambda}, \tag{5.38}$$

gdzie $\boldsymbol{\lambda}$ to mnożniki Lagrange'a, które wymuszają warunek ciągłości przemieszczeń na interfejsach. Jeśli wyrugowane zostaną zmienne $\mathbf{u}_i$ z równ. (5.38) i zostaną wstawione do równ. (5.37) to uzyska się:

$$\underbrace{(-\sum_{i=1}^{n} \mathbf{B}_i\mathbf{K}_i^{-1}\mathbf{B}_i^T)}_{\mathbf{F}}\,\boldsymbol{\lambda} = \underbrace{\sum_{i=1}^{n} \mathbf{B}_i^T\mathbf{K}_i^{-1}\mathbf{f}_i}_{\mathbf{g}}, \tag{5.39}$$

Pod pewnymi warunkami (patrz [46]), można wykazać, że macierz $\mathbf{F}$ jest sumą odwrotności uzupełnień Schura dla subdomen. Warto zaznaczyć, że macierz $\mathbf{B}_i$ jest operatorem

boolowskim, a to oznacza, że wynik jej zastosowania do macierzy lub wektora, powinien być interpretowany jako proces selekcji, a nie mnożenia macierzy przez macierz lub macierzy przez wektor.

W oryginalej metodzie FETI, równ. (5.39) jest rozwiązane za pomocą metody iteracyjnej, a dokładniej metody PCG (ang. *Preconditioned Conjugate Gradient*). Wykorzystując metodę iteracyjną, nie trzeba jawnie tworzyć macierzy $\mathbf{F}$, ponieważ potrzebny jest tylko wynik mnożenia macierzy $\mathbf{F}$ przez wektor. Takie mnożenie może być wykonane równolegle na każdej subdomenie z osobna a wynik może być użyty w następnej iteracji.

W metodach iteracyjnych często stosuje się tzw. preconditioner, czyli macierz przez którą mnożona jest macierz wejściowa i dobraną tak, aby uzyskać macierz o mniejszym wskaźniku uwarunkowania, co prowadzi do mniejszej liczby iteracji w procesie iteracyjnego rozwiązywania układu. W przypadku metody FETI zazwyczaj stosuje się dwa preconditionery:

1. preconditioner Dirichlet'a (ang. *Dirichlet preconditioner*), tzn. wykorzystuje się uzupełnienie Schura $\mathbf{S}_i$ subdomeny, tak jak zostało opisane w rozdz. 5.2.2. W pracy [48] pokazano, że taki preconditioner jest optymalny.

2. obcięty preconditioner (ang. *lumped preconditioner*), gdy wykorzystuje się macierz $\mathbf{C}_i$. Największą jego zaletą jest niski koszt obliczeniowy, ponieważ macierz ta jest już stworzona na każdej subdomenie. Warto zauważyć, że obcięty preconditioner to zredukowany preconditioner Dirichlet'a, gdyż wykorzystuje on tylko pierwszy człon $\mathbf{S}_i$ z równ. (5.7).

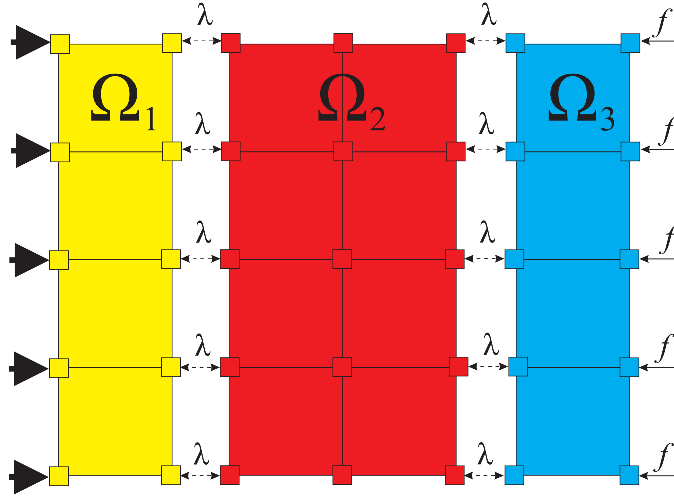
FETI „pływające" subdomeny. Przykładowy podział domeny, w którym wystąpiły „pływające" subdomeny (ang. *floating subdomains*) został pokazany na rys. 5.16, gdzie Ω_2 to „pływająca" subdomena. Subdomena ta jest „pływająca", ponieważ nie zostały ustalone żadne warunki brzegowe ani siły działające na nią. Macierz $\mathbf{K}_k$, która została zbudowana dla k-tej „pływającej" subdomeny jest osobliwa. Można skonstruować macierz $\mathbf{R}_k$, której kolumny to baza jądra dla macierzy $\mathbf{K}_k$ (6 kolumn dla problemów 3D i 3 dla problemów 2D). Dla macierzy $\mathbf{K}_k$ zdefiniuje się macierz $\mathbf{K}_k^+$, tzw. macierz pseudoodwrotną, tzn. taką, która spełnia warunek $\mathbf{K}_k\mathbf{K}_k^+\mathbf{K}_k = \mathbf{K}_k$. Teraz można obliczyć rozwiązanie dla $\mathbf{u}_k = \mathbf{K}_k^+(\mathbf{f}_k + \mathbf{B}_k^T\boldsymbol{\lambda}) + \mathbf{R}_k\boldsymbol{\alpha}$. Ostatnie równanie ma rozwiązanie wtedy i tylko wtedy gdy $\mathbf{f}_k + \mathbf{B}_k^T\boldsymbol{\lambda}$ nie ma elementów z jądra $\mathbf{K}_k$, tzn.:

$$\mathbf{R}_k^T(\mathbf{f}_k + \mathbf{B}_k^T\boldsymbol{\lambda}) = \mathbf{0} \tag{5.40}$$

Fizycznie $\mathbf{R}_k$ reprezentuje ruchy sztywne ciała Ω_k, a $\boldsymbol{\lambda}$ określa ich liniową kombinację. Teraz można połączyć równ. (5.39) oraz równ. (5.40), z których wynika:

$$\begin{bmatrix} \mathbf{F} & -\mathbf{B}_k\mathbf{R}_k \\ -\mathbf{B}_k^T\mathbf{R}_k^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ -\mathbf{R}_k^T\mathbf{f}_k \end{bmatrix}, \tag{5.41}$$

gdzie $\mathbf{F}$ i $\mathbf{g}$ są zdefiniowane w ten sam sposób jak w równ. (5.39), ale dla „pływającej" subdomeny k używa się macierzy pseudoodwrotnej $\mathbf{K}_k^+$.



Rysunek 5.16: Subdomena Ω_2 jest subdomeną „pływającą".

Jednak gdy zostanie użyta metoda iteracyjna dla problemów MES, szczególnie płytowych i powłokowych, to wskaźnik uwarunkowania macierzy interfejsowej rośnie bardzo szybko wraz z liczbą elementów w subdomenie [47]. Aby temu zapobiec w pracy [48] zaproponowano metodę dwupoziomową FETI 2 (ang. *two-level FETI method*), która jest opisana w następnym podrozdziale. Wadą metody FETI 2 jest poziom jej skomplikowania, dlatego powstała metoda FETI-DP (ang. *FETI Dual-Primal method*), w której opis matematyczny jest prostszy i eliminuje wady metody FETI. FETI-DP została opisana w kolejnym podrozdziale.

5.4.2 Wprowadzenie do FETI-DP

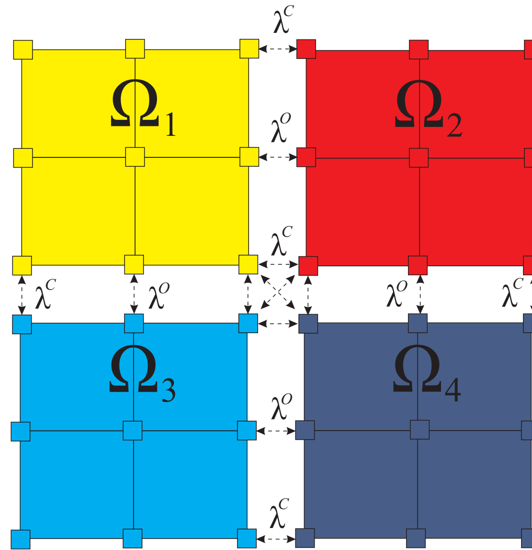
FETI 2. FETI 2 została zaprezentowana w pracy [48] jako rozwiązanie problemów z użyciem metody FETI dla zadań płytowych i powłokowych. Główny pomysł jest następujący: w procesie iteracyjnym (gdy rozwiązywany jest układ równań dla interfejsów) po każdej iteracji nie ma ciągłości na interfejsach, dlatego trzeba wybrać kilka miejsc gdzie będzie wymagana ciągłość również po każdej iteracji. Mnożniki Lagrange'a $\boldsymbol{\lambda}$ zostają podzielone na dwa zbiory: pierwszy, w którym ciągłość jest zachowana po każdej iteracji $\boldsymbol{\lambda}^C$, tzw. naroża; drugi, gdzie ciągłość zostaje osiągnięta dopiero po całym procesie iteracyjnym $\boldsymbol{\lambda}^O$.

$$\mathbf{C}^T(\sum_{i=1}^{n}\mathbf{B}_i\mathbf{u}_i) = \mathbf{0}, \tag{5.42}$$

gdzie $\mathbf{C}^T$ wybiera się tak, że:

$$\mathbf{C}^T\boldsymbol{\lambda} = \boldsymbol{\lambda}^C. \tag{5.43}$$

Na rysunku 5.17 przedstawiono intuicyjnie metodę FETI 2. Metoda jest nazywana dwupoziomową (ang. *two-level method*), ponieważ warunki ciągłości po każdej iteracji są wymuszane w ten sam sposób, jak warunki ciągłości dla całego problemu. Naroża są wybierane jako punkty wspólne dla co najmniej trzech subdomen się stykają, albo gdzie koncentrują się siły zewnętrzne.



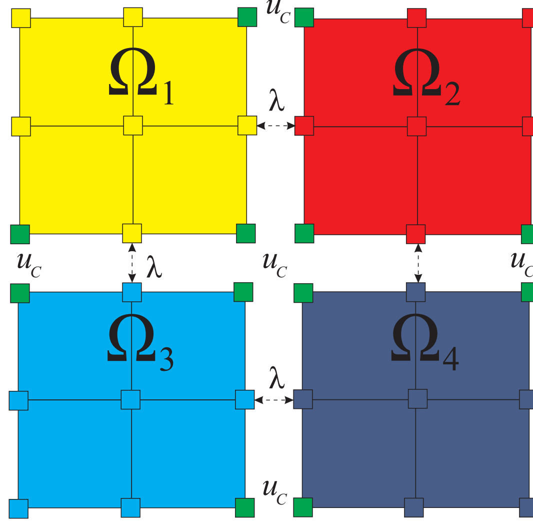
Rysunek 5.17: Intuicja metody FETI 2, interfejsy połączone za pomocą mnożników Lagrange'a, ale w narożach ciągłośc jest wymuszana po każdej iteracji.

Opis metody FETI 2 w pracy [48] sugeruje, że dla problemów powłokowych, ciągłość pola przemieszczeń normalnych do powierzchni powłoki powinno być wymuszone w narożach poprzez iteracje algorytmu PCG. Jeden ze sposobów implementacji tego wymagania i ominięcia trudności z definiowaniem normalnych do niegładkiej powierzchni powłoki wymaga wymuszania ciągłości wszystkich trzech komponentów przemieszczeń na narożach subdomeny. Te podejście automatycznie prowadzi do wymuszania także ciągłości normalnych pola przemieszczeń w narożach, z minimalnym zmianami implementacji metody FETI 2. Taka zmodyfikowana metoda FETI 2 dla problemów powłokowych jest nazywana metodą FETI 2-ACD, gdzie ACD (ang. *„all components of the displacement”*) oznacza wszystkie komponenty pola przemieszczeń w narożach. W pracy [48] bazując na intensywnych eksperymentach numerycznych pokazano, że metoda FETI 2-ACD wraz z preconditionerem Dirichlet'a jest optymalna dla problemów powłokowych.

Nie opisywano tutaj metody FETI 2 bardziej szczegółowo, ponieważ stanowi ona etap pośredni do stworzenia metody FETI-DP, która jest ulepszoną wersją FETI 2 a jej matematyczny opis jest prostszy.

FETI-DP. Główna idea metody FETI-DP (ang. *FETI Dual-Primal*) wywodzi się z metody FETI 2, ale zamiast wymuszania ciągłości na narożach za pomocą procesu iteracyjnego, naroża będą traktowane tak jak zmienne interfejsowe w metodzie z uzupełnieniem Schura, które nazywa się zmiennymi podstawowymi (ang. *primal*). Z kolei na pozostałej części interfejsu zostaną zastosowane mnożniki Lagrange'a, jak w oryginalnej metodzie FETI; te zmienne będą nazwane dualnymi (ang. *dual*). Użycie zmiennych podstawowych i dualnych tłumaczy nazwę metody.

Na rysunku 5.18 przedstawiono ideę metody FETI-DP. Metoda została opisana po raz pierwszy w pracy [49]. Do jej wyprowadzenia dla każdej subdomeny należy zbudować



Rysunek 5.18: Idea metody FETI-DP, interfejsy połączone za pomocą mnożników Lagrange'a, ale naroża sa traktowane jak interfejsy w metodzie z uzupełnieniem Schura.

następujący układ równań:

$$\begin{bmatrix} \mathbf{K}_i & \mathbf{L}_i^T \\ \mathbf{L}_i & \mathbf{C}_i \end{bmatrix} \begin{bmatrix} \mathbf{u}_i \\ \mathbf{u}^C \end{bmatrix} = \begin{bmatrix} \mathbf{f}_i + \mathbf{B}_i^T \boldsymbol{\lambda} \\ \mathbf{f}_i^C \end{bmatrix}, \tag{5.44}$$

gdzie ograniczenia są następujące:

$$\sum_{i=1}^{n} \mathbf{B}_i \mathbf{u}_i = \mathbf{0}, \tag{5.45}$$

a $\mathbf{B}_i$ jest zdefiniowana jak w rozdz. 5.4.1, tj.:

$$|\mathbf{B}_i| \mathbf{u}_i = \mathbf{u}^I, \tag{5.46}$$

ale $\mathbf{u}^I$ są zmiennymi interfejsowymi bez zmiennych dla naroży. Teraz można z równ. (5.44) wyciągnąć zmienne $\mathbf{u}_i$, które należy wstawić do równ. (5.45):

$$\sum_{i=1}^{n} \mathbf{B}_i \mathbf{K}_i^{-1} (\mathbf{f}_i + \mathbf{B}_i^T \boldsymbol{\lambda} - \mathbf{L}_i^T \mathbf{u}^C) = \mathbf{0}, \tag{5.47}$$

$$\sum_{i=1}^{n} \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{f}_i + \sum_{i=1}^{n} \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{B}_i^T \boldsymbol{\lambda} - \sum_{i=1}^{n} \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{L}_i^T \mathbf{u}^C = \mathbf{0}, \tag{5.48}$$

$$\underbrace{\sum_{i=1}^{n} -\mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{B}_i^T}_{\mathbf{F}} \boldsymbol{\lambda} + \underbrace{\sum_{i=1}^{n} \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{L}_i^T}_{\mathbf{G}_c} \mathbf{u}^C = \underbrace{\sum_{i=1}^{n} \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{f}_i}_{\mathbf{g}}. \tag{5.49}$$

Jeśli zmienne wewnętrzne zostaną wyrugowane dla subdomeny z równ. (5.44), otrzyma się:

$$\mathbf{L}_i \mathbf{K}_i^{-1} \mathbf{B}_i^T \boldsymbol{\lambda} + (\mathbf{C}_i - \mathbf{L}_i \mathbf{K}_i^{-1} \mathbf{L}_i^T) \mathbf{u}^C = \mathbf{f}_i^C - \mathbf{L}_i \mathbf{K}_i^{-1} \mathbf{f}_i. \tag{5.50}$$

Sumując dla wszystkich subdomen:

$$\underbrace{\sum_{i=1}^{n} \mathbf{L}_i \mathbf{K}_i^{-1} \mathbf{B}_i^T}_{\mathbf{G}_c^T} \lambda + \underbrace{\sum_{i=1}^{n} (\mathbf{C}_i - \mathbf{L}_i \mathbf{K}_i^{-1} \mathbf{L}_i^T)}_{-\mathbf{C}} \mathbf{u}^C = \underbrace{\sum_{i=1}^{n} \mathbf{f}_i^C - \mathbf{L}_i \mathbf{K}_i^{-1} \mathbf{f}_i}_{\mathbf{f}}. \tag{5.51}$$

Po połączeniu równ. (5.49) z równ. (5.51) jest:

$$\begin{bmatrix} \mathbf{F} & \mathbf{G}_c \\ \mathbf{G}_c^T & -\mathbf{C} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{u}^C \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ \mathbf{f} \end{bmatrix}. \tag{5.52}$$

Równanie (5.52) opisuje problem dualno-podstawowy, ponieważ łączy zmienne dualne mnożników Lagrange'a $\boldsymbol{\lambda}$ ze zmiennym podstawowymi dla przemieszczeń $\mathbf{u}^C$. Po wyeliminowaniu zmiennych $\mathbf{u}^C$ można przekształcić równ. (5.52) do nastepującego symetrycznego i dodatniego problemu dualnego dla interfejsów:

$$(\mathbf{F} + \mathbf{G}_c \mathbf{C}^{-1} \mathbf{G}_c^T) \boldsymbol{\lambda} = \mathbf{g} - \mathbf{G}_c \mathbf{C}^{-1} \mathbf{f}. \tag{5.53}$$

Równanie (5.53) jest bardzo podobne do równania dla interfejsów z oryginalnej metody FETI (patrz 5.4.1). W pracy [49] równanie (5.53) jest rozwiązywane za pomocą metody PCG (ang. *Preconditioned Conjugate Gradient*). W każdej iteracji k metody PCG , trzeba obliczyć reziduum, poprzez mnożenia macierzy przez wektor tzn. $(\mathbf{F} + \mathbf{G}_c \mathbf{C}^{-1} \mathbf{G}_c^T) \boldsymbol{\lambda}^k$, co można wykonać w dwóch krokach:

$$\text{S1} \;:\; \boldsymbol{\delta}^k = \mathbf{F} \boldsymbol{\lambda}^k = -\sum_{i=1}^{n} \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{B}_i^T \boldsymbol{\lambda}^k,$$

$$\text{S2} \;:\; \boldsymbol{\delta}^k = \boldsymbol{\delta}^k + \mathbf{G}_c \mathbf{C}^{-1} \mathbf{G}_c^T \boldsymbol{\lambda}^k.$$

Krok S1 jest podobny do głównego kroku w metodzie FETI (patrz rozdz. 5.4.1). Krok ten jest również łatwy do zrównoleglenia, ponieważ zawiera tylko obliczenia lokalne dla subdomeny - tzn. lokalne rozwiązania i wymaga komunikacji tylko między sąsiednimi subdomenami.

Z kolei krok S2 można podzielić na następujące trzy podkroki:

$$\text{S2-1} \;:\; \mathbf{y}^k = \mathbf{G}_c^T \boldsymbol{\lambda}^k = \sum_{i=1}^{n} \mathbf{L}_i \mathbf{K}_i^{-1} \mathbf{B}_i^T \boldsymbol{\lambda}^k,$$

$$\text{S2-2} \;:\; \text{Rozwiąż } \mathbf{C} \mathbf{x}^k = \mathbf{y}^k,$$

$$\text{S2-3} \;:\; \mathbf{z}^k = \mathbf{G}_c \mathbf{x}^k = \sum_{i=1}^{n} \mathbf{B}_i \mathbf{K}_i^{-1} \mathbf{L}_i^T \mathbf{x}^k.$$

Kroki S2-1 oraz S2-3 zawierają tylko obliczenia lokalne, które można łatwo sparalelizować. Krok S2-2 może być rozumiany jako rozwiązanie problemu pomocniczego, którego rozmiar jest równy liczbie naroży pomnożonej przez liczbę stopni swobody na naroże. Ten pomocniczy problem jest nazywany problemem zgrubnym (ang. *coarse problem*) FETI-DP.

Z równania (5.51), widać również, że macierz problemu zgrubnego $\mathbf{C}$ jest macierzą rzadką i może być stworzona w sposób równoległy. Ważną obserwacją jest także fakt, że w przeciwieństwie do metody FETI lub FETI 2, każdy problem subdomenowy w metodzie FETI-DP jest zawsze nieosobliwy, tzn. że nie występują „pływające" subdomeny, więc nie trzeba ich traktować w specjalny sposób. Dlatego ten sam solwer FETI-DP może być użyty zarówno do rozwiązania problemów statycznych jak i dynamicznych.

W metodzie FETI-DP stosuje się identyczne preconditionery jak w metodzie FETI (patrz rozdz. 5.4.1).

5.4.3 Implementacja FETI-DP

Metoda bezpośrednia dla problemu interfejsowego. W tym podejściu wszystkie macierze potrzebne w metodzie FETI-DP zostaną wyznaczone jawnie tzn. macierze po prawej stronie równ. (5.53) są zsumowane, a później równ. (5.53) jest rozwiązane metodą bezpośrednią. Zostało to wykonane, aby porównać FETI-DP z metodą bezpośrednią opracowaną w rozdz. 5.3 oznaczoną jako Solwer3 oraz z metodą z mieszaną precyzją opracowaną w rozdz. 5.3.5 oznaczoną jako SolwerMIX.

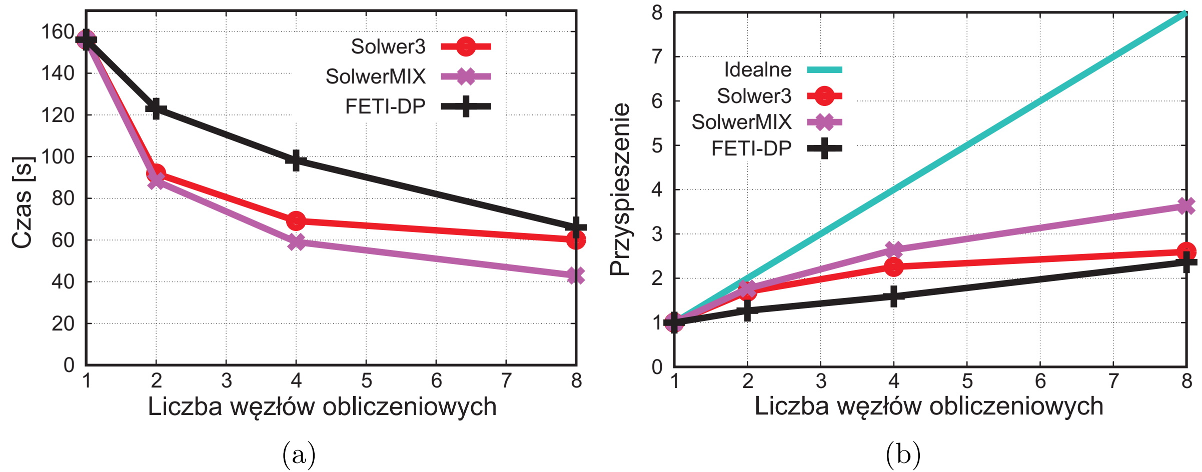
W takim podejściu trzeba wykonać następujące kroki:

1. Utworzenie macierzy:
   (a) $\mathbf{B}_i\mathbf{K}_i^{-1}\mathbf{B}_i^T$ z równ. (5.49),
   (b) $\mathbf{L}_i\mathbf{K}_i^{-1}\mathbf{L}_i^T$ z równ. (5.49),
   (c) $\mathbf{B}_i\mathbf{K}_i^{-1}\mathbf{L}_i^T$ z równ. (5.50).
2. Utworzenie macierzy dla problemu z równ. (5.53):
   (a) $\mathbf{G}_c\mathbf{C}^{-1}\mathbf{G}_c^T$,
   (b) $\mathbf{F}$.
3. Faktoryzacja macierzy $\mathbf{F} + \mathbf{G}_c\mathbf{C}^{-1}\mathbf{G}_c^T$ (do rozwiązania równ. (5.53)).

Po wykonaniu powyższych kroków, można rozwiązać cały problem wykonując tylko podstawienia wstecz i w przód (ang. *back-/forward substitutions*).

Krok 1a jest obliczony za pomocą częściowej faktoryzacji opisanej w rozdz. 5.2.3. W trakcie częściowej faktoryzacji, zostanie również wykonana faktoryzacja macierzy $\mathbf{K}_i$, dzięki czemu można obliczyć części $\mathbf{K}_i^{-1}\mathbf{L}_i^T$ dla kroków 1b oraz 1c. Ponieważ liczba naroży jest bardzo mała w stosunku do rozmiaru subdomen, koszt tych obliczeń jest również bardzo mały. Następnie krok 1b jest obliczony za pomocą procedury mnożenia macierzy rzadkiej z biblioteki MKL z macierzą $\mathbf{L}_i$. Dzięki temu, że macierz $\mathbf{B}_i$ jest macierzą boolowską, krok 1c sprowadza się do wyciągnięcia odpowiednich wierszy z macierzy $\mathbf{K}_i^{-1}\mathbf{L}_i^T$. Krok 2a jest wykonany z wykorzystaniem procedury do mnożenia macierzy z biblioteki MKL. Krok 2b to suma z poprzednich kroków. Faktoryzacja w kroku 3 jest wykonana tak, jak faktoryzacja macierzy interfejsowej w rozdz. 5.3.

Na rysunku 5.19 przedstawiono porównanie Solwera3 oraz SolweraMIX (patrz rozdz. 5.3.4 i rozdz. 5.3.5) z metodą bezpośrednią FETI-DP. Widać, że oba solwery



Rysunek 5.19: Porównanie czasu wykonania (a) i przyspieszenia (b) Solwera3 i metody bezpośredniej FETI-DP. Test kostki $N = 64$.

własne Solwer3, SolwerMIX są szybsze od metody FETI-DP dla każdej liczby węzłów obliczeniowych. Jest to zrozumiałe, ponieważ w metodzie FETI-DP, trzeba wykonać podobne operacje, co w metodzie wykorzystywanej przez własne solwery, a problem interfejsowy w kroku 3 ma podobny rozmiar jak problem interfejsowy w Solwerze3 i SolwerzeMIX, ale częściowa faktoryzacja wykonana w kroku 1a jest wykonywana na większej macierzy, niż częściowa faktoryzacja w solwerach własnych. Dlatego metoda FETI-DP nie powinna być implementowana jako metoda bezpośrednia. Jej koszt obliczeniowy jest większy niż metod bezpośrednich bazujących na uzupełnieniu Schura.

Metoda iteracyjna dla problemu interfejsowego. Główną zaletą rozwiązywania problemu na interfejsach w metodzie FETI-DP za pomocą metody iteracyjnej, jest fakt, że nie trzeba tworzyć całej macierzy interfejsowej, ponieważ wystarczy samo mnożenie macierzy przez wektor, które można wykonać równolegle, gdyż macierz interfejsowa to suma macierzy z każdej subdomeny.

W jednej iteracji algorytmu PCG w metodzie FETI-DP trzeba wykonać następujące mnożenia:

1. Mnożenie $\mathbf{F}\boldsymbol{\lambda}^k$:
   (a) $\mathbf{B}_i\mathbf{K}_i^{-1}\mathbf{B}_i^T\boldsymbol{\lambda}^k$ dla kroku S1.
2. Mnożenie $\mathbf{z}^k = \mathbf{G}_c\mathbf{C}^{-1}\mathbf{G}_c^T\boldsymbol{\lambda}^k$ dla kroku S2:
   (a) $\mathbf{y}^k = \mathbf{L}_i\mathbf{K}_i^{-1}\mathbf{B}_i^T\mathbf{v}$ dla kroku S2-1,
   (b) $\mathbf{x}^k = \mathbf{C}^{-1}\mathbf{y}^k$ dla kroku S2-2,
   (c) $\mathbf{z}^k = \mathbf{B}_i\mathbf{K}_i^{-1}\mathbf{L}_i^T\mathbf{x}^k$ dla kroku S2-3.
3. Mnożenie przez preconditioner.
   (a) Obcięty preconditioner:

i. $\mathbf{C}_i\boldsymbol{\lambda}^k$, z równ. (5.6).

(b) Preconditioner Dirichlet'a:

i. $\mathbf{C}_i\boldsymbol{\lambda}^k$ z równ. (5.6),

ii. $\mathbf{S}_i^{-1}\boldsymbol{\lambda}^k$ z równ. (5.6) .

Mnożenie przez macierz boolean'owską $\mathbf{B}_i$ jest traktowane jako proces wyboru odpowiednich kolumn/wierszy, a więc praktycznie nic nie kosztuje. Mnożenie przez macierz $\mathbf{K}_i^{-1}$ jest wykonane jako rozwiązanie układu równań $\mathbf{K}_i\mathbf{v}' = \mathbf{v}$. Faktoryzacja użyta do rozwiązania takiego układu jest wykonana w następujący sposób. Najpierw zostaje wykonana blokowa częściowa faktoryzacja macierzy (patrz dod. I):

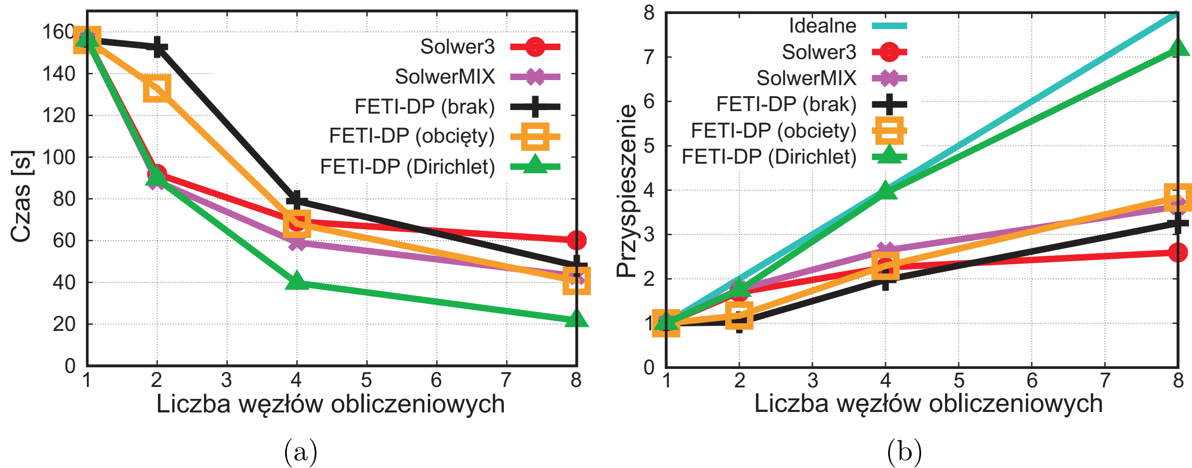
$$\begin{bmatrix} \mathbf{K}_i & \mathbf{L}_i^T \\ \mathbf{L}_i & \mathbf{C}_i \end{bmatrix} = \begin{bmatrix} \mathbf{K}_{ii} & \mathbf{L}_{ib}^T & \mathbf{L}_{ii}^T \\ \mathbf{L}_{ib} & \mathbf{C}_{ibb} & \mathbf{L}_{ibb}^T \\ \mathbf{L}_{ii} & \mathbf{L}_{ibb} & \mathbf{C}_i \end{bmatrix}. \tag{5.54}$$

W ten sposób można wykonać podstawienie wprzód i wstecz dla macierzy $\mathbf{K}$, jak i dla lokalnych uzupełnień Schura $\mathbf{S}_i$, które są potrzebne do zastosowania Preconditionera Dirichlet'a wymaganego w kroku 3(b)ii. Dzięki temu otrzymany zostanie również udział subdomeny w problemie zgrubnym $\mathbf{C}_i - \mathbf{L}_i\mathbf{K}_i\mathbf{L}_i^T$. Podsumowując należy wykonać następujące obliczeniowo kosztowne operacje:

1. Blokowa częściowa faktoryzacja z równ. (5.54).

2. Trzy podstawienia wstecz/wprzód - pierwsze dla kroku 1 i kroku 2a, drugie dla kroku 2c i trzecie tylko wtedy gdy używany jest preconditioner Dirichlet'a dla kroku 3(b)ii.

3. Faktoryzacja problemu głównego dla kroku 2b.



Rysunek 5.20: Porównanie czasu wykonania (a) i przyspieszenia (b) Solwera3 i metody FETI-DP. W nawiasach podano nazwy użytych preconditionerów. Test kostki $N = 64$.

Na rysunku 5.20a pokazano czas poszczególnych solwerów dla testu kostki $N = 64$. Liczba węzłów obliczeniowych jest równa liczbie subdomen. Widać, że FETI-DP z preconditionerem Dirichleta wypada w tym zestawieniu zdecydowanie najlepiej. Metoda FETI-DP bez preconditionera wykonuje się w zbliżonym czasie do Solwera3. Metoda FETI-DP z obciętym preconditionerem jest szybsza od metody bezpośredniej tylko dla przypadku 8 subdomen, ale równocześnie jest wolniejsza od SolweraMIX. Porównanie liczby potrzebnych iteracji i czasu wykonania pojedynczej iteracji różnych preconditionerów zostało przedstawione w tab. 5.12. Na rysunku 5.20b pokazano skalowalność zaimplementowanych metod. Solwer FETI-DP z preconditionerem Dirichleta wykazuje skalowalność ok. 7 dla 8 węzłów obliczeniowych.

Tabela 5.12: Porównanie preconditionerów dla metody FETI-DP.

Preconditioner	Liczba iteracji	Czas względny (1.00=brak preconditionera)
Brak	93	1.00
Obcięty	80	1.01
Dirichlet'a	30	1.50

5.4.4 Wnioski

W tym podrozdziale zaprezentowano metodę FETI-DP zarówno jako metodę bezpośrednią, jak i metodę iteracyjną. Pokazano, że implementacja metody FETI-DP jako metody bezpośredniej daje gorsze rezultaty niż solwery własne Solwer3 i SolwerMIX bazujące na uzupełnieniu Schura. Metoda FETI-DP z iteracyjnym rozwiązywaniem problemu interfejsowego działa szybciej od własnego bezpośredniego Solwera3 zaprezentowanego w rozdz. 5.3.4 oraz od SolweraMIX z mieszaną precyzją z rozdz. 5.3.5, ale tylko wtedy gdy zostanie użyty preconditioner Dirichlet'a. Zaprezentowano również wiele szczegółów implementacyjnych, które rzadko są opisywane w literaturze.

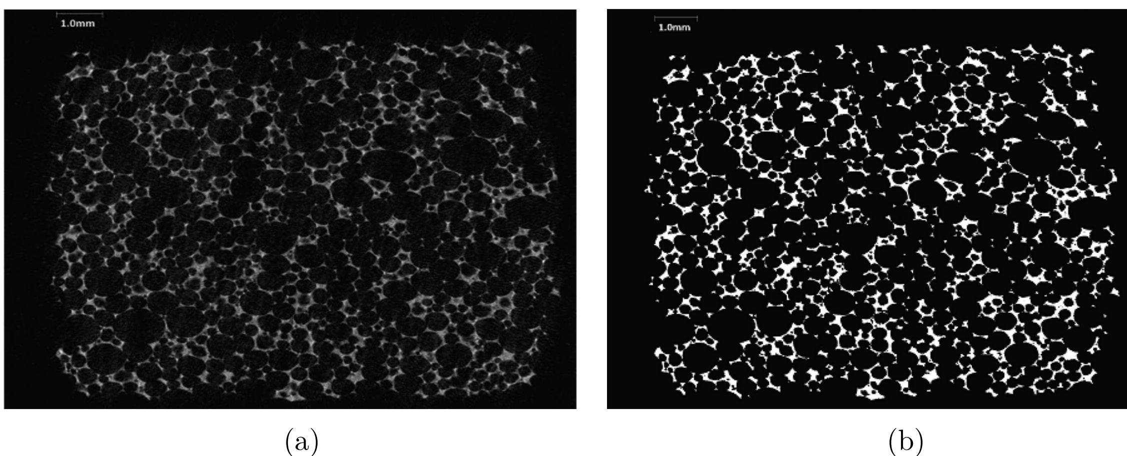
6 Przykłady rozwiązywania układów równań liniowych dla wybranych materiałów

W tym rozdziale opisano zastosowanie opracowanych we wcześniejszych rozdziałach algorytmów do obliczeń efektywnych sztywności zastępczych wykorzystując metodę Reprezentatywnego Elementu Objętościowego (ang. *Representative Volume Element* (RVE)).

6.1 Pianka korundowa

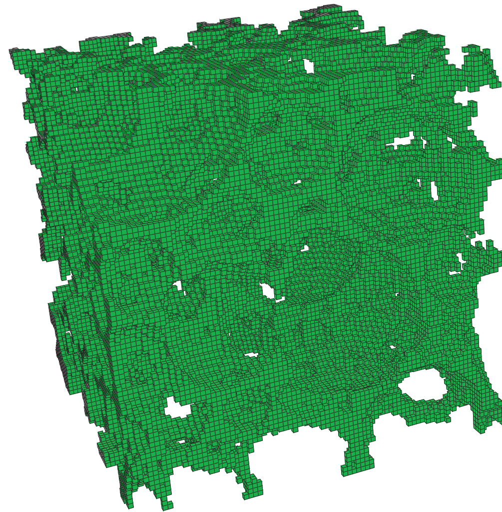
Komórkowa ceramika przebadana w [82] to korundowa pianka z otwartymi porami stworzona w procesie odlewu żelowego (ang. *gelcasting*). Taka pianka charakteryzuje się względnymi gęstościami od 0.5 do 0.9. Korund to minerał będący tlenkiem glinu Al_2O_3.

Aby stworzyć model numeryczny, została przebadana próbka pianki o 86% porowatości. Dla pianek, do rekonstrukcji geometrii, najczęściej stosuje się obrazy z tomografii komputerowej. Przykładowy przekrojowy obraz dla wybranego materiału przedstawiono na rys. 6.1a. Takie obrazy są później konwertowane do obrazów binarnych. Rysunek 6.1b przedstawia te same przekrojowe obrazy jak na rys. 6.1a, ale po procesie binaryzacji (separacji fazy matrycy i pustek). Rozkład materiału Al_2O_3 jest zaznaczony białym kolorem. Skany z tomografii komputerowej za pomocą oprogramowania komercyjnego ScanIP oraz ScanFE (www.simpleware.com) zostały przetworzone na model złożony z elementów skończonych. Oprogramowanie to w pełni automatycznie wykonuje dysktretyzację objętościową na elementy skończone. Może to zrobić za pomocą dwóch algorytmów: (i) tradycyjny oparty na obrazach algorytm tworzący elementy heksagonalne i tetragonalne (FE-Grid), (ii) nowy algorytm, który pozwala na adaptację siatki i znacznie redukuje rozmiar siatki (FE-Free). W tej pracy wykorzystano algorytm FE-Grid, który wygenerował bardzo wysokiej jakości siatkę dla tak skomplikowanych segmentacji, patrz rys. 6.2. Siatka została sprawdzona przez autora niniejszej rozprawy w programie VERDE [108] (opis w dodatku C).



(a) (b)

Rysunek 6.1: Przekrojowe obrazy: (a) prawdziwej ceramicznej pianki Al_2O_3 oraz (b) pianki po procesie binaryzacji. Próbka o porowatości 86%.

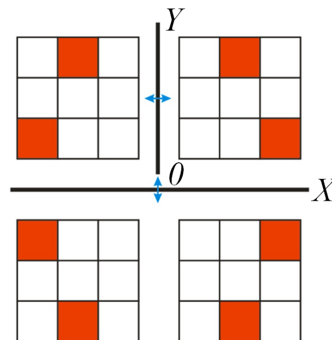


Rysunek 6.2: Siatka MES dla pianki ceramicznej (3.8 mln niewiadomych).

Dla celów testowych obliczono macierz efektywnych sztywności zastępczych dla pianki korundowej, tak by później można ją było wykorzystać w bardziej skomplikowanej strukturze. Macierz taka powinna być symetryczna, jednak próbka która została autorowi udostępniona była za mała i dlatego otrzymano macierz, która nie była symetryczna. Dyskusja symetryczności macierzy zostanie przeprowadzona w następnym podrozdziale.

6.1.1 Niesymetryczna macierz konstytutywna

Pierwszy sposób na uzyskanie symetrycznej macierzy konstytutywnej z macierzy niesymetrycznej to wyznaczenie symetrycznej części tej macierzy (M1). Drugim sposobem jest wygenerowanie lustrzanych odbić modelu wykorzystując 3 ortogonalne płaszczyzny (M2, patrz rys. 6.3). Opracowano program MeshSymmetry, który realizuje po-



Rysunek 6.3: Metoda symetryzacji macierzy konstytutywnej. Odbicia lustrzane przykład dwuwymiarowy. W pracy zastosowano dodatkowo odbicie względem płaszczyzny rysunku dla elementów trójwymiarowych.

wyższy sposób, a jego najważniejszy fragment przedstawiono w Kodzie 6.1. Warto zauważyć, że przy tworzeniu lustrzanych odbić, nie tylko odbijane są współrzęd-

ne punktów, ale również zmienia się kolejność numeracji wierzchołków w elemencie.

Kod 6.1: Fragment procedury z programu MeshSymmetry

```
1  coor = 3       ! ktora plaszczyzna
2  s = 1.0d0      ! wspolrzedne plaszczyzny
3  do i = 1, nnod
4   x(coor,i) = 2 * s - x(coor,i)
5  enddo
6  do i = 1, nelm
7   do j = 1, 8
8    el(i,j) = 2 * el(i,j) - 1
9   enddo
10  if(coor.eq.1) then
11   eln(1)=el(2); eln(2)=el(1); eln(3)=el(4); eln(4)=el(3)
12   eln(5)=el(6); eln(6)=el(5); eln(7)=el(8); eln(8)=el(7)
13  endif
14  if(coor.eq.2) then
15   eln(1)=el(5); eln(2)=el(6); eln(3)=el(7); eln(4)=el(8)
16   eln(5)=el(1); eln(6)=el(2); eln(7)=el(3); eln(8)=el(4)
17  endif
18  if(coor.eq.3) then
19   eln(1)=el(4); eln(2)=el(3); eln(3)=el(2); eln(4)=el(1)
20   eln(5)=el(8); eln(6)=el(7); eln(7)=el(6); eln(8)=el(5)
21  endif
22 enddo
```

Wykorzystując lustrzane odbicia w programie MeshSymmetry uzyskano 8-krotnie większy model i zmniejszono błąd symetryczności macierzy, patrz tab. 6.1.

Na podstawie obrazu z tomografii komputerowej z rozdz. 6.1 wygenerowano 4 modele MES. Największy model został stworzony na podstawie 400×400 pikseli wyciętych ze zdjęcia z rys. 6.1a. Każdy kolejny model otrzymano wykorzystując odpowiednio 200×200, 100×100 i 50×50 pikseli ze środka obszaru 400×400 pikseli. W tab. 6.2 podano błędy względne dla elementów macierzy konstytutywnej dla obu metod generowane przez zmniejszenie modelu, przy czym za wartości referencyjne przyjęto te dla największego modelu (400×400 pikseli). Zauważmy, że różnica między błędami względnymi otrzymanymi metodami M1 i M2 nie jest duża.

Warto zauważyć, że metodę odbić lustrzanych można także wykorzystać do tworzenia większych modeli obliczeniowych (np. rzędu 30 mln niewiadomych), patrz rozdz. 6.1.3 i 6.2.2.

Tabela 6.1: Błąd niesymetryczności dla metody odbić lustrzanych M2.

Model	Błąd niesymetryczności [%]	
	Przed	Po
Kompozyt	3.4×10^{-2}	3.6×10^{-14}
Pianka	4.4×10^{-1}	1.6×10^{-5}

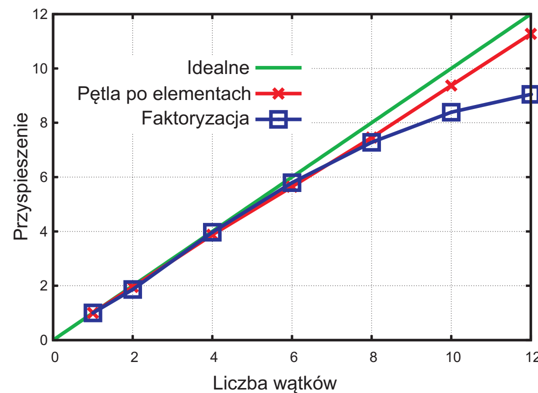
Tabela 6.2: Błąd względny dla naiwnej symetryzacji (M1) i metody odbić lustrzanych (M2) generowany przez zmniejszenie modelu.

	Błąd względny [%]					
Element	50 × 50		100 × 100		200 × 200	
macierzy	M1	M2	M1	M2	M1	M2
1 1	18.03	18.03	12.04	12.04	11.48	11.48
2 2	19.25	19.25	13.27	13.27	12.81	12.81
3 3	38.68	38.68	17.57	17.57	11.20	11.20
4 4	58.24	57.16	19.42	17.03	14.77	13.36
5 5	42.95	40.19	20.97	19.28	13.07	11.56
6 6	3.01	0.51	20.88	17.58	16.26	15.16
1 2	5.09	5.09	12.62	12.62	15.07	15.07
1 3	38.43	38.43	16.22	16.22	14.13	14.13
2 3	36.34	36.34	18.42	18.42	13.88	13.88

6.1.2 Wyniki testów solwera na jednym węźle

Na rys. 6.4 rozróżniono dwie fazy rozwiązania: (1) pętla po elementach, w tym tworzenie macierzy sztywności (2) faktoryzacja za pomocą zmodyfikowanego solwera modMA86.

Można zauważyć, że pętla po elementach skaluje się w ok. 11.27 razy dla 12 wątków, a faktoryzacja skaluje się ok. 9.05 razy dla 12 wątków, a oba procesy 9.18 razy. Zaobserwowano również wzrost zapotrzebowania na pamięć o 12%, z 15.45 do 17.68 GB. W tab. 6.3 przedstawiono czasy wykonań za pomocą różnych metod przenumerowania. Widać, że algorytmy zagnieżdżonego podziału (jedno- i wielowątkowy) przeważają nad metodą AMD.



Rysunek 6.4: Skalowalność równoległych części obliczeń. 3.8 miliona niewiadomych. Przenumerowanie za pomocą METIS.

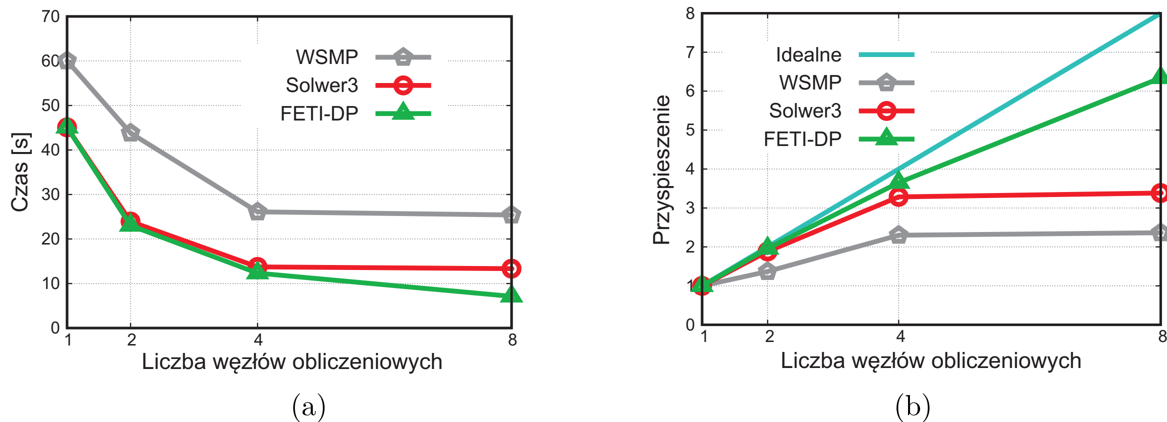
Tabela 6.3: Czas i przyspieszenie dla różnych metod przenumerowania macierzy. 3.8 miliona niewiadomych.

Wątki	Faza	AMD	METIS	mtMETIS
			Time [secs]	
1	Przenumerowanie	7.74	73.98	73.98
	Faktoryzacja	1813.78	421.08	421.08
	Razem	1821.52	495.06	495.06
12	Przenumerowanie	7.74	73.98	9.00
	Faktoryzacja	279.18	46.55	45.09
	Razem	286.92	120.53	**54.09**
			Speedup ratio	
	Faktoryzacja	6.49	9.05	9.34
	Razem	6.35	4.11	9.15

6.1.3 Wyniki testów solwera na wielu węzłach

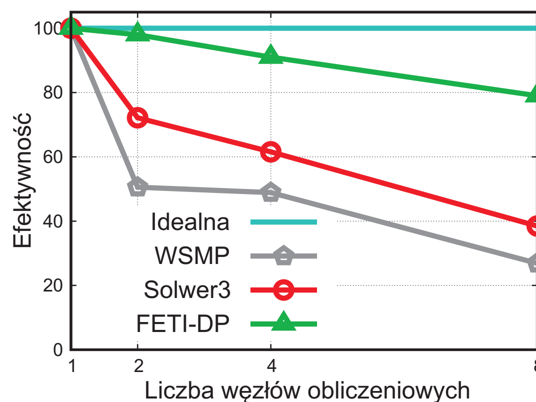
Na rys. 6.5a pokazano czas wykonania w zależności od liczby węzłów obliczeniowych. Czas faktoryzacji dla Solwera3 jest krótszy ok. 2x niż czas dla WSMP dla 8 węzłów. Rysunek 6.5b przedstawia skalowalność i można zauważyć, że Solwer3 wykazuje lepszą skalowalność niż WSMP; wynosi ona ok. 3.38 razy dla 8 węzłów. Dla porównania na rys. 6.5a i 6.5b przedstawiono również wydajność metody FETI-DP opisanej w rozdz. 5.4.3, można zauważyć, że metoda FETI-DP wykazuje najkrótszy czas wykonania oraz przyspieszenie ok. 6.5 razy dla 8 węzłów.

Solwery na wielu węzłach wykorzystuje się, aby móc obliczać zadania o większych rozmiarach niż te możliwe do rozwiązania na jednym węźle. Dlatego obliczono również efektywność słabą solwera (ang. *weak efficiency*, patrz 2.2.1 oraz [89]) używając zadania o zmiennym rozmiarze. W celu uzyskania większych zadań zastosowano technikę odbić lustrzanych (patrz rozdz. 6.1.1). W ten sposób otrzymano zadania o rozmiarach 3.8, 7.6,



Rysunek 6.5: Czas wykonania (a) i przyspieszenie (b) etapu faktoryzacji, porównanie własnej implementacji z WSMP. 3.8 miliona niewiadomych.

15.2 i 30.4 milionów niewiadomych. Efektywność solwerów została zaprezentowana na rys. 6.6. Widać, że efektywność Solwera3 jest lepsza o ok. 11% od solwera WSMP. Dla porównania pokazano również efektywność metody FETI-DP, jest ona najwyższa (ok. 80%).



Rysunek 6.6: Efektywność solwerów na wielu węzłach obliczeniowych. Zadania o zmiennym rozmiarze: 3.8 mln, 7.6 mln, 15.2 mln i 30.4 mln niewiadomych.

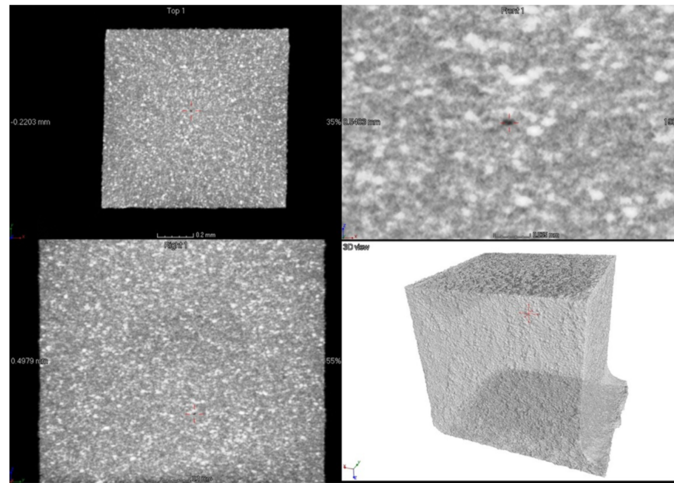
6.2 Ceramiczny kompozyt

Przykład ten wykorzystuje kompozyt złożony z chromu, aluminium i renu, można go otrzymać za pomocą różnych metod, najczęściej pod obciążeniem jednoosiowym spiekania pod ciśnieniem (ang. *hot pressing*) lub bezciśnieniowego (swobodnego) spiekania (ang. *pressureless sintering*). W pracy [111] stworzono próbki takiego kompozytu za pomocą metody iskrowego spiekania plazmowego (ang. *spark plasma sintering*). Do przygotowania kompozytu użyto następujących proszków:

- Cr (F.W. Winter) ze średnim rozmiarem ziarna 2-5 μm,

- Al_2O_3 (New Met Koch, 99.99% czystości) ze średnim rozmiarem ziarna 5 μm,
- Re (KGHM Ecoren S.A., min. 99.9% czystości) ze średnim rozmiarem ziarna 80 μm.

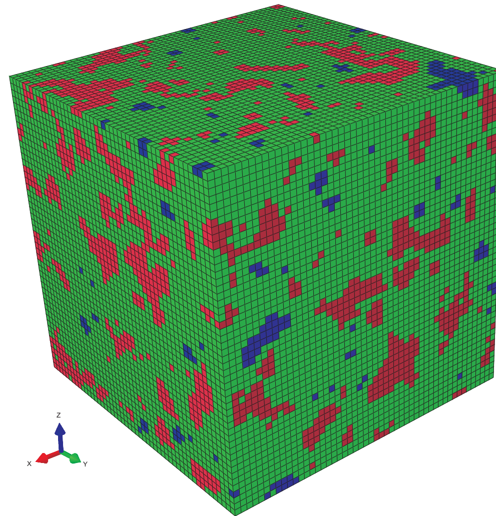
Próbki były poddane spiekaniu w temperaturze 1300 C i pod ciśnieniem 30 MPa. Szczegóły na temat samego procesu tworzenia kompozytu można znaleźć w [111]. We wspomnianej pracy podjęto również próbę wyliczenia efektywnego modułu Young'a dla takiego kompozytu, a w niniejszej pracy zostały te obliczenia powtórzone i przedstawiono wydajność dla algorytmów równoległych.



Rysunek 6.7: Skan micro-CT dla kompozytu $Cr/25Al_2O_3$ z 2% domieszką objętościową renu.

Siatka MES tego kompozytu została przedstawiona na rys. 6.8. Została ona stworzony na podstawie zdjęć mikrostruktury materiału. Aby ją otrzymać, prawdziwe próbki kompozytu były przeskanowane promieniami X tomografii mikrokomputerowej używając Nanotom M (Phoenix - Xray) z rozdzielczością voxela 0.9 μm. Próbki były kostkami sześciennymi 1 x 1 x 1 mm, tak aby zapewnić dobrą rozdzielczość skanów micro-CT, oraz aby zapewnić reprezentatywność mikrostruktury. Podobnie jak w rozdz. 6.1 wykorzystano oprogramowania komercyjne ScanIP oraz ScanFE do przetworzenia skanów micro-CT na model złożony z elementów skończonych. W pracy [111] wykorzystano algorytm FE-Grid, który wygenerował bardzo wysokiej jakości siatki dla nawet bardzo skomplikowanych segmentacji. Siatka została dodatkowo sprawdzona przez autora niniejszej rozprawy w programie VERDE [108] (opis w dodatku C).

W tej pracy wykorzystano model MES przekazany przez autorów pracy [111]. Użyty został przedstawiony model kompozytu z 5% domieszką renu oraz z 25% wkładem aluminium.



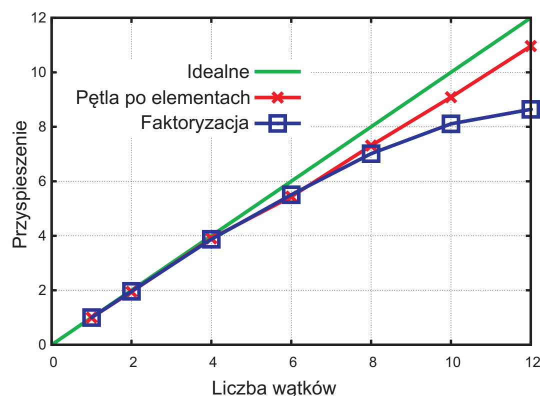
Rysunek 6.8: Siatka MES dla kompozytu $Cr/25Al_2O_3+5Re$ wielkość ziarna 5 mikronów (czerwony - chrom, zielony - aluminium, niebieski - ren), 0.4 mln niewiadomych.

6.2.1 Wyniki testów solwera na jednym węźle

Na rys. 6.9 rozróżniono dwie fazy rozwiązania: (1) pętla po elementach, w tym tworzenie macierzy sztywności (2) faktoryzacja za pomocą zmodyfikowanego solwera HSL MA86 - modMA86.

Można zauważyć, że pętla po elementach skaluje się ok. 10.97 razy dla 12 wątków, a faktoryzacja skaluje się ok. 8.08 razy dla 12 wątków, a dla obu faz 9.13. Zaobserwowano również wzrost zapotrzebowania na pamięć o 12% z 5.47 do 6.15 GB.

W tab. 6.4 przedstawiono czasy wykonań za pomocą różnych metod przenumerowania. Widać, że algorytmy zagnieżdżonego podziału (jedno- i wielowątkowy) przeważają nad metodą AMD.



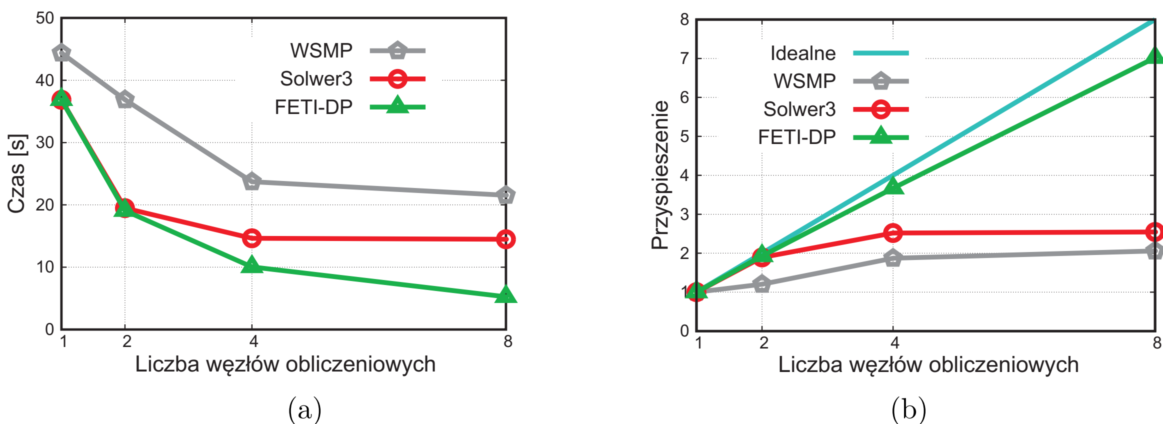
Rysunek 6.9: Skalowalność równoległych części obliczeń. 0.4 miliona niewiadomych. Przenumerowanie za pomocą METIS.

Tabela 6.4: Czas i przyspieszenie dla różnych metod przenumerowania macierzy. 0.4 miliona niewiadomych.

Wątki	Faza	AMD	METIS	mtMETIS
			Czas [sek.]	
1	Przenumerowanie	0.53	8.33	8.33
	Faktoryzacja	1316.06	318.67	318.67
	Razem	1316.59	327.00	327.00
12	Przenumerowanie	0.53	8.33	1.70
	Faktoryzacja	147.85	36.88	50.20
	Razem	148.38	45.21	51.90
			Przyspieszenie	
	Faktoryzacja	8.90	8.64	6.51
	Razem	8.87	7.23	6.30

6.2.2 Wyniki testów solwera na wielu węzłach

Na rys. 6.10a pokazano czas wykonania faktoryzacji w zależności od liczby węzłów dla zadania z ok. 0.4 miliona niewiadomymi. Czas dla własnego Solwera3 jest krótszy niż dla WSMP. Rysunek 6.10b przedstawia skalowalność dla tego samego zadania i widać, że Solwer3 ma lepszą skalowalność niż WSMP; wynosi ona ok. 2.54 razy dla 8 węzłów. Dla porównania na rys. 6.10a i 6.10b pokazano również wydajność metody FETI-DP opisanej w rozdz. 5.4.3, można zauważyć, że metoda FETI-DP wykazuje najkrótszy czas wykonania oraz przyspieszenie ok. 7 razy dla 8 węzłów.

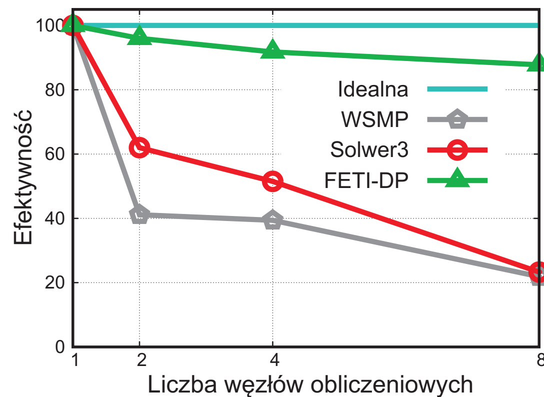


(a) (b)

Rysunek 6.10: Czas wykonania (a) i przyspieszenie (b) etapu faktoryzacji. 0.4 miliona niewiadomych.

Podobnie jak w rozdz. 6.1 obliczono również efektywność słabą solwerów (ang. *weak efficiency*, patrz 2.2.1 oraz [89]) używając zadania o zmiennym rozmiarze. Do tego celu wykorzystano zadanie z rozdz. 6.2.1, które zostało symetrycznie odbite względem jednej płaszczyzny. Postępowanie to opisano w rozdz. 6.1.1. W ten sposób uzyskano dwa razy

większe zadanie, które obliczono na dwóch węzłach. Wykorzystując kolejne lustrzane odbicia opisane w rozdz. 6.1.1 wygenerowano również siatki dla zadania dla 4 węzłów i 8 węzłów. W trakcie testów wykorzystano siatki o następujących rozmiarach: 0.4 mln, 0.8 mln, 1.6 mln i 3.2 mln niewiadomych. Efektywność Solwera3 została zaprezentowana na rys. 6.11. Widać, że jego efektywność jest lepsza o ok. 2% od solwera WSMP. Dla porównania na rys. 6.11 pokazano również efektywność metody FETI-DP, jest ona najwyższa (ok. 90%).



Rysunek 6.11

Rysunek 6.12: Efektywność Solwera3, porównanie z solwerem WSMP. Zadania o zmiennym rozmiarze 0.4 mln, 0.8 mln, 1.6 mln i 3.2 mln niewiadomych.

7 Podsumowanie

Zdaniem autora, w rozprawie otrzymano następujące oryginalne rezultaty:

1. Opracowano podstawy teoretyczne obliczeń równoległych w rozdz. 2. W sposób jednolity opisano podstawowe prawa obliczeń równoległych (Amdahla i Gustafsona) oraz pojęcie skalowalności. Dodatkowo przedstawiono model „Roofline" dla procesorów klastra GRAFEN.

2. Zrównoleglono pętlę po elementach w kodzie programu FEAP z wykorzystaniem OpenMP, patrz rozdz. 3. Zastosowane rozwiązanie charakteryzuje się: (1) redukcją macierzy elementowych do macierzy globalnej natychmiast po wygenerowaniu, bez dodatkowego magazynowania ich w pamięci, (2) użyciem standardowych dyrektyw OpenMP, (3) zastosowaniem dyrektywy *ATOMIC* do redukcji macierzy elementowych do macierzy globalnej, (4) przyspieszeniem rzędu 11 razy na 12 rdzeniach i efektywnością rzędu 95%. Przeprowadzono szereg testów dla trójwymiarowych i powłokowych elementów skończonych. Przedstawiono również analizę implementacji pętli po elementach za pomocą modelu „Roofline".

3. Podjęto szereg kwestii dotyczących równoległego rozwiązywanie układu równań liniowych na jednym węźle obliczeniowym, patrz rozdz. 4:

   (a) Opracowano zestaw wyników odniesienia dla następujących solwerów bezpośrednich dla macierzy rzadkich: (1) PARDISO, (2) Intel MKL PARDISO, (3) WSMP, (4) HSL MA86, i (5) HSL MA97. Zaimplementowano interfejs między programem FEAP a tymi solwerami, przetestowano czasy wykonań i skalowalność oraz analizę powyższych solwerów w modelu „Roofline" tworząc zasób wyników odniesienia.

   (b) Przeanalizowano strukturę zadań wykonywanych w solwerze HSL MA86 i zbadano czasy wykonań poszczególnych zadań podczas fazy faktoryzacji. Zaproponowano wzór na maksymalną liczbę zadań w puli zadań dla wątku, dzięki czemu czas wykonania fazy faktoryzacji zmniejszył się o 5%. Dodatkowo zaproponowano dwie inne zmiany w kodzie solwera, co łącznie skróciło czas fazy faktoryzacji o 11.5%. Przyśpieszenie wynosi 9.05 dla 12 rdzeni. Zmodyfikowany solwer oznaczono modMA86.

   (c) Zaimplementowano metodę iteracyjnego poprawiania rozwiązania układu równań, która umożliwiła opracowanie solwera z mieszaną precyzją oznaczonego modMA86mix. Faktoryzacja jest wykonywana w pojedynczej precyzji, co skraca jej czas ponad dwukrotnie, a iteracyjne poprawianie w podwójnej precyzji. Metoda ta skaluje się lepiej niż standardowa metoda w podwójnej precyzji - przyspieszenie wynosi 10.28 dla 12 rdzeni, a wykorzystanie pamięci jest zmniejszone o 43% w porównaniu do tej dla podwójnej precyzji, co umożliwia obliczanie większych zadań.

4. W celu zwiększenia efektywności rozwiązywania układu równań opracowano solwer na klaster, bazujący na dekompozycji obszaru i obliczaniu uzupełnienia Schura za pomocą techniki częściowej faktoryzacji, patrz rozdz. 5:

   (a) Zaimplementowano metodę częściowej faktoryzacji w solwerze HSL MA86, dzięki czemu skrócono czas obliczenia uzupełnienia Schura przeszło 12 razy porównując do standardowej metody rozwiązywania w pakietach po 12 prawych stron.

   (b) Zaprojektowano i zaimplementowano algorytm sekwencyjnej częściowej faktoryzacji na pojedynczym węźle, który zmniejsza zapotrzebowania na pamięć o ok. 32%. Zmodyfikowany solwer oznaczono modMA86mem.

   (c) Wyprowadzono wzory dla hierarchicznej faktoryzacji macierzy blokowej, które podane są tylko w postaci końcowej w pracy [34]. Zaimplementowano Solwer3 wykorzystujący tę hierarchiczną faktoryzację, którego skalowalność oraz szybkość jest porównywalna z solwerem komercyjnym WSMP. Skalowalność fazy faktoryzacji tego solwera wynosi 2.69 dla 8 węzłów obliczeniowych. Uwzględniając wielowątkowość obliczeń na pojedynczym węźle, solwer ten zapewnia przyspieszenie ponad 25-krotne.

   (d) Zaproponowano zastosowanie mieszanej precyzji do rozwiązania macierzy dla interfejsów w solwerze Solwer3, co poprawiło skalowalność fazy faktoryzacji z 2.69 do 3.63 dla 8 węzłów obliczeniowych. Zmodyfikowany solwer oznaczono SolverMIX. Uwzględniając wielowątkowość obliczeń na pojedynczym węźle, solwer ten zapewnia przyspieszenie ponad 36-krotne.

   (e) Zaprezentowano metodę FETI-DP, z solwerem iteracyjnym wykorzystanym do rozwiązania równania dla interfejsów, jako alternatywę dla obliczeń z solwerem bezpośrednim. Metoda ta wraz z preconditionerem Dirichleta wykazała skalowalność 7 dla 8 węzłów obliczeniowych. Uwzględniając wielowątkowość obliczeń na pojedynczym węźle, solwer ten zapewnia przyspieszenie ponad 70-krotne.

Podsumowując, w pracy zaprezentowano następujące własne implementacje:

1. ompFEAP - wielowątkowa wersja programu FEAP - opis w rozdz. 3.

2. modMA86 - zmodyfikowana wersja solwera HSL MA86 - opis w rozdz. 4.3.

3. modMA86mix - zmodyfikowana wersja solwera HSL MA86, która wykorzystuje mieszaną precyzję - opis w rozdz. 4.4.2.

4. modMA86mem - zmodyfikowana wersja solwera HSL MA86, która korzysta z częściowej faktoryzacji i uzupełnienia Schura do obniżenia zapotrzebowania na pamięć - opis w rozdz. 5.2.5.

5. Solwer1 - rozproszony solwer wykorzystujący częściową faktoryzację i solwer modMA86 dla macierzy interfejsowej - opis w rozdz. 5.3.2.

6. Solwer2 - rozproszony solwer wykorzystujący częściową faktoryzację i solwer MA64 dla macierzy interfejsowej - opis w rozdz. 5.3.2.
7. Solwer3 - rozproszony solwer wykorzystujący częściową faktoryzację i solwer MA64 i hierarchiczną faktoryzację dla macierzy interfejsowej - opis w rozdz. 5.3.4.
8. SolwerMIX - zmodyfikowany Solwer3 wykorzystujący mieszaną precyzję w trakcie faktoryzacji macierzy interfejsowej - opis w rozdz. 5.3.5.
9. MeshSymmetry - program służący do tworzenia lustrzanych odbić modelu MES - opis w rozdz. 6.1.1.
10. wrapPAPI - procedury opakowujące bibliotekę PAPI, które ułatwiają korzystanie z niej - opis w Dodatku H.

Literatura

[1] Amdahl G.M.: *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities.* AFIPS Conference Proceedings, Vol. 30, pp. 483–485 (1967) 18

[2] Amestoy P.R., Davis T.A., Duff I.S.: *An approximate minimum degree ordering algorithm.* SIAM J. Matrix Analysis & Applic, Vol. 17 (4), pp. 886–905 (1996) 77

[3] Arioli M., Duff I., Gratton S., Pralet S.: *A note on GMRES preconditioned by a perturbed* LDL^T *decomposition with static pivoting.* SIAM J. Scientific Computing, Vol. 29, pp. 2024–2044 (2007) 93

[4] Arioli M., Duff I.: *FGMRES to obtain backward stability in mixed precision.* Technical Report RAL-TR-2008-006, Rutherford Appleton Laboratory (2008) 93

[5] Ashcraft C.: *The fan-both family of column-based distributed cholesky factorization algorithms.* Graph Theory and Sparse Matrix Computations. Springer-Verlag, New York, pp.159–190 (1993) 95

[6] Ashcraft C., Eisenstat S.C., Liu J. W.-H., Sherman A.H.: *A comparison of three column based distributed sparse factorizationschemes.* Technical Report YALEU/DCS/RR-810,Yale University,New Haven, pp.159–190 (1991) 96

[7] Biocentrum Ochota, http://www.biocentrumochota.pan.pl/ 164

[8] Benkner S., Brandes T.: *Efficient Parallelization of Unstructured Reductions on Shared Memory Parallel Architectures.* Parallel and Distributed Processing. Lecture Notes in Computer Science, Vol. 1800, 435-442 Springer, 2000 41, 47, 57

[9] Brenner S.C.: *The condition number of the Schur complement in domain decomposition.* University of Minnesota Digital Conservancy, http://hdl.handle.net/11299/3174, (1998) 117

[10] Bunch J.R., Kaufman L.: *Some stable methods for calculating inertia and solving symmetric linear systems.* Mathematics of Computation, Vol. 31, 163–179, 1977 74

[11] Buttari A., Dongarra J., Kurzak J., Luszczek P.: *Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy.* Journal ACM Transactions on Mathematical Software, Vol. 34 (4), 1-22, 2008 92

[12] Cecka C., Lew A.J., Darve E.: *Assembly of Finite Element Methods on Graphics Processors.* Int. J. Num. Meth. Engng, Vol. 5, 640-669 (2011) 42, 47

[13] Chen G., Malkowski K., Kandemir M., Raghavan P.: *Reducing power with performance constraints for parallel sparse applications.* Parallel Computing, Vol. 5, 65-74 (1987) 72

[14] Choi J., Dongarra J.J., Walker D.W.: *The design of scalable software libraries for distributed memory concurrent computers.* Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), Workshop 11, Volume 12 (2005) 95

[15] Chu E., George A.: *Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor.* Parallel Processing Symposium, Cancun, 792–799 (1987) 95

[16] Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R.: *Parallel Programming in OpenMP.* Acadaemic Press, 2001 26, 27, 28, 29, 48

[17] Czech Z.: *Wprowadzenie do obliczeń równoległych* Wydawnictwo Naukowe PWN SA. Warszawa; 2010. 16, 33, 34

[18] Davis T.A., Rajamanickam S., Wissam M. S-L.: *A survey of direct methods for sparse linear systems* Acta Numerica, Vol. 25, pp. 383–566 (2016) 59, 72

[19] Demmel J.W., Eisenstat S.C., Gilbert J.R., Li X.S., Liu J.W.H.: *A supernodal approach to sparse partial pivoting* SIAM J. Matrix Anal. Appl., Vol. 20 (3), pp. 720–755 (2009) 66

[20] Desperez F., Dongarra J.J., Tourancheau B.: *A supernodal approach to sparse partial pivoting* Parallel Process. Lett., Vol. 5 (2), pp. 157–169 (1995) 95

[21] Dollar H.S., Scott J.A.: *A note on fast approximate minimum degree orderings for symmetric matrices with some dense rows.* Numerical Linear Algebra with Applications, Vol. 17, pp. 43–55 (2009) 77

[22] Duff I.S.: *On the number of nonzeros added when Gaussian elimination is performed on sparse random matrices.* Math. Comp, Vol. 28, pp. 219–230 (1974) 59

[23] Duffland I.S., Reid J.K.: *MA27 - A set of Fortran subroutines for solving sparse symmetric sets of linear equations.* Technical Report AERE R-10533, HMSO, London (1982) 76

[24] Duffland I.S., Reid J.K.: *MA47, a Fortran code for direct solution of indefinite sparse symmetric linear systems.* Technical Report RAL-95-001, Didcot, Oxon, UK (1995) 76

[25] Taylor R.L.: *FEAP.* Ver. 8.4. (2014) , 9, 12, 13, 40, 42, 57, 93

[26] Fialko S.: *PARFES: A method for solving finite element linear equations on multi-core computers.* Advances in Engineering Software, Vol. 41, 1256-1265 (2010) 41

[27] Fialko S.: *Modelowanie zagadnień technicznych.* Skrytp. Wydział Fizyki, Matematyki i Informatyki Politechniki Krakowskiej, Kraków (2011) 71

[28] Furuichi M., May D.A., Tackley P.J.: *Development of a Stokes flow solver robust to large viscosity jumps using a Schur complement approach with mixed precision arithmetic.* Journal of Computational Physics, Vol. 230, pp. 8835-–8851 (2011) 117

[29] GRAFEN, http://info.grafen.ippt.pan.pl/ 9, 25, 43, 74, 105, 164

[30] George A., Liu J.W.-H., Ng E.G.-Y.: *Communication reduction in parallel sparse Cholesky factorization on a hypercube.* Parallel Computing, Vol. 10(3) pp. 287–298 (1989) 95

[31] Gilbert J.R., Schreiber R.: *Highlyparallel sparse Cholesky factorization.* SIAMJournal onScientific and Statistical Computing, Vol. 13 pp. 1151–1172 (1992) 95

[32] Giraud L., Haidar A., Watson L.T.: *Mixed-Precision Preconditioners in Parallel Domain Decomposition Solvers.* Domain Decomposition Methods in Science and Engineering XVII. Lecture Notes in Computational Science and Engineering, Vol. 60, Springer, Berlin, (2008) 117

[33] Gruttmann F., Wagner W.: *A coupled two-scale shell model with applications to layered structures.* Int. J. Num. Meth. Engng, Vol. 94, pp. 1233–1254 (2013) 11, 41

[34] Gueye I., Arem S. E., Feyel F., Roux F.-X., Cailletaud G.: *A new parallel sparse direct solver: Presentation and numerical experiments in large-scale structural mechanics parallel computing.* Int. J. Num. Meth. Engng, Vol. 88 (4), pp. 370–384 (2011) 110, 141

[35] Gupta A., Karypis G., Kumar V.: *A Shared- and distributed-memory parallel general sparse direct solver* IEEE Trans. Parallel Distrib. Syst., Vol. 8(5), pp. 502–520 (1997) 95

[36] Gupta A.: *A Shared- and distributed-memory parallel general sparse direct solver* Applicable Algebra in Engineering, Communication and Computing, Vol. 18(3), pp. 263–277 (2007) 96

[37] Gupta A.: *WSMP: Watson Sparse Matrix Package. Part I – direct solution of symmetric systems. Version 15.01* IBM Research Report, Watson (2015) , 10, 12, 14, 73, 95, 104

[38] Gupta A.: *Parallel Sparse Direct Methods: A short tutorial.* IBM Research Report, Watson (2010) 58

[39] Guo X., Gorman G., Sunderland A., Ashworth M.: *Developing Hybrid OpenMP-MPI Parallelism for Fluidity-ICOM - Next Generation Geophysical Fluid Modelling Technology.* Cray User Group 2012: Greengineering the Future (CUG2012), Stuttgart, Germany, 29th April-3rd May 2012 41

[40] Gustafson J.L.: *Reevaluating Amdahl's law.* Communications of the ACM, Vol. 31 (5), pp. 532–533 (1988) 18

[41] Hogg J.D., Scott J.A.: *A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems.* ACM Transactions on Mathematical Software, Vol. 37 (2), 1-–24, 2010. 93

[42] Hogg J.D., Scott J.A.: *A note on the solve phase of a multicore solver.* Technical Report TR-RAL-2010-007 (2010) 97

[43] Hogg J.D., Scott J.A.: *An indefinite sparse direct solver for multicore machines.* Technical Report TR-RAL-2010-011 (2010) , 12, 13, 73

[44] HYPRE, www.llnl.gov/casc/hypre 164, 165

[45] HSL 2013: *A collection of Fortran codes for large scale scientific computation,* http://www.hsl.rl.ac.uk/. 9, 161

[46] Farhat C., Roux F-X., *A Method of Finite Element Tearing and Interconnecting and its Parallel Solution Algorithm.* Int. J. Num. Meth. Engng, **32**: 1205-1227, 1991. 119, 120

[47] Farhat C., et al., *Optimal convergence properties of the FETI domain decomposition method.* Computer Methods in Applied Mechanics and Engineering, Vol. 115, 365–385, 1994. 122

[48] Farhat C., et al., *The two-level FETI method Part II: Extension to shell problems, parallel implementation and performance results.* Computer Methods in Applied Mechanics and Engineering., Vol. 155(1-2), 153–179, 1998. 12, 121, 122, 123

[49] Farhat C., et al., *FETI-DP: a dual-primal unified FETI method part I: A faster alternative to the two-level FETI method.* Int. J. Num. Meth. Engng, Vol. 50, 1523–1544, 2001. 123, 125

[50] Goddeke D., Strzodka R., Turek S.: *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations.* International Journal of Parallel, Emergent and Distributed Systems, Vol. 22 (4), 221–256, 2007. 93

[51] Ikuno S., Takayama T., Kamitani A.: *Application of parallel processing technique to shielding current analysis on HTS thin film.* Physica C 463-465, 1013-1016 (2007) 40

[52] Intel Inspector 2015 Update 1. 43

[53] Jarzebski P., Wisniewski K.: *On parallelization of the loop over elements for composite shell computations.* 39th Solid Mechanics Conf. (SOLMECH 2014) Zakopane, Sept. 1-5, 2014, Book of Abstracts (Z. L. Kowalewski ed.) p. 153-4 41

[54] Jarzebski P., Wisniewski K., Taylor R.L.: *On paralelization of the loop over elements in FEAP.* Computational Mechanics, Vol. 56 (1), pp. 77–86 (2015) 12

[55] Jarzebski P., Wisniewski K.: *Performance of the parallel FEAP in calculations of effective material properties using RVE.* Advances in Mechanics: Theoretical, Computational and Interdisciplinary Issues, Kleiber et al. (Eds), Taylor & Francis Group, London, pp. 241–244 (2016) 13, 14

[56] Jarzebski P., Wisniewski K.: *Application of partial factorization for domain decomposition solver.* W przygotowaniu 13

[57] Higham N.J.: *Accuracy and Stability of Numerical Algorithms.* SIAM, 1996 92

[58] Karp A.H., Flatt H.P.: *Measuring parallel processor performance.* Communications of the ACM, Vol. 33 (5), pp. 539–543 (1990) 20

[59] Karypis G., Kumar V.: *A fast and high quality multilevel scheme for partitioning irregular graphs.* International Conference on Parallel Processing, 113-122 (1995) 77, 154

[60] Karypis G., Kumar V.: *Multilevel k-way Partitioning Scheme for Irregular Graphs.* Journal of Parallel and Distributed Computing, Vol.48(1), 96-129 (1998) 77, 153

[61] Kratzer S.G., Cleary A.J.: *Sparse matrix factorization on simd parallel computers.* Graph Theory and Sparse Matrix Computations, Springer-Verlag, New York, pp. 211–228 (1993) 95

[62] Klinkel S., Gruttmann F., Wagner W.: *A continuum based 3d-shell element for laminated structures.* Computers & Structures, Vol. 71, 43-62 (1999) 11

[63] Kocak S., Akay H.U.: *Parallel Schur complement method for large-scale systems on distributed memory computers.* Applied Mathematical Modelling, Vol. 25 (10), 873–886 (2001) 106

[64] Kung H.T.: *Memory requirements for balanced computer architectures.* In International Symposium on Computer Architecture (ISCA), 49–54 (1986) 24

[65] Anderson E., et al.: *LAPACK Users' Guide* SIAM, Philadelphia, 1999 161

[66] Li G., Coleman T.F.: *A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors.* SIAM J. Sci. and Stat. Comput., Vol. 10(2), 382–396 (1989) 95

[67] Langou J., Langou J., Luszczek P., Kurzak J., Buttari A.; Dongarra J. *Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems).* SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, (2006) 92

[68] Li X.S., Demmel J.W.: *SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems.* ACM Transactions on Mathematical Software, Vol. 29(2), 110–140 (2003) 95

[69] Li X.S.: *Factorization-based Sparse Solvers and Preconditioners.* 4th Gene Golub SIAM Summer School, (2013) 95

[70] Lo S.H.: *Parallel Delaunay triangulation - Application to two dimensions.* Finite Elements in Analysis and Design, Vol. 62, 37-48 (2012) 41

[71] MacNeal R.H., Harder R.L.: *A proposed standard set of problems to test finite element accuracy.* Finite Element in Analysis and Design, Vol. 1, pp. 3–20 (1985) 161

[72] Manne F., Hafsteinsson H.: *Efficient sparse Cholesky factorization on a massively parallel SIMD computer* SIAM J. Sci. COMPUT., Vol. 16(4), pp. 934–950 (1995) 95

[73] Markall G.R., Slemmer A., Ham D.A., Kelly P.H.J., Cantwell C.D., Sherwin S.J.: *Finite element assembly strategies on multi-core and many-core architectures.* Int. J. Numer. Meth. Fluids, Vol. 71, pp. 80–97 (2013) 42

[74] METIS, http://glaros.dtc.umn.edu/gkhome/metis/metis/overview 9, 65, 74, 76, 153, 164

[75] LaSalle D., Karypis G.: *Efficient nested dissection for multicore architectures.* EuroPar (2015) 76

[76] Intel MKL, http://software.intel.com/en-us/intel-mkl 9, 12, 25, 43, 74

[77] Moler, C. B.: *Iterative Refinement in Floating Point.* J. ACM, Vol. 2, pp. 316–321 (1967) 117

[78] MUMPS: a MUltifrontal Massively Parallel sparse direct Solver, http://mumps.enseeiht.fr/ 9, 95

[79] Moto Mpong S., de Montleau P., Godinas A., Habraken A.M.: *Acceleration of finite element analysis by parallel processing.* Proceedings of the 5th international ESAFORM Conference on Material Forming, 47-50 (2002) 40

[80] MPI, http://www.mpi-forum.org , 12, 13, 34, 43

[81] MPI Tutorial, http://mpitutorial.com/ 35

[82] Nowak M., Nowak Z., Pęcherski R.B., Potoczek M. i Śliwa R.E.: *On the reconstruction method of ceramic foam structures and the methodology of young modulus determination,* Archive of Metallurgy and Materials, Vol. 58, pp. 1219-1222 (2013). 130

[83] OpenMP, http://openmp.org , 11, 12, 13, 43, 46

[84] Ofenbeck G., Steinmann R., Cabezas V.C., Spampinato D.G., Püschel M.: *Applying the Roofline Model.* Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 76-85 (2014) 25

[85] Pantale O.: *Parallelization of an object-oriented FEM dynamics code: influence of the strategies on the speedup.* Advances in Engineering Software, Vol. 36, 361-373 (2005) 11, 41

[86] PAPI, http://icl.cs.utk.edu/papi/ 10, 24, 169

[87] Intel MKL PARDISO, https://software.intel.com/articles/intel-mkl-pardiso 73

[88] Paris J., Colominas I., Navarrina F., Casteleiro M.: *Parallel computing in topology optimization of structures with stress constraints.* Computers & Structures, Vol. 125, 62-73 (2013) 41

[89] Petra C.G., Schenk O., Lubin M., Gaertner K.: *An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization.* SIAM J. Sci. Comput., Vol. 36 (2), C139–C162 (2014) 52, 99, 134, 138

[90] PETSc, http://www.mcs.anl.gov/petsc/ 10, 164

[91] Press W.H., et al.: *Numerical Recipes in Fortran 77* Cambridge Univeristy Press, 1999 161

[92] Karbowski A., Niewiadomska-Szynkiewicz E.: *Programowanie równoległe i rozproszone.* Oficyna Wydawnicza Politechniki Warszawskiej. Warszawa; 2009. 15, 16, 26, 34

[93] Robert Y., Tourancheau B., Villard G.: *Data allocation strategies for the Gauss and Jordan algorithms on a ring of processors.* Information Processing Letters, Vol. 31(1), pp. 21–29 (1989) 95

[94] Saad Y.: *A flexible inner-outer preconditioned GMRES algorithm.* SIAM J. Scientific and Statistical Computing, Vol. 14(2), pp. 461–469 (1993) 162

[95] Saad Y.: *SPARSKIT: a basic tool kit for sparse matrix computations.* Tech. Rep. CSRD TR 1029, CSRD, University of Illinois, Urbana (1994) 60

[96] Saad Y.: *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, 2003 58, 71

[97] Savic S.V., Ilic A.Z., Notaros B. M., Ilic M.M.: *Acceleration of Higher Order FEM Matrix Filling by OpenMP Parallelization of Volume Integrations.* 20th Telecommunications forum TELFOR 2012 , 370-384 (2012) 41

[98] Schenk O., Gärtner K. Fichtner W.: *Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors.* BIT, Vol. 40(1), pp. 158–176 (2000) 10, 12, 72

[99] Schoof L.A., Yarberry V.R.: *EXODUS II: A Finite Element Data Model* Sandia Report, Sandia National Laboratories, Albuquerque, Livermore (1994) 156

[100] Simo J.C., Tarnow N.: *On a stress resultant geometrically exact shell model. Part VI. 5/6 dof treatments.* Int. J. Num. Meth. Engng, Vol. 34, pp. 117–164 (1992) 54

[101] Skillicorn D.B.: *A Taxonomy for Computer Architectures.* Journal Computer, Vol.21, pp.46-57 (2001) 16

[102] STREAM, https://www.cs.virginia.edu/stream/ 25

[103] Taylor R.L.: *Finite element analysis of linear shell problems.* In: Whiteman J.R. (ed.) "The Mathematics of Finite Elements and Applications VI. MAFELAP 1987". Academic Press, London, 1988 54

[104] Trefethen L.N., Bau D. III,: *Numerical Linear Algebra* SIAM, Philadelphia, 1997 58, 60

[105] Tang G., D'Azevedo E.F., Zhang F., Parker J.C., Watson D.B., Jardine P.M.: *Application of a hybrid MPI/OpenMP approach for parallel groundwater model calibration using multi-core computers.* Computers and Geosciences, Vol. 36, 1451-1460 (2010) 11, 41

[106] Tallec P., Mandel J., Vidrascu M.: *A Neumann-Neumann domain decomposition algorithm for solving plate and shell problems.* SIAM J. Numer. Anal., Vol. 35(2), 836–867 (2010) 12

[107] Terboven C., Spiegel A., an Mey D., Gross S., Reichelt V.: *Experiences with the OpenMP Parallelization of DROPS, a Navier-Stokes Solver Written in C++.* OpenMP Shared Memory Parallel Programming. Lecture Notes in Computer Science, Vol. 4315, 95-106 (2008) 11, 41

[108] Merkley K.G., Meyers R.J.: *Verde.* Version 2.6 (2001) 130, 136

[109] Voigt A., Witkowski T.: *Hybrid parallelization of an adaptive finite element code.* Kybernetika, Vol.46 (2), pp.316-327 (2010) 41

[110] Dodds R., et al.: *Warp3D.* Release 17.5.3 (2014) 40, 42, 43, 53, 57, 165

[111] Węglewski W., Bochenek K., Basista M., Schubert T., Jehring U., Litniewski J., Mackiewcz S.: *Comparative assessment of Young's modulus measurements of metal–ceramic composites using mechanical and non-destructive tests and micro-CT based computational modeling.* Computational Materials Science, Vol. 77, pp. 19-30 (2013) 135, 136

[112] Wilkinson J. H.: *Rounding Errors in Algebraic Processes* Englewood Cliffs, NJ: Prentice Hall, 1963 91, 92

[113] Williams S., Waterman A., Patterson D.: *Roofline: an insightful visual performance model for multicore architectures* Communications of the ACM, Vol. 52(4), pp. 65-76 (2009) 23, 24

[114] Wisniewski K.: *Finite Rotation Shells. Basic Equations and Finite Elements for Reissner Kinematics.* Springer, 2010 54

[115] Wisniewski K., Turska E.: *Four-node mixed Hu-Washizu shell element with drilling rotation.* Int. J. Num. Meth. Engng, Vol. 90, pp. 506–536 (2012) 54, 161

[116] Wong H. Ivy Bridge Power Consumption, http://blog.stuffedcow.net/2012/07/ivy-bridge-power-consumption/ 72

[117] Yamazaki I., Li X.S.: *On techniques to improve robustness and scalability of a parallel hybrid linear solver.* Proceeding VECPAR'10 Proceedings of the 9th international conference on High performance computing for computational science, pp. 421–434 (2010) 117

[118] Yannakakis M.: *Computing the Minimum Fill-In is NP-Complete.* SIAM J. Alg. & Disc. Meth., Vol. 2, pp. 77–79 (1981) 70

[119] Zienkiewicz O.C, Taylor R.L., Fox D.D.: *The Finite Element Method for Solid and Structural Mechanics. Seventh Edition.* Butterworth-Heinemann (2013) 42

Dodatki

A Dodatek A
Instrukcja dodawania elementu do ompFEAP

1. Obliczenia wykonane dla elementu nie mogą zależeć od obliczeń dla elementu poprzedniego (poprzedniego w sensie wykonania sekwencyjnego). To znaczy, że niezależnie od kolejności elementów, obliczenia dadzą tą samą macierz wynikową. To kluczowy warunek, aby można było wykonać pętle po elementach w sposób równoległy.

2. Procedura elementu i każda procedura lub funkcja wywołana z procedury elementu, nie może zawierać atrybutu „SAVE” z języka Fortran, patrz Kod A.1. To podstawowy warunek, aby zapewnić bezpieczeństwo wątkowe dla wywołanej procedury lub funkcji (ang. „thread-safe”).

3. Jedynie tablice „s” i „p” z parametrów wejściowych mogą być w bezpieczny sposób modyfikowane. Pozostałych tablic wejściowych nie należy modyfikować.

4. Jeśli potrzebne jest użycie pliku COMMON (własnego bądź wbudowanego w FEAPa), w którym będą przechowywane dane specyficzne dla elementu, tzn. dla każdego elementu będą w danej zmiennej przechowywane inne dane (nie są to dane wspólne dla danej klasy elementów), należy:

   (a) W pliku COMMON oznaczyć dany blok COMMON jako THREADPRIVATE, np. „c$omp THREADPRIVATE(/eldata/)”, patrz np. plik „includeOMP/eldata.h” Kod A.2.

   (b) W pliku „programOMP/pform.f” należy dodać linię „include ’nazwa.h’ ”

   (c) W pliku „programOMP/pform.f” w sekcji „COPYIN” dyrektywy „PARALLEL DO” należy dodać odpowiednią nazwę bloku COMMON (okolice linii nr 208), patrz Kod A.3.

5. Program należy skompilować z włączonym OpenMP, flaga -openmp (Intel) -fopenmp (GCC).

6. Dla równomiernego rozkładu między rdzenie procesora należy ustawić THREAD AFFINITY. W zależności od kompilatora, należy ustawić zmienną środowiskową:

   (a) Intel - „KMP_AFFINITY=granularity=fine,scatter”.

   (b) GCC - „GOMP_CPU_AFFINITY="0-23:2" ”.

Kod A.1: Zabroniony atrybut „SAVE”!

```
subroutine elem()
         real*8     a,b
c There should not be any save attribute for variables
c      save

     compute matrix

end
```

Kod A.2: Zawartość pliku „includeOMP/eldata.h”

```
      real*8          dm
      integer            n,ma,mct,iel,nel,pstyp,eltyp
      common /eldata/ dm,n,ma,mct,iel,nel,pstyp,eltyp
c$OMP THREADPRIVATE(/eldata/)
```

Kod A.3: Fragment pliku „programOMP/pform.f”

```
c     Loop over active elements
c$OMP PARALLEL DO NUM_THREADS(nthreads),
c$OMP& PRIVATE(un,dun,ul,xl,tl,ld,temp,erotflg,nov,msflg,
c$OMP& rel,nrot,jj,i,mdfl,p,n1
c$OMP& ),
c$OMP& LASTPRIVATE(s),
c$OMP& COPYIN(
c$OMP& /eldata/,/strnum/,
c$OMP& /crotas/,/erotas/,/ddata/,
c$OMP& /npoints/,/pointer/,/cdata/
c$OMP& )
      do n1 = nn1,nn2,nn3
```

B Dodatek B
Algorytm podziału grafu

Aby zadanie mogło być rozwiązane na wielu węzłach obliczeniowych musi zostać ono podzielone. Podział rozumiany jest jako podział siatki zadania na wiele części - dokładnie tyle, ile jest dostępnych węzłów. Do tego celu można wykorzystać algorytmy dzielące grafy na wiele części. Siatkę na graf można zamienić na dwa sposoby:

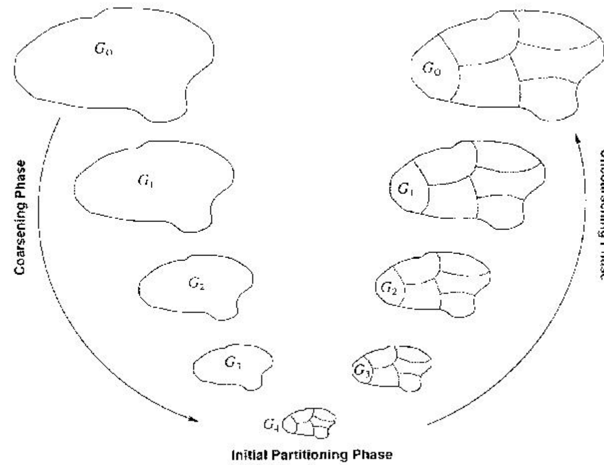
1. **na graf dualny** w tym przypadku elementy siatki stają się wierzchołkami grafu, a krawędzie są ustalane na podstawie minimalnej liczby wspólnych wierzchołków między elementami w siatce. Tzn. jeżeli minimalna liczba zostanie ustalona na 2, to między wierzchołkami grafu będzie krawędź tylko i tylko wtedy gdy elementy w siatce posiadają co najmniej dwa wspólne wierzchołki.

2. **na graf wierzchołkowy (ang. *nodal graph*)** tutaj wierzchołkami w grafie stają się wszystkie wierzchołki w siatce, krawędzie natomiast zostaje ustalona dla wszystkich wierzchołków w danym elemencie. Tzn. Wierzchołek w grafie łączy się z innym wierzchołkiem w grafie, jeśli wierzchołki na siatce należą do tego samego elementu (przynajmniej jednego).

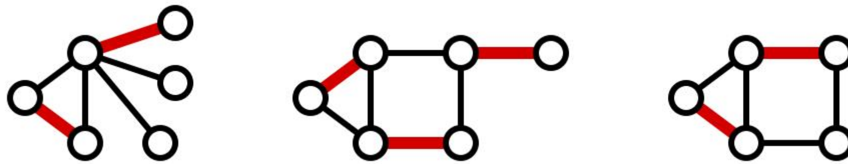
Jednym z najpopularniejszych i najwydajniejszych algorytmów do podziału grafu jest algorytm „Multilevel k-way partitioning” przedstawiony w pracy Karypisa i Kumara [60]. Algorytm ten został zaimplementowany przez jego autorów w programie METIS [74].

Idea algorytmu jest następująca. Dzielenie grafu łatwiej wykonać na mniejszym grafie niż na większym, a więc graf jest zmniejszany, do tego stopnia, aż będzie można wykorzystać standardowe algorytmy do podziału grafu. Zmniejszanie grafu polega na grupowaniu wierzchołków i tworzenie z nich jednego wierzchołka. Gdy mniejszy graf zostanie podzielony, to następuje powrót do grafu oryginalnego poprzez odwrócenie procesu zmniejszania. Podczas powrotu trzeba jednak sprawdzić, czy nie da się poprawić podziału poprzez wymianę wierzchołków miedzy różnymi częściami. Te trzy etapy przedstawiono na rys. B.1.

1. **Etap zmniejszania** W tym etapie wykorzystuje się pojęcie „skojarzenia”. Skojarzenie to taki zbiór krawędzi grafu, z którego żadne dwie krawędzie nie mają wspólnego wierzchołka. Przykładowe skojarzenia są przedstawione na rys. B.2. Każda krawędź ze skojarzenia staje się wierzchołkiem nowego, mniejszego grafu. Jego waga to suma wag wierzchołków należących do krawędzi. Krawędzie, które nie należały do skojarzenia przechodzą do nowego grafu bez zmian. Najprostszym sposobem na znalezienie maksymalnego skojarzenia jest algorytm losowy. Należy wybrać losowo jeden wierzchołek i jednego z jego sąsiadów, który także nie został jeszcze skojarzony. Krawędź między tymi dwoma wierzchołkami zostaje dodana do skojarzenia, a wierzchołki oznaczone jako skojarzone. Losowanie jest kontynuowane, aż nie będzie wierzchołków, z których będzie można wybierać. Wybrany zbiór krawędzi, to maksymalne skojarzenie. Ten algorytm można ulepszyć poprzez wybieranie sąsiada na podstawie wag, które są do niego przypisane. Wybieranie największych wag powoduje zmniejszenie wag krawędzi, które będą docelowo łączyły podziały w grafie.



Rysunek B.1: Etapy algorytmu „Multilevel k-way partitioning”.



Rysunek B.2: Przykładowe skojarzenia w grafie zostały oznaczone czerwonym kolorem.

2. **Etap dzielenia** Ten etap wykorzystuje algorytm „Multilevel bisection algorithm”, który jest uproszczeniem opisywanego algorytmu dzielącego graf na dwie części. Algorytm „Multilevel bisection algorithm” został opisany w [59]. Także posiada trzy etapy, z tym że etap dzielenia jest wykonany za pomocą algorytmu bisekcji widmowej lub heurystycznym powiększaniem grafu z wyborem wagi, bądź heurystycznym powiększaniem grafu z wyborem ilości cięć. Algorytm bisekcji widmowej polega na utworzeniu macierzy $\mathbf{L} = \mathbf{D} - \mathbf{A}$, gdzie macierz $\mathbf{D}$ to macierz stopni (macierz diagonalna, na której i-ty wiersz ma wartość stopnia wierzchołka i-tego), a macierz $\mathbf{A}$ to macierz sąsiedztwa (jedynka w i-tym wierszu i j-tej kolumnie oznacza krawędź między wierzchołkami i-tym i j-tym). Dla tak utworzonej macierzy $\mathbf{L}$ poszukiwane są wartości własnych i odpowiadającym im wektorom własnym. Należy wybrać drugą najmniejszą wartość własną i wektor własny jej odpowiadający. Jeżeli i-ta współrzędna wybranego wektora własnego jest dodatnia, to i-ty wierzchołek należy do pierwszego podziału, jeżeli ujemna to do drugiego. Heurystyczne powiększanie grafu polega na losowym wybraniu wierzchołka do pierwszego podziału i przeglądaniu grafu wszerz od tego wierzchołka. Trzeba dodać kolejne przeglądane wierzchołki, aż połowa wartości wag wierzchołków zostanie osiągnięta w przypadku wyboru wag. Z wyborem ilości cięć dobiera się kolejne wierzchołki w ten sposób, aby zmniejszyć ilość cięć między dwoma podziałami.

3. **Etap powrotu** Ten etap jest wykonywany na podstawie etapu pierwszego, tzn. wierzchołek, który był stworzony poprzez połączenie kliku wierzchołków zostaje podzielony. W ten sposób może się okazać, że powstały graf ma lepszy podział i

trzeba go znaleźć. Do tego stosuje się algorytm iteracyjny. Po pierwsze dla każdego wierzchołka liczony jest zysk (strata), gdyby był on przeniesiony do innego podziału. Z wierzchołków wybrany zostaje ten z największym zyskiem i zostaje przeniesiony. W tej iteracji już nie będzie brany pod uwagę. Później aktualizowane są zyski (straty). Powyższe kroki są powtarzane, aż przez x przesunięć nie było zmniejszenia liczby cięć (zazwyczaj $x = 50$). Iteracja zostaje powtórzona, jeżeli spowodowała zmniejszenie liczby cięć. Liczbę iteracji można także ograniczyć, autorzy algorytmu proponują stosowanie jednej iteracji. Kluczowe jest liczenie zysku i jego przechowywanie. Autorzy liczą zysk już podczas podziału, a później aktualizuje się zysk tylko dla tych wierzchołków, które są na krawędziach podziału. Wybór do którego podziału należy przenieść wierzchołek można robić: albo zachłannie (wybrany zostaje taki podział, gdzie waga najbardziej się zmniejszy), albo poprzez wybór podziału gdzie stopień zewnętrzny wierzchołka (taki, który nie jest w danym podziale) jest największy i jest zachowany pewien warunek równowagi.

C Dodatek C Program Verde

Verde (ang. *Verification of Discrete Elements*) to program zaprojektowany do weryfikowania własności siatki elementów skończonych w formacie Exodus II [99]. Program ten został wykorzystany w rozdz. 6.1 do sprawdzenia siatki korundowej pianki oraz w rozdz. 6.2 do zweryfikowania modelu ceramicznego kompozytu. Verde używa szerokiej gamy algorytmów weryfikujących do zanalizowania indywidualnych i połączonych własności siatki i pomaga użytkownikowi w detekcji różnych problemów.

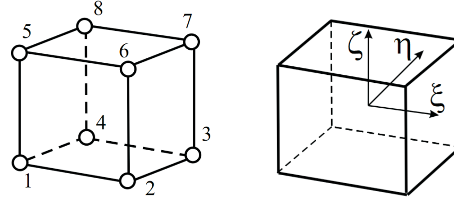
Verde potrafi przeprowadzić weryfikację siatki w trzech obszarach:

1. **Metryki elementów** Charakterystyka pojedynczego elementu może być zweryfikowana poprzez obliczenie powszechnych metryk w zależności od typu elementu. Verde wylicza jedną lub więcej metryk dla każdego wspieranego typu elementu. Dla każdej metryki, minimum, maksimum, średnia wartość i odchylenie standardowe jest śledzone, a także identyfikowane są te elementy, dla których minimum i maksimum jest osiągane. Wszystkie elementy, których wartości metryk wychodzą poza zakres zdefiniowany przez użytkownika zostają oznaczone jako niepoprawne.

2. **Sprawdzenie topologii** Topologia siatki jest dokładnie analizowana, w poszukiwaniu błędów w połączeniach siatki lub ciągłości, które mają wpływ na poprawność siatki. Verde raportuje liczbę zewnętrznych obszarów siatki, sprawdza czy siatka jest rozmaitością topologiczną, a jeśli tak oblicza liczbę Eulera i wyprowadza prawdopodobną całościową topologię. Dodatkowo, sprawdza połączone czworokątne powierzchnie w siatce, które mają wspólne tylko trzy węzły.

3. **Sprawdzenie interfejsu** Zewnętrzne powierzchnie siatki są dokładnie przebadane przez Verde. Elementy zewnętrzne są sprawdzane pod względem zgodności. W ten sposób potencjalne problemy, takie jak nieprawidłowe połączenia w siatce i wielokrotnie zdyskretyzowane obszary mogą być znalezione. Można również wyeksportować do pliku (w formacie Exodus II) informacje, które zawierają zewnętrzne ściany siatki, znalezione krawędzie modelu, oraz nieprawidłowe elementy.

D Dodatek D Trójwymiarowy izoparametryczny element ośmiowęzłowy



Rysunek D.1: Trójwymiarowy element przestrzenny ośmiowęzłowy i jego reprezentacja w lokalnym układzie współrzędnych.

Izoparametryczny element przestrzenny ośmiowęzłowy został przedstawiony na rys. D.1. Określenie „izoparametryczny” oznacza, że geometria i pole przemieszczeń w elemencie są zdefiniowane w sposób parametryczny i są interpolowane za pomocą tych samych funkcji. Funkcje kształtu użyte do interpolacji to wielomiany lokalnych współrzędnych ξ, η oraz ζ ($-1 \leqslant \xi, \eta, \zeta \leqslant 1$):

$$\begin{aligned} \mathbf{u} &= \mathbf{N}\mathbf{q} \\ \mathbf{u} &= [u\ v\ w]^T \\ \mathbf{q} &= [u_1\ v_1\ w_1\ u_2\ v_2\ w_2\ \dots\ u_8\ v_8\ w_8\ ]^T \end{aligned} \tag{D.1}$$

$$\begin{aligned} \mathbf{x} &= \mathbf{N}\mathbf{x}^e \\ \mathbf{x} &= [x\ y\ z]^T \\ \mathbf{x}^e &= [x_1\ y_1\ z_1\ x_2\ y_2\ z_2\ \dots\ x_8\ y_8\ z_8\ ]^T. \end{aligned} \tag{D.2}$$

Gdzie u, v, w to przemieszczenia w punkcie o lokalnych współrzędnych ξ, η, ζ; u_i, v_i, w_i to wartości przemieszczeń w węzłach; x, y, z to współrzędne punktu, a x_i, y_i, z_i to współrzędne węzła. Macierz funkcji kształtu przedstawia się następująco:

$$\mathbf{N} = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & \dots & N_8 & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & \dots & 0 & N_8 & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & \dots & 0 & 0 & N_8 \end{bmatrix}. \tag{D.3}$$

A poszczególne funkcje kształtu są równe:

$$\begin{gathered} N_i = \frac{1}{8}(1+\xi_0)(1+\eta_0)(1+\zeta_0), \\ \xi_0 = \xi\xi_i,\ \eta_0 = \eta\eta_i,\ \zeta_0 = \zeta\zeta_i. \end{gathered} \tag{D.4}$$

Wektor odkształceń $\boldsymbol{\epsilon}$ składa się z sześciu komponentów tensora odkształceń:

$$\boldsymbol{\epsilon} = [\epsilon_x\ \epsilon_y\ \epsilon_z\ \gamma_{xy}\ \gamma_{yz}\ \gamma_{zx}], \tag{D.5}$$

gdzie:

$$\gamma_{xy} = 2\epsilon_{xy}\, \gamma_{yz} = 2\epsilon_{yz}\, \gamma_{zx} = 2\epsilon_{zx}. \tag{D.6}$$

Macierz odkształceniowo-przemieszczeniowa dla trójwymiarowych elementów przedstawia się następująco:

$$\mathbf{B} = \mathbf{DN} = [\mathbf{B}_1\, \mathbf{B}_2\, \dots\, \mathbf{B}_8],$$
$$\mathbf{B}_i = \begin{bmatrix} \partial N_i/\partial x & 0 & 0 \\ 0 & \partial N_i/\partial y & 0 \\ 0 & 0 & \partial N_i/\partial z \\ \partial N_i/\partial y & \partial N_i/\partial x & 0 \\ 0 & \partial N_i/\partial z & \partial N_i/\partial y \\ \partial N_i/\partial z & 0 & \partial N_i/\partial x \end{bmatrix}. \tag{D.7}$$

Pochodne funkcji kształtu dla współrzędnych globalnych są otrzymywane ze wzorów:

$$\begin{bmatrix} \partial N_i/\partial x \\ \partial N_i/\partial y \\ \partial N_i/\partial z \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \partial N_i/\partial \xi \\ \partial N_i/\partial \eta \\ \partial N_i/\partial \zeta \end{bmatrix}, \tag{D.8}$$

a Jakobian jest równy

$$\mathbf{J} = \begin{bmatrix} \partial x/\partial \xi & \partial y/\partial \xi & \partial z/\partial \xi \\ \partial x/\partial \eta & \partial y/\partial \eta & \partial z/\partial \eta \\ \partial x/\partial \zeta & \partial y/\partial \zeta & \partial z/\partial \zeta \end{bmatrix}. \tag{D.9}$$

Pochodne cząstkowe dla x, y, z względem ξ, η, ζ mogą być znalezione poprzez różniczkowanie przemieszczeń wyrażonych za pomocą funkcji kształtu i wartości przemieszczeń w węzłach:

$$\begin{aligned} \frac{\partial x}{\partial \xi} &= \sum \frac{\partial N_i}{\partial \xi} x_i,\ \frac{\partial x}{\partial \eta} = \sum \frac{\partial N_i}{\partial \eta} x_i,\ \frac{\partial x}{\partial \zeta} = \sum \frac{\partial N_i}{\partial \zeta} x_i, \\ \frac{\partial y}{\partial \xi} &= \sum \frac{\partial N_i}{\partial \xi} y_i,\ \frac{\partial y}{\partial \eta} = \sum \frac{\partial N_i}{\partial \eta} y_i,\ \frac{\partial y}{\partial \zeta} = \sum \frac{\partial N_i}{\partial \zeta} y_i, \\ \frac{\partial z}{\partial \xi} &= \sum \frac{\partial N_i}{\partial \xi} z_i,\ \frac{\partial z}{\partial \eta} = \sum \frac{\partial N_i}{\partial \eta} z_i,\ \frac{\partial z}{\partial \zeta} = \sum \frac{\partial N_i}{\partial \zeta} z_i. \end{aligned} \tag{D.10}$$

Przejście całki z globalnego układu współrzędnych do lokalnego układu jest zrobione wraz z użyciem wyznacznika Jakobianu

$$dV = dxdydz = |\mathbf{J}| d\xi d\eta d\zeta. \tag{D.11}$$

Równanie równowagi dla elementu przyjmuje postać

$$\mathbf{Kq} = \mathbf{p}, \tag{D.12}$$

gdzie macierz sztywności jest równa

$$\mathbf{K} = \int_V \mathbf{B}^T \mathbf{C} \mathbf{B} dV, \tag{D.13}$$

a wektor sił (obciążenie objętościowe i powierzchniowe)

$$\mathbf{p} = \int_V \mathbf{N}^T \mathbf{p}^V dV + \int_S \mathbf{N}^T \mathbf{p}^S dS. \tag{D.14}$$

Macierz sprężystości jest równa

$$\mathbf{C} = \begin{bmatrix} \lambda + 2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda + 2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda + 2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda & 0 & 0 \\ 0 & 0 & 0 & 0 & \lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & \lambda \end{bmatrix}, \tag{D.15}$$

gdzie λ oraz μ to stałe sprężystości Lame'go, które mogą być wyrażone poprzez moduł sprężystości E i współczynnik Poisson'a ν:

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)}. \tag{D.16}$$

Całkowanie macierzy sztywności dla elementu izoparametrycznego jest przeprowadzone w lokalnym układzie współrzędnych ξ, η, ζ:

$$\mathbf{K} = \int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} \mathbf{B}(\xi, \eta, \zeta)^T \mathbf{C} \mathbf{B}(\xi, \eta, \zeta)] |\mathbf{J}| d\xi d\eta d\zeta. \tag{D.17}$$

Aby takie całkowanie przeprowadzić, stosuje się trzykrotnie jednowymiarową kwadraturę Gaussa, co prowadzi do następującej procedury całkowania numerycznego:

$$I = \int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta, \zeta) d\xi d\eta d\zeta = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} f(\xi_i, \eta_j, \zeta_k) w_i w_j w_k \tag{D.18}$$

W pracy została użyta reguła całkowania $2 \times 2 \times 2$.

E Dodatek E Redukcja macierzy elementowych

Mieszane/wzbogacone (ang. *mixed/enhanced*) skończone elementy powłokowe mogą zawierać dodatkowe parametry lokalne, które muszą zostać skondensowane, aby zredukować wielkość stycznej macierzy elementowej do standardowej wielkości.

Liczba parametrów elementowych może być bardzo duża, czasami przekracza ona liczbę zmiennych węzłowych związanych z elementem. Kondensacja zazwyczaj jest wykonana poprzez eliminację poszczególnych typów parametrów jeden po drugim poprzez odwracanie odpowiednich podmacierzy. Takie podejście musi być dostosowywane do każdego nowego sformułowania elementu, co jest oczywistą wadą takiego rozwiązania.

Alternatywnym podejściem jest kondensacja wszystkich parametrów jednocześnie, co wymaga obliczenia uzupełnienia Schura. Aby przyspieszyć ten proces, zostanie wykorzystana technika *częściowej faktoryzacji* opisana w rozdz. 5.2.3, która zastąpi standardowe obliczenia uzupełnienia Schura. W tym dodatku została przetestowana ta technika i porównana wydajność dla różnych elementów powłokowych, solid-shell oraz elementów 3D.

Dla rozważanej klasy elementów skończonych, podstawowy funkcjonał F zależy od węzłowych przemieszczeń i rotacyjnych parametrów $\mathbf{u}_I$ oraz od mnożników elementowych $\mathbf{q}$. Dla kinematycznych nieliniowych problemów, warunek stacjonarności dla $F(\mathbf{u}_I, \mathbf{q})$ wyznacza układ równań liniowych dla elementu, $\mathbf{r}_u(\mathbf{u}_I, \mathbf{q}) = \mathbf{0}$ i $\mathbf{r}_q(\mathbf{u}_I, \mathbf{q}) = \mathbf{0}$. Zlinearyzowana forma tych równań jest następująca

$$\begin{bmatrix} \mathbf{K} & \mathbf{L} \\ \mathbf{L}^T & \mathbf{K}_{qq} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{u}_I \\ \Delta\mathbf{q} \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_u \\ \mathbf{r}_q \end{bmatrix}, \tag{E.1}$$

gdzie $\mathbf{K} \doteq \partial\mathbf{r}_u/\partial\mathbf{u}_I$, $\mathbf{L} \doteq \partial\mathbf{r}_u/\partial\mathbf{q}$ oraz $\mathbf{K}_{qq} \doteq \partial\mathbf{r}_q/\partial\mathbf{q}$. Należy zwrócić uwagę, że $\mathbf{K}$ i $\mathbf{K}_{qq}$ są symetryczne i w ogólności nieokreślone.

Aby wyeliminować mnożniki na poziomie elementu i aby zredukować wielkość macierzy stycznej do standardowej wielkości, zdefiniowanej jako liczba węzłów w elemencie i liczba stopni swobody na węzeł, trzeba obliczyć $\Delta\mathbf{q}$ z drugiego równania, i wykorzystać to w pierwszym równaniu, co prowadzi do

$$\mathbf{K}^*\Delta\mathbf{u}_I = -\mathbf{r}^*, \qquad \text{gdzie} \qquad \mathbf{K}^* = \mathbf{K} - \mathbf{L}\mathbf{K}_{qq}^{-1}\mathbf{L}^T \quad \text{i} \quad \mathbf{r}^* = \mathbf{r}_u - \mathbf{L}\mathbf{K}_{qq}^{-1}\mathbf{r}_q. \tag{E.2}$$

Widać, że zredukowana (skondensowana) macierz $\mathbf{K}^*$ jest zdefiniowana jako uzupełnienie Schura dla $\mathbf{K}$, i zazwyczaj jest wyliczane w trzech krokach: (1) triangularyzacja (faktoryzacja) $\mathbf{K}_{qq}$, (2) jedno podstawienie wstecz dla każdej kolumny $\mathbf{L}^T$ aby uzyskać $\mathbf{K}_{qq}^{-1}\mathbf{L}^T$, oraz (3) mnożenie $\mathbf{L}$ przez $\mathbf{K}_{qq}^{-1}\mathbf{L}^T$ i odjęcie wyniku od $\mathbf{K}$. Aby przyspieszyć powyższy proces kondesacji, zaimplementowano i przetestowano alternatywną metodę - częściową faktoryzację.

Aby zdefiniować metodę częściowej faktoryzacji dla tego problemu, najpierw należy zamienić kolejność podmacierzy $\mathbf{u}_I$ i $\mathbf{q}$ tak że zamiast macierzy z równ. (E.1) otrzymano macierz $\mathbf{A}$ zdefiniowaną poniżej. Dekompozycja **LU** macierzy $\mathbf{A}$ przedstawia się następująco:

$$\mathbf{A} \doteq \begin{bmatrix} \mathbf{K}_{qq} & \mathbf{L}^T \\ \mathbf{L} & \mathbf{K} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ 0 & \mathbf{U}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11}\mathbf{U}_{11} & \mathbf{L}_{11}\mathbf{U}_{12} \\ \mathbf{L}_{21}\mathbf{U}_{11} & \mathbf{L}_{21}\mathbf{U}_{12} + \mathbf{L}_{22}\mathbf{U}_{22} \end{bmatrix}. \tag{E.3}$$

Tabela E.1: Przyspieszenia dla poszczególnych metod i powiązanych solverów w stosunku do czasu dla metody z równ. (E.3) i solwera z [91].

		Elementy powłokowe			Elementy „solid-shells”			Elementy 3D		
Solver	Metoda	EAS10	HW35	HW43	EAS10	HW29	HW47	EADG12	EAS30	HW60
DSYTRF	1RHS	0.26	0.48	0.68	0.30	0.56	0.60	0.29	0.45	0.75
MA64	1RHS	0.57	1.02	1.36	0.72	1.37	1.51	0.84	1.23	1.74
DSYTRF	mRHS	0.97	1.72	2.06	1.04	1.93	1.82	1.12	1.49	2.20
MA64	mRHS	0.62	1.04	1.37	0.77	1.39	1.82	0.90	1.26	1.75
DSYTRF	ownPF	**1.27**	**2.47**	**3.37**	**1.37**	**2.64**	**2.52**	**1.57**	**1.86**	**2.41**
MA64	PF	0.58	1.41	1.91	0.72	1.90	2.10	0.85	1.49	2.31
Gestość macierzy [%]		91.20	100.00	58.70	98.40	48.20	29.20	40.60	22.20	13.70
Czas [sek.]		6.77	49.96	129.08	10.46	75.19	108.19	13.34	53.88	192.54

Warto zauważyć, że $\mathbf{L}_{21}\mathbf{U}_{11} = \mathbf{L}$ oraz $\mathbf{L}_{11}\mathbf{U}_{12} = \mathbf{L}^T$, wtedy $\mathbf{L}_{21}\mathbf{U}_{12} = \mathbf{L}\mathbf{U}_{11}^{-1}\mathbf{L}_{11}^{-1}\mathbf{L}^T = \mathbf{L}\mathbf{K}_{qq}^{-1}\mathbf{L}^T$. Dlatego,

$$\mathbf{K} - \mathbf{L}_{21}\mathbf{U}_{12} = \mathbf{K} - \mathbf{L}\mathbf{K}_{qq}^{-1}\mathbf{L}^T = \mathbf{K}^*, \tag{E.4}$$

gdzie $\mathbf{K}^*$ to macierz zredukowana z równ. (E.2).

Testy zostały przeprowadzone na 1 rdzeniu maszyny wielordzeniowej (2 procesory Xeon X5650 2.66GHz z 6 rdzeniami każdy, system Linux). Wybrano tylko jeden rdzeń, z uwagi na to, że autor zaimplementował równoległą pętlę po elementach (patrz rozdz. 3), w której każdy rdzeń przetwarza inny element skończony.

Macierze sztywności, które zostały użyte w obliczeniach, uzyskano dla centralnych elementów „patch” testu [71]. Obliczenia były powtórzone milion razy dla każdej macierzy. Przyspieszenia zostały zamieszczone w tab. E.1, najlepsze rezultaty dla każdej macierzy zostały pogrubione.

Jako metodę referencyjną wybrano metodę wyrażoną równ. (E.2) oraz metodę LUDCMP z [91]. Ponadto przetestowano dwie metody bazujące na eliminacji Gaussa: DSYTRF z biblioteki LAPACK [65] oraz metodę MA64 z biblioteki HSL [45]. Do otrzymania $\mathbf{K}_{qq}^{-1}\mathbf{L}^T$, wykorzystano dwa sposoby wywołania procedur dla podstawiania wstecz: albo dla każdej kolumny z $\mathbf{L}^T$ osobno („1RHS”), albo dla wszystkich kolumn z $\mathbf{L}^T$ jednocześnie („mRHS”). Dodatkowo z użyciem „ownPF” oznaczono metodę z własnymi modyfikacjami kodu, do otrzymania częściowej faktoryzacji.

Przetestowano trzy typy elementów: czterowęzłowe elementy powłokowe, ośmiowęzłowe elementy „solid-shells”, ośmiowęzłowe elementy trójwymiarowe. Ich sformułowania są oznaczone następująco: HW - opiera się na funkcjonalne Hu-Washizu i został rozszerzony dla elementów powłokowych i „solid-shell” [115], EAS - opiera się na energii potencjalnej z ulepszonym założonym odkształceniem (ang. *Enhanced Assumed Strain*), oraz EADG - opiera się na energii potencjalnej z ulepszonym założonym gradientem przemieszczenia (ang. *Enhanced Assumed Displacement Gradient*). Liczba dodatkowych parametrów znajduje się za powyższymi akronimami.

Podsumowując, wykonane testy pokazują, że implementacja częściowej faktoryzacji, daje zysk czasowy prawie dla wszystkich typów elementów, a metoda wykorzystująca „DSYTRF” z „ownPF” daje najlepsze przyspieszenia.

F Dodatek F
FGMRES

FGMRES (ang. *Flexible Generalized Minimal Residual Method*) to iteracyjna metoda rozwiązywania niesymetrycznych układów równań liniowych. Metoda ta aproksymuje rozwiązanie za pomocą wektora z przestrzeni Krylova z minimalnym residuum. Do znalezienia tego wektora używa się iteracji Arnoldiego. Metoda FGMRES różni się od metody GMRES, tym że w każdej iteracji może być stosowany inny preconditioner, czyli macierz, za pomocą której zamienia się podstawowy problem w problem, który można łatwiej rozwiązać z punktu widzenia numerycznego. Aby przedstawić algorytm FGMRES, najpierw zostanie opisany algorytm GMRES na podstawie [94].

1. **Start:** Wybierz x_0 oraz wymiar m przestrzeni Krylova. Zdefiniuj macierz $\bar{H}_m$ rozmiaru $(m+1)\times m$ i zainicjalizuj tę macierz ze wszystkimi elementami $h_{i,j}$ równymi zero.

2. **Iteracja Arnoldiego:**

   (a) Oblicz $r_0 = b - Ax_0,\ \beta = \|r_0\|_2$ oraz $v_1 = r_0/\beta$

   (b) Dla $j = 1, ..., m$ wykonaj:

   - Oblicz $z_j := M^{-1}v_j$
   - Oblicz $w := z_j$
   - Dla $i = 1, ..., j$ wykonaj $y = \begin{cases} h_{i,j} := (w, v_i) \\ w := w - h_{i,j}v_i \end{cases}$
   - Oblicz $h_{j+1,i} = \|w\|_2$ oraz $v_{j+1} = w/h_{j+1,j}$

   (c) Zdefiniuj $V_m := [v_1, ..., v_m]$

3. **Utwórz przybliżone rozwiązanie:** Oblicz $x_m = x_0 + M^{-1}V_m y_m$, gdzie $y_m = \operatorname{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$ oraz $e_1 = [1, 0, ..., 0]^T$.

4. **Restart:** Jeśli warunek stopu został osiągnięty to kończ, w przeciwnym wypadku $x_0 \leftarrow x_m$ i idź do kroku 2.

Iteracja Arnoldiego konstruuje bazę ortogonalną do przestrzeni Krylova z preconditionerem:

$$Span\{r_0, AM^{-1}r_0, ..., (AM^{-1})^{m-1}r_0\} \tag{F.1}$$

za pomocą zmodyfikowanego procesu Grama-Schmidta, gdzie nowy wektor, który będzie zortogonalizowany, jest otrzymany z poprzedniego wektora w procesie. Ostatni krok powyższego algorytmu tworzy rozwiązanie, które jest liniową kombinacją wektorów z preconditionerem: $z_i = M^{-1}v_i,\ i = 1, ..., m$. Dzięki temu, że wszystkie wektory zostały otrzymane poprzez użycie tego samego preconditionera M^{-1}, nie ma potrzeby ich przechowywać. Trzeba tylko zastosować M^{-1} do liniowych kombinacji wektora v, tj. $V_m y_m$.

Jednak jeśli na każdym nowym kroku algorytmu zostanie wykorzystany nowy preconditioner, tzn. niech z_j będzie zdefiniowany następująco:

$$z_j = M_j^{-1} v_j \tag{F.2}$$

a następnie wektory te będą zachowywane, aby później można byłoby ich użyć w procesie aktualizacji x_m w kroku 3, to otrzyma się elastyczną (ang. *flexible*) modyfikację algorytmu GMRES, w skrócie FGMRES:

1. **Start:** Wybierz x_0 oraz wymiar m przestrzeni Krylova. Zdefiniuj macierz $\bar{H}_m$ rozmiaru $(m+1) \times m$ i zainicjalizuj tę macierz ze wszystkimi elementami $h_{i,j}$ równymi zero.

2. **Iteracja Arnoldiego:**

   (a) Oblicz $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$ oraz $v_1 = r_0/\beta$

   (b) Dla $j = 1, ..., m$ wykonaj:

   - Oblicz $z_j := M_j^{-1} v_j$
   - Oblicz $w := z_j$
   - Dla $i = 1, ..., j$ wykonaj $y = \begin{cases} h_{i,j} := (w, v_i) \\ w := w - h_{i,j} v_i \end{cases}$
   - Oblicz $h_{j+1,i} = \|w\|_2$ oraz $v_{j+1} = w/h_{j+1,j}$

   (c) Zdefiniuj $Z_m := [z_1, ..., z_m]$

3. **Utwórz przybliżone rozwiązanie:** Oblicz $x_m = x_0 + Z_m y_m$, gdzie $y_m = \text{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$ oraz $e_1 = [1, 0, ..., 0]^T$.

4. **Restart:** Jeśli warunek stopu został osiągnięty to kończ, w przeciwnym wypadku $x_0 \leftarrow x_m$ i idź do kroku 2.

G Dodatek G PARFEAP i Warp3D

Dla dużych zadań metody elementów skończonych (rzędu kilku, kilkunastu milionów równań) zwykłe stacje robocze stają się mało efektywne biorąc pod uwagę czas obliczeń, a także często na takiej stacji jest za mało pamięci operacyjnej, aby uruchomić zadanie. Z pomocą przychodzą tutaj programy wykorzystujące techniki równoległego programowania, takie jak zrównoleglanie po wątkach (np. OpenMP), czy zrównoleglanie za pomocą procesów z przesyłaniem komunikatów (z wykorzystanie MPI). Ciekawą grupą programów są programy wykorzystujące obie techniki - jest to podejście hybrydowe. Programem metody elementów skończonych, który wykorzystuje to podejście jest Warp3D. Głównym celem tego dodatku jest zaprezentowanie wyników, jakie można uzyskać za pomocą tego programu, a także porównanie go z programem, który wykorzystuje tylko podejście zrównoleglania po procesach - równoległa wersja programu FEAP - parFEAP.

Sprzęt i oprogramowanie wykorzystane do testów Wszystkie testy wykonano na tych samych węzłach obliczeniowych klastra obliczeniowego GRAFEN [29] infrastruktury informatycznej Biocentrum Ochota [7]. Komputer Grafen to:

1. **Klaster**
   - 24 identyczne węzły obliczeniowe,
   - 2 procesory 6-rdzeniowe Xeon X5650, 2.66 GHz każdy rdzeń,
   - 24GB pamięci RAM DDR3 1333MHz,
   - Połączenia międzywęzłowe: InfiniBand 40Gb/s.
2. **VSMP - Virtual Symmetric Multiprocessing**
   - użytkownik widzi pojedynczą maszynę,
   - 72 rdzenie (6 x 2 procesory 6-rdzeniowe Xeon X5650, 2.66 GHz każdy rdzeń),
   - 576 GB pamięci RAM.

FEAP - wersja DM-MIMD Program FEAP posiada wersję sparalelizowaną, która nazywa się parFEAP. Wersja ta wykorzystuje bibliotekę PETSc [90], która jest sparalelizowana wykorzystując interfejs MPI. Wersja sparalelizowana także po wątkach jest na razie w stanie developerskim. Do wykonania testów użyto wersji PETSc 3.3. Przed uruchomieniem zadania trzeba także wykonać dekompozycję obszaru. Została ona wykonana także w programie FEAP, który do tego celu wykorzystuje program METIS [74]. Algorytmy wykorzystywane przez METIS zostały opisane w rozdz. 4.2.4 oraz w dodatku B. Dane podawane dla tego przypadku nie zawierają informacji o procesie budowania zadania, tylko dane o procesie samego rozwiązania zadania. W przypadku testów liniowych wykorzystano do rozwiązania układu równań metodę gradientów sprzężonych zaimplementowaną w solwerze Hypre [44] oraz użyto preconditionera BoomerAMG, w wersji 2.9.0b.

Warp3D - wersja DM-MIMD Jak już wspomniano wcześniej program Warp3D kompiluje się do dwóch wersji. W tym przypadku testowano wersję hybrydową, tzn. korzystającą jednocześnie z MPI do zrównoleglenia po węzłach obliczeniowych i z OpenMP do zrównoleglenia po wątkach na danym węźle. Do rozwiązania układu równań w tym przypadku również wykorzystano metodę gradientów sprzężonych z solwera Hypre [44] z precoditionerem BoomerAMG, w wersji 2.9.0b. Razem z Warp3D jest dostarczony program patwarp, który potrafi wczytać siatkę w formacie PATRAN Neutral File, a później za pomocą programu METIS, wykonać dekompozycję tej siatki. Niestety program działał niestabilnie dla większej liczby węzłów, dlatego zdecydowano się na wykorzystanie trzech węzłów oraz czterech wątków, co w sumie daje tyle samo procesów, co w przypadku parFEAP.

Testowe zadanie Jako test wybrano zagadnienie liniowo sprężyste przedstawione w rozdz. 3.2.1, głównie z powodu prostoty i aby sam rodzaj zadania nie wpływał na uzyskane wyniki porównawcze.

1. **FEAP** Element wykorzystywany do obliczeń to 8-węzłowy element sześcienny. Typ materiału: SOLID, ELASTIC ISOTROPIC. Trzeba zaznaczyć, że proces budowania siatki, także jest wykonywany w trakcie przebiegu programu FEAP za pomocą procedury BLOCK.

2. **Warp3D** Typ materiału to 'deformation' [110, s. 3.4-1], któremu ustalono granicę plastyczności na 10^{20}, aby modelować liniową sprężystość. Użyty element to 'l3disop' [110, s. 3.1-1], który odpowiada elementowi wykorzystywanemu w FEAPie. Wyłączono także sformułowanie B, które domyślnie jest uruchomione dla tego elementu. Warp3D nie ma domyślnie stworzonej procedury podobnej do BLOCK z programu FEAP. Dlatego zaimplementowano program, który tworzy odpowiednią siatkę w formacie PATRAN Neutral File, który to format może być wczytany przez dodatkowy program dołączony do Warp3D - patwarp. W tym przypadku proces budowania siatki nie jest włączony w cały proces rozwiązania zadania.

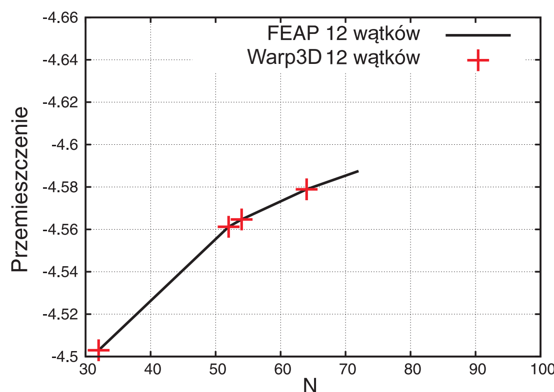
Przeprowadzone testy skupiały się na porównaniu wydajności dwóch programów FEAP oraz Warp3D. Wydajność rozumiana jest tutaj, jako czas potrzebny na zakończenie zadania oraz wykorzystaną pamięć operacyjną dla danego zadania. Testy podzielono na dwie grupy - wykorzystujące tylko wątki oraz wykorzystujące system komunikatów MPI. Poniżej znajduje się opis przeprowadzonych testów dla każdego z dwóch testowanych programów dla każdej z wyżej wymienionych grup.

Przemieszczenie Przemieszczenia sprawdzano w punkcie C (tak jak w rozdz. 3.2.1) niezależnie od wyboru metody rozwiązania otrzymywano dokładnie te same wyniki. W przypadku programów tylko z wątkami program FEAP policzył zadanie składające się z ponad 1 milionem niewiadomych (1 151 064). Program Warp3D zatrzymał się na liczbie nieco ponad 800 tysięcy niewiadomych (811 200). Warto też zauważyć, że program Warp3D ma ograniczoną na sztywno liczbę elementów, która wynosi dwa miliony. Dlatego maksymalne N, dla którego wykonano obliczenia w przypadku programu Warp3D

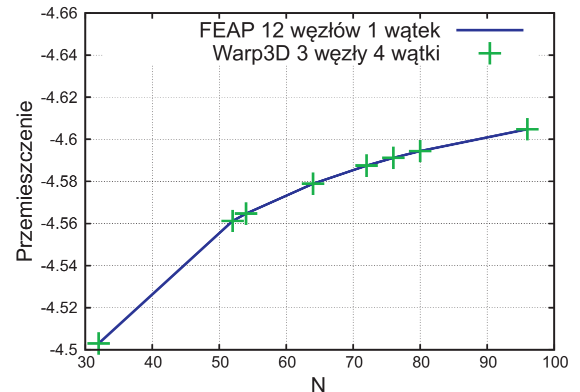
wynosi 124 (1 906 624 elementów), co daje prawie 6 milionów niewiadomych (5 859 375). Tabela G.1 zawiera informację ile równań i ile elementów odpowiada danemu N.

Tabela G.1: Liczba równań i liczba elementów dla danego N.

N	Równań	Elementów
32	104 544	32 768
52	438 204	140 608
54	490 050	157 464
64	811 200	262 144
72	1 151 064	373 248
76	1 351 812	438 976
80	1 574 640	512 000
96	2 709 792	884 736
124	5 859 375	1 906 624
256	51 000 000	16 777 216
300	81 700 000	27 000 000
322	101 000 000	33 386 248
400	190 000 000	64 000 000



(a)

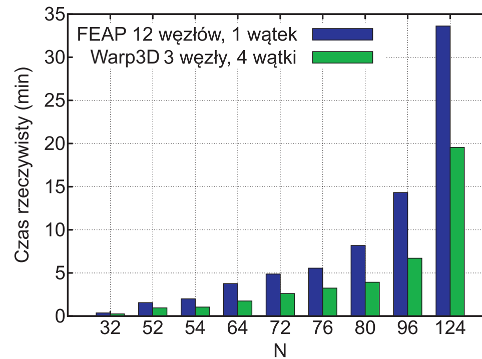


(b)

Rysunek G.1: Przemieszczenie w punkcie C dla wersji SM-MIMD (a) oraz wersji DM-MIMD (b).

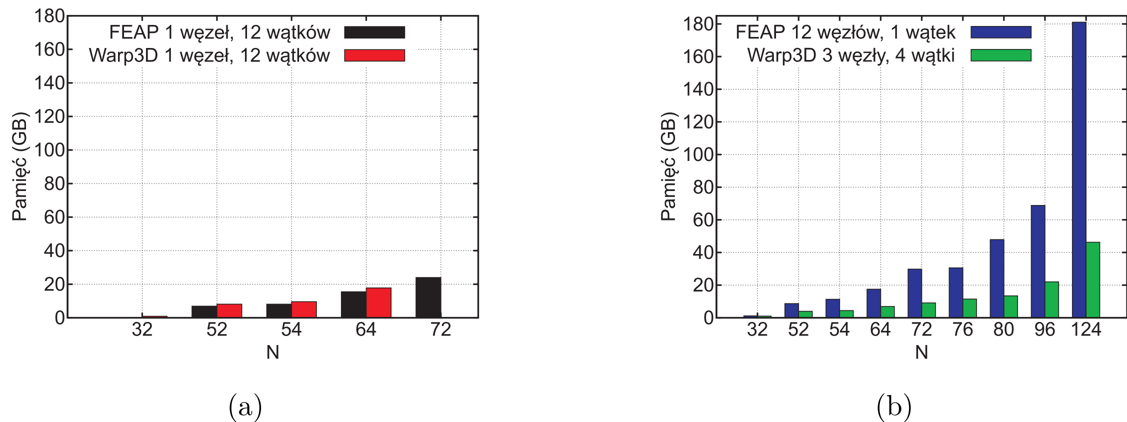
Porównanie czasów dla wersji DM-MIMD W tym przypadku porównano tylko rzeczywiste czasy wykonania. Na rys. G.2 widać, że Warp3D jest szybszy, dla $N = 124$ ok. 1.8 razy. Niestety Warp3D ma ograniczenie do 2 milionów węzłów i największym zadaniem jakie udało się przeliczyć było właśnie $N = 124$, czyli ok. 5.8 miliona niewiadomych. Z czasów tych widać również, że komunikacja MPI wymaga więcej czasu, niż komunikacja w obrębie jednego węzła obliczeniowego ze wspólną pamięcią.

Porównanie użycia pamięci Rysunek G.3a potwierdza wnioski z rozdz. 3.2, Warp3D z PARDISO zużywa więcej pamięci niż FEAP z PARDISO. Główną tego przyczyną jest fakt, że Warp3D alokuje statycznie potrzebne tablice do rozwiązania zadania.



Rysunek G.2: Czasy rzeczywiste wykonywania obliczeń.

Na rys. G.3b widać, że Warp3D z Hypre używa prawie 3 razy mniej pamięci niż FEAP z Hypre. FEAP do paralelizacji wykorzystuje bibliotekę PETSc, a Warp3D sam obsługuje komunikację między poszczególnymi procesami.



Rysunek G.3: Porównanie zapotrzebowania na pamięć operacyjną przez oba programy dla wersji SM-MIMD (a) oraz wersji DM-MIMD (b).

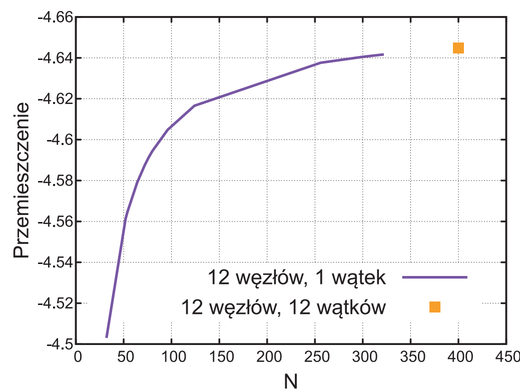
Największy przeliczony przykład Podjęto również próbę przeliczenia zadań rzędu 200 milionów niewiadomych, aby sprawdzić, czy parFEAP jest w stanie takie zadanie przetworzyć. Największe przeliczone zadanie było wielkości $N = 400$, co stanowi 64 miliony węzłów MES, a to oznacza ok. 190 milionów równań.

Dekompozycja zadania została zrobiona na węźle VSMP, ze względu na architekturę parFEAPa, który alokuje wiele tablic o rozmiarze rzędu liczby równań, które wcale nie są mu potrzebne podczas wykonywania dekompozycji. Czas trwania dekompozycji zadania to ok. 114 minut.

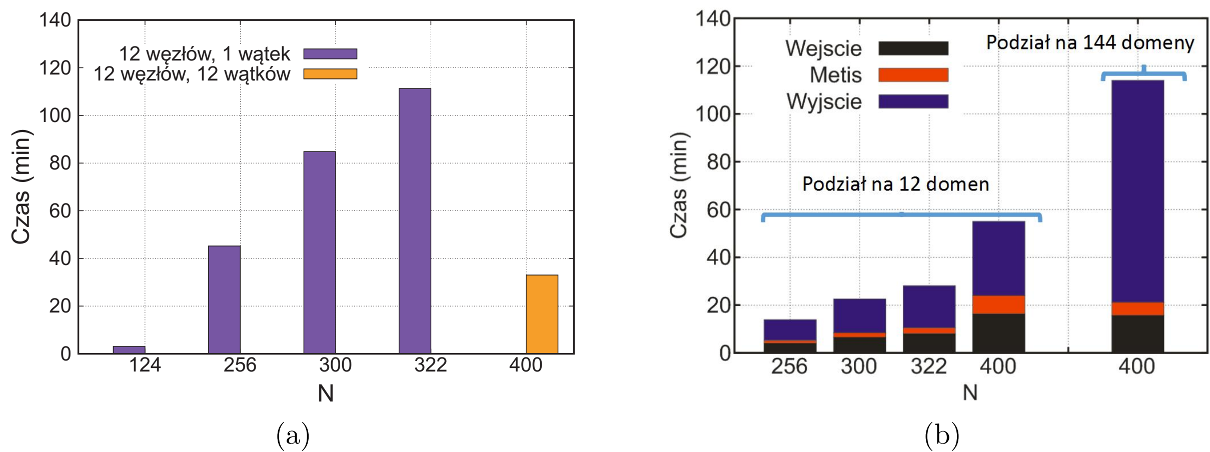
Samo zadanie zostało rozwiązane na klastrze przy wykorzystaniu 12 węzłów, a na każdym węźle utworzono 12 procesów MPI, co oznacza w sumie 144 procesy MPI. Rozwiązywanie trwało 33 min, a czas procesora to ok. 6h. Wykorzystano w sumie 193.4 GB z 576 GB dostępnych pamięci operacyjnej. W tym przypadku korzystano ze standardowego solwera CG z biblioteki PETSc i preconditionera Jacobi także z tej biblioteki.

Dla zadań o $N < 400$ korzystano z jednego rdzenia na każdym węźle obliczeniowym. Rysunek G.4 przedstawia przemieszczenia w punkcie C dla największych przeliczonych zadań. Dla $N \leqslant 124$ uzyskano przemieszczenia identyczne z otrzymanymi w poprzednich testach. Na rys. G.5a można zauważyć, że mimo komunikacji poprzez MPI między wątkami, czas wykonania dla zadania dla $N = 400$ był mniejszy ponad 2 razy.

Rysunek G.5b przedstawia czasy wykonania dekompozycji zadania na węźle VSMP. Wejście jest tutaj rozumiane jako wczytanie pliku źródłowego i wygenerowanie węzłów i elementów. Wyjście to czas zapisu zbiorów wyjściowych, czyli wejściowych dla zadania właściwego. Widać na rys. G.5b, że najdłuższy jest zapis zbiorów wyjściowych, użycie dysków SSD nie poprawiło rezultatów. Warto zwrócić uwagę na fakt, że podział na 144 domeny przez METIS trwał krócej niż podział na 12 domen. Wynika to z faktu, że w przypadku podziału na 144 domen liczba kroków w etapie zmniejszania jest mniejsza, ponieważ szybciej się uzyskuje graf, który można podzielić. Mniejsza liczba w etapie zmniejszania, to również mniejsza liczba kroków w etapie zwiększania, co w konsekwencji daje nam mniejszy czas wykonania.



Rysunek G.4: Przemieszczenia w punkcie C dla największych zadań.



(a) (b)

Rysunek G.5: Czasy wykonania obliczeń (a) oraz dekompozycji (b).

H Dodatek H
Procedury opakowujące bibliotekę PAPI

W ramach pracy stworzono zestaw procedur opakowujących mających na celu ułatwienie korzystania z biblioteki PAPI [86]. Procedury te wykorzystano w rozdz. 2.3.4, rozdz. 3.4.3 oraz w rozdz. 4.2.5 do stworzenia modelu „Roofline". Dzięki tym procedurom z punktu widzenia użytkownika końcowego wystarczy zainicjować bibliotekę procedurą wrap_PAPI_INIT, następnie określić, którą z dwóch metryk biblioteka ma zmierzyć (wydajność, czy użycie pamięci), potem użytkownik uruchamia rozpoczęcie mierzenia metryki procedurą wrap_PAPI_START, a gdy obliczenia zostaną zakończone, użytkownik wywołuje procedurę wrap_PAPI_STOP. Zostało to przedstawione w Kodzie H.1.

Kod H.1: Kolejność wywołań procedur opakowujących PAPI

```
1  program helloPAPI
2        type(papiControl) :: control
3        integer :: operation
4  ! Init PAPI
5        call wrap_PAPI_INIT(control)
6        control%metric = operation !1 - FLOP; 2-Bandwidth
7  !Start PAPI
8        call wrap_PAPI_START(control)
9  !Do calculations
10       call do_flops()
11 !Stop PAPI
12       call wrap_PAPI_STOP(control)
13 !Print results
14        if(control%metric.eq.1) then
15         write (*,*) "gflops", control%flops
16        elseif(control\%metric.eq.2) then
17         write (*,*) "bandwidth", control%bandwidth
18        endif
19 end
```

W Kodzie H.2 został zaimplementowany typ kontrolujący działanie biblioteki PAPI. Typ ten zawiera również dane wyjściowe, na temat badanych metryk.

Kod H.2: Moduł z typem kontrolującym PAPI

```
1        module papiTypes
2              type papiControl
3               integer,dimension(:), allocatable :: eventSet
4               integer*8 :: uso, usn
5               integer :: threads
6               double precision :: flops, bandwidth, time
7               integer :: metric ! 1 for flops, 2 for bandwidth
8               integer*8 :: flop, bytes
9              end type papiControl
```

```
10       end module papiTypes
```

W Kodzie H.3 został zaimplementowana procedura obsługująca inicjowanie biblioteki PAPI.

Kod H.3: Procedura inicjująca PAPI

```
1         use papiTypes
2         use omp_lib
3 
4         implicit none
5         include "f77papi.h"
6 
7         type(papiControl) :: control
8 
9         integer :: retval
10 
11 !$omp parallel
12        control%threads = omp_get_num_threads()
13 !$omp end parallel
14 
15        allocate(control%eventSet(control%threads))
16 
17        retval = PAPI_VER_CURRENT
18        call PAPIf_library_init(retval)
19        if ( retval.NE.PAPI_VER_CURRENT) then
20         write(*,*) 'error␣', retval
21        end if
22 
23        call PAPIF_thread_init(omp_get_thread_num, retval)
24        if ( retval.NE.PAPI_OK) then
25          write(*,*) 'error␣thread'
26        end if
```

W Kodzie H.4 został zaimplementowana procedura obsługująca rozpoczęcie mierzenia metryk za pomocą bibilioteki PAPI. W zależności, która metrykę użytkownik chce zmierzyć (wydajność, czy użycie pamięci), biblioteka musi rozpocząć działanie w inny sposób.

Kod H.4: Procedura rozpoczynająca mierzenie za pomocą PAPI

```
1        subroutine wrap_PAPI_START(control)
2         use papiTypes
3         use omp_lib
4 
5         implicit none
6         include "f77papi.h"
7 
```

```
8        type(papiControl) :: control
9
10       integer :: retval, eventsCounter,threadId,i
11       integer :: eventCode
12       integer,dimension(:), allocatable :: events
13       CHARACTER(len=128) :: eventNAME
14
15
16       if(control%metric.eq.1) then
17        call wrap_PAPI_START_FP(control)
18       elseif(control%metric.eq.2) then
19        call wrap_PAPI_START_UNC(control)
20       endif
21
22      call PAPIf_get_real_usec(control%uso)
23
24      end subroutine wrap_PAPI_START
```

W Kodzie H.5 został zaimplementowana procedura obsługująca rozpoczęcie mierzenia wydajności. Wydajność jest mierzona za pomocą zdarzenia predefiniowanego „PAPI_DP_OPS”, które dla procesora bazującego na architekturze Nehalem (np. Xeon 5670), jest wyznaczone za pomocą zdarzeń: FP_COMP_OPS_EXE:SSE_DOUBLE_PRECISION oraz FP_COMP_OPS_EXE:SSE_FP_PACKED. Zdarzenia te są liczone dla każdego wątku osobno, dla tego trzeba uruchomić mierzenie dla każdego wątku z osobna.

Kod H.5: Procedura rozpoczynająca mierzenie wydajności

```
1       subroutine wrap_PAPI_START_FP(control)
2        use papiTypes
3        use omp_lib
4
5        implicit none
6        include "f77papi.h"
7
8        type(papiControl) :: control
9        integer :: retval, eventsCounter,threadId,i
10       integer :: eventCode
11       integer,dimension(:), allocatable :: events
12       CHARACTER(len=128) :: eventNAME
13
14       eventsCounter = 1
15
16 !$omp parallel private( retval, i, events,  threadId)
17      call PAPIF_register_thread(retval)
18      if ( retval.NE.PAPI_OK) then
19        write(*,*) 'error thread', retval
```

```
20        end if
21
22        threadId = omp_get_thread_num() + 1
23        allocate(events(eventsCounter))
24        events(1) = PAPI_DP_OPS
25
26        control%eventSet(threadId) = PAPI_NULL
27
28        call PAPIf_create_eventset(control%eventSet(threadId),
29       1 retval)
30        if ( retval.NE.PAPI_OK) then
31          write(*,*) 'error␣es', retval
32        end if
33
34        do i=1, eventsCounter
35         call PAPIf_add_event(control%eventSet(threadId),
36       1 events(i), retval)
37         if ( retval .NE. PAPI_OK ) then
38          write(*,*) 'error␣add␣event␣', i, retval
39         end if
40        enddo
41
42        call PAPIf_start(control%eventSet(threadId), retval)
43
44        if ( retval .NE. PAPI_OK ) then
45          write(*,*) 'error␣start␣', retval
46        end if
47
48  !$omp end parallel
49
50
51        end subroutine wrap_PAPI_START_FP
```

W Kodzie H.6 został zaimplementowana procedura obsługująca rozpoczęcie mierzenia użycia pamięci. Jest ono mierzone za pomocą zdarzeń „uncore”: „wsm_unc::UNC_QMC_NORMAL_READS:ANY” oraz „wsm_unc::UNC_QMC_WRITES:FULL_ANY”. Zdarzenia te są liczone dla każdego procesora osobno, dla tego trzeba uruchomić mierzenie dla każdego procesora z osobna.

Kod H.6: Procedura rozpoczynająca mierzenie użycia pamięci

```
1         subroutine wrap_PAPI_START_UNC(control)
2          use papiTypes
3          use omp_lib
4
5          implicit none
```

```
6         include "f77papi.h"
7
8         type(papiControl) :: control
9
10        integer :: retval, eventsCounter,socket,i
11        integer :: eventCode,eventCode2,socketNum
12        integer,dimension(:), allocatable :: events
13        CHARACTER(len=128) :: eventNAME
14
15 !for Nehalem
16       eventNAME = "wsm_unc::UNC_QMC_NORMAL_READS:ANY"
17
18       call PAPIF_event_name_to_code(eventNAME, eventCode,
19      1 retval)
20       if ( retval.NE.PAPI_OK) then
21         write(*,*) 'error␣eventCode␣',retval
22       end if
23 !for Nehalem
24       eventNAME = "wsm_unc::UNC_QMC_WRITES:FULL_ANY"
25
26       call PAPIF_event_name_to_code(eventNAME,eventCode2,
27      1 retval)
28       if ( retval.NE.PAPI_OK) then
29         write(*,*) 'error␣eventCode␣',retval
30       end if
31
32 !Nehalem
33       socketNum = 2
34
35       do socket = 1, socketNum
36        eventsCounter = 2
37        allocate(events(eventsCounter))
38        events(1) = eventCode
39        events(2) = eventCode2
40
41        control%eventSet(socket) = PAPI_NULL
42
43        call PAPIf_create_eventset(control%eventSet(socket),
44      1 retval)
45
46        call papif_prepareuncore(control%eventSet(socket),
47      1  (socket-1))
48
49        do i=1, eventsCounter
50         call PAPIf_add_event(control%eventSet(socket),
```

```
51      1    events(i), retval)
52         if ( retval .NE. PAPI_OK ) then
53          write(*,*) 'error␣add␣event␣', i, retval
54         end if
55        enddo
56
57        deallocate(events)
58       enddo
59
60       do socket = 1, socketNum
61        call PAPIf_start(control%eventSet(socket), retval)
62        if ( retval .NE. PAPI_OK ) then
63         write(*,*) 'error␣start␣', retval
64        end if
65       enddo
66
67       end subroutine wrap_PAPI_START_UNC
```

W Kodzie H.7 został zaimplementowana procedura kończąca mierzenie, podobnie jak w Kodzie H.4 inaczej trzeba obsłużyć mierzenie operacji zmiennoprzecinkowych i inaczej użycia pamięci.

Kod H.7: Procedura kończąca mierzenie

```
1
2        subroutine wrap_PAPI_STOP(control)
3         use papiTypes
4         use omp_lib
5
6         implicit none
7         include "f77papi.h"
8
9         type(papiControl) :: control
10
11        integer :: retval, eventsCounter,threadId,i
12        integer :: threads, sockets
13        integer*8 :: ops(10)
14        integer*8 :: values(10), cycles
15        integer*8 :: ldops,srops
16
17       call PAPIf_get_real_usec(control%usn)
18
19       control%time = DFLOAT(control%usn-control%uso) / 1.0d6
20       ops = 0
21
22        if(control%metric.eq.1) then
23         call wrap_PAPI_STOP_FP(control)
```

```
24          elseif(control%metric.eq.2) then
25           call wrap_PAPI_STOP_UNC(control)
26          endif
27
28         end subroutine wrap_PAPI_STOP
```

W Kodzie H.8 został zaimplementowana procedura kończąca mierzenie operacji zmiennoprzecinkowych, podobnie jak w Kodzie H.5 trzeba wykonać zakończenie obliczeń dla każdego wątku z osobna.

Kod H.8: Procedura kończąca mierzenie operacji zmiennoprzecinkowych

```
1
2          subroutine wrap_PAPI_STOP_FP(control)
3           use papiTypes
4           use omp_lib
5
6           implicit none
7           include "f77papi.h"
8
9           type(papiControl) :: control
10
11          integer :: retval, eventsCounter,threadId,i
12          integer :: threads, sockets
13          integer*8 :: ops(10)
14          integer*8 :: values(10), cycles
15          integer*8 :: ldops,srops
16
17         ops = 0
18
19
20   !$omp parallel private(retval, i, values, threadId )
21         threadId = omp_get_thread_num() + 1
22
23         call PAPIf_stop(control%eventSet(threadId), values(1), retval)
24
25         if ( retval .NE. PAPI_OK ) then
26           write(*,*) 'error stop ', retval
27         end if
28
29   !$omp ATOMIC
30           ops(1) = ops(1) + values(1)
31
32   !$omp ATOMIC
33           ops(2) = ops(2) + values(2)
34
35           threads = omp_get_num_threads()
```

```
36
37 !$omp end parallel
38
39         control%time = DFLOAT(control%usn-control%uso) / 1.0d6
40
41         if(control%metric.eq.1) then
42
43          control%flops = ops(1)/(control%time)
44          control%flops = control%flops / 1.0d9
45         endif
46
47
48       end subroutine wrap_PAPI_STOP_FP
```

W Kodzie H.9 został zaimplementowana procedura kończąca mierzenie użycia pamięci, podobnie jak w Kodzie H.6 trzeba wykonać zakończenie obliczeń dla każdego procesora z osobna.

Kod H.9: Procedura kończąca mierzenie użycia pamięci

```
1        subroutine wrap_PAPI_STOP_UNC(control)
2         use papiTypes
3         use omp_lib
4
5         implicit none
6         include "f77papi.h"
7
8         type(papiControl) :: control
9
10        integer :: retval, eventsCounter,threadId,i
11        integer :: threads, sockets
12        integer*8 :: ops(10)
13        integer*8 :: values(10), cycles
14        integer*8 :: ldops,srops,socketNum
15
16       ops = 0
17
18 ! for Nehalem
19       socketNum = 2
20
21       do threadId = 1, socketNum
22        call PAPIf_stop(control%eventSet(threadId), values(1),
23      1  retval)
24
25        if ( retval .NE. PAPI_OK ) then
26          write(*,*) 'error␣stop␣', retval, 'id␣',threadId
27        end if
```

```
28
29          ops(1) = ops(1) + values(1)
30          ops(2) = ops(2) + values(2)
31
32         enddo
33
34         ldops = ops(1)*64
35         srops = ops(2)*64
36
37
38         if(control%metric.eq.2) then
39
40          control%bandwidth = (ldops+srops)/(control%time)
41          control%bandwidth = control%bandwidth / 1.0d9
42         endif
43
44         end subroutine wrap_PAPI_STOP_UNC
```

I Dodatek I
Blokowa faktoryzacja macierzy symetrycznej

Blokowa faktoryzacja macierzy symetrycznej została wykorzystana w rozdz. 5.4.3, aby w trakcie częściowej faktoryzacji uzyskać również precoditionery. Faktoryzacja **LU** dla macierzy symetrycznej prowadzi do postaci:

$$\bar{\mathbf{A}} = \bar{\mathbf{L}}\bar{\mathbf{L}}^T. \tag{I.1}$$

Wtedy układ równań $\bar{\mathbf{A}}\mathbf{x} = \mathbf{b}$ może być rozwiązany w dwóch krokach, podstawianie w przód:

$$\bar{\mathbf{L}}\mathbf{y} = \mathbf{b} \tag{I.2}$$

i podstawianie w tył:

$$\bar{\mathbf{L}}^T\mathbf{x} = \mathbf{y}. \tag{I.3}$$

Ale jeśli dekompozycja **LU** zostanie wykonana na blokowej symetrycznej macierzy to otrzyma się:

$$\bar{\mathbf{A}} = \begin{bmatrix} \mathbf{A} & \mathbf{D}^T \\ \mathbf{D} & \mathbf{C} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^T & \mathbf{L}_{21}^T \\ \mathbf{0} & \mathbf{L}_{22}^T \end{bmatrix} \tag{I.4}$$

gdzie $\mathbf{L}_{ij}$ to blokowe części faktoryzacji dolnej części trójkątnej macierzy. Aby rozwiązać taki system również można skorzystać, z podstawień w przód/w tył. Jednak czasami potrzebne jest rozwiązanie, nie dla całego układu $\bar{\mathbf{A}}$, a tylko do układów $\mathbf{A}\mathbf{x} = \mathbf{b}$ lub $\mathbf{S}\mathbf{x} = \mathbf{b}$, gdzie $\mathbf{S}$ to uzupełnienie Schura macierzy $\bar{\mathbf{A}}$, tzn. $\mathbf{S} = \mathbf{C} - \mathbf{D}\mathbf{A}^{-1}\mathbf{D}^T$. Wtedy można wykonać podstawienie w przód / w tył w następujący sposób:

$$\begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{y}' \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix} \tag{I.5}$$

$$\begin{bmatrix} \mathbf{L}_{11}^T & \mathbf{L}_{21}^T \\ \mathbf{0} & \mathbf{L}_{22}^T \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{x}' \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} \tag{I.6}$$

co pozwoli na uzyskanie rozwiązania dla układu $\mathbf{A}\mathbf{x} = \mathbf{b}$. Jest tak ponieważ z pierwszego równania w równ. (I.5), jest $\mathbf{y} = \mathbf{L}_{11}^{-1}\mathbf{b}$ a z pierwszego równania w równ. (I.6) jest $\mathbf{x} = (\mathbf{L}_{11}^T)^{-1}\mathbf{y} - (\mathbf{L}_{11}^T)^{-1}\mathbf{L}_{21}^T\mathbf{x}'$, ale z drugiego równania w równ. (I.6), wiadomo, że $\mathbf{x}' = 0$, więc $\mathbf{x} = (\mathbf{L}_{11}^T)^{-1}\mathbf{L}_{11}^{-1}\mathbf{b}$ oraz $\mathbf{x}$ jest rozwiązaniem układu $\mathbf{A}\mathbf{x} = \mathbf{b}$.

Gdy zostaną wykonane następujące operacje:

$$\begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{y}' \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{b} \end{bmatrix} \tag{I.7}$$

$$\begin{bmatrix} \mathbf{L}_{11}^T & \mathbf{L}_{21}^T \\ \mathbf{0} & \mathbf{L}_{22}^T \end{bmatrix} \begin{bmatrix} \mathbf{x}' \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{y} \end{bmatrix} \tag{I.8}$$

można otrzymać rozwiązanie dla układu $\mathbf{S}\mathbf{x} = \mathbf{b}$. Jest tak, ponieważ z drugiego równania w równ. (I.8), jest $\mathbf{x} = (\mathbf{L}_{22}^{-1})^T\mathbf{y}$ a z drugiego równania w równ. (I.7) jest $\mathbf{y} = \mathbf{L}_{22}^{-1}\mathbf{b} - \mathbf{L}_{22}^{-1}\mathbf{L}_{21}\mathbf{y}'$, ale z pierwszego równania w równ. (I.7), wiadomo, że $\mathbf{y}' = 0$, więc $\mathbf{x} = (\mathbf{L}_{22}^{-1})^T\mathbf{L}_{22}^{-1}\mathbf{b}$ oraz $\mathbf{x}$ jest rozwiązaniem układu $\mathbf{S}\mathbf{x} = \mathbf{b}$.