Instytut Podstawowych Problemów Techniki
Polska Akademia Nauk

Zarządzanie rozproszoną siatką obliczeniową w równoległych symulacjach adaptacyjną metodą elementów skończonych

Kazimierz Michalik

Promotor: dr hab. inż. Krzysztof Banaś, prof. AGH

Katedra Modelowania i Informatyki Stosowanej
Wydział Inżynierii Metali i Informatyki Przemysłowej
Akademia Górniczo-Hutnicza im. St. Staszica w Krakowie

Kraków, 2017

Institute of Fundamental Technological Research
Polish Academy of Sciences

Management of distributed meshes in parallel adaptive finite element simulations

Kazimierz Michalik

Supervisor: Krzysztof Banaś, Ph.D. eng., prof. AGH

Department of Applied Computer Science and Modelling
Faculty of Metals Engineering and Industrial Computer Science
AGH University of Science and Technology in Cracow

Cracow, 2017

Streszczenie

Niniejsza praca podejmuje tematykę z zakresu nauk obliczeniowych, w szczególności zadań i roli, jaką zajmuje rozproszona adaptacyjna siatka obliczeniowa w symulacjach z wykorzystaniem równoległych architektur komputerowych.

Zastosowanie zaawansowanych metod siatkowych, w tym adaptacyjnej metody elementów skończonych, jest niemożliwe bez teoretycznego opracowania i praktycznego zrealizowania pewnego modelu organizacji i funkcjonalności siatki obliczeniowej. W zależności od docelowej architektury przetwarzania komputerowego i dostępnych poziomów równoległości model ten jest różny. Mimo wielu publikacji i praktycznych zastosowań pojęcie *zarządzania siatką obliczeniową* nie zostało dotychczas spójnie zdefiniowane, mimo iż jest używane w kontekście prowadzonych badań naukowych.

Praca składa się z dwóch głównych części. Pierwsza z nich obejmuje zagadnienia wprowadzające oraz przedstawia modele ukazujące, czym różnią się od siebie wykorzystywane siatki obliczeniowe. Zaprezentowane są różne modele reprezentacji siatek obliczeniowych, uwzględniając m.in. strukturę powiązań obiektów w adaptacyjnej siatce obliczeniowej i ich wzajemnych relacji. Na bazie tych modeli został przedstawiony opis funkcjonalności oferowanych przez siatki obliczeniowe, niezależnie od ich struktur danych, w postaci tzw. *operatorów siatki*. Przy ich pomocy możliwe stało się zdefiniowanie różnych rodzajów siatek, ze względu na zbiór operatorów, które realizują, a w konsekwencji zdefiniowanie w sposób ścisły czym jest *zarządzanie siatką obliczeniową*.

Druga część dotyczy zagadnień praktycznego odwzorowania istotnych operatorów rozproszonej siatki adaptacyjnej na trzy aktualnie kluczowe poziomy równoległego przetwarzania komputerowego: wektoryzację, wielowątkowość z pamięcią wspólną i przetwarzanie wieloprocesowe w modelu pamięci rozproszonej. Dla każdego z nich zostały wybrane istotne operatory, w stosunku do których opracowano nowe metody ich realizacji, ze szczególnym naciskiem na efektywne wykorzystanie zasobów sprzętowych. Stworzono między innymi nowe algorytmy dotyczące skompresowanego zapisu dyskretnej przestrzeni punktów oraz rozproszonego przydzielania globalnie jednoznacznych identyfikatorów. Mogą one mieć szersze zastosowanie w naukach obliczeniowych, wykraczające poza zagadnienia związane z zarządzaniem siatkami obliczeniowymi.

Wszystkie opracowane rozwiązania zostały zaprezentowane zarówno teoretycznie, jak i praktycznie uwzględniając wyniki z ich wykorzystania w przykładach akademickich oraz z praktyki przemysłowej.

Abstract

The subject of the dissertation, belonging to the field of computational sciences, is the role of the distributed adaptive mesh in simulations using parllel computer architectures and the associated particular tasks.

Application of advanced mesh based methods, including the adaptive finite element method, is impossible without theoretical elaboration and practical realization of a model for organization and operation of computational meshes. Depending on the target computing architecture, and the available levels of parallelism, such a model can vary greatly. Despite many publications and practical applications, the concept of *computational mesh management* has not yet been defined consistently, even though it is used in the context of scientific research.

The work consists of two main parts. The first covers introductory issues and presents models of computational mesh. Different models of representation of the computational meshes are presented, taking into account, among others, the structure and relationships of objects in the adaptive computational mesh. On the basis of these models a description of the functionality offered by the meshes, regardless of their data structures, in the form of *mesh operators* has been presented. With their use, it became possible to define different types of meshes, based on a set of operations they fulfil and, consequently, to define precisely what *computational mesh management* is.

The second part deals with the practical issues of realisation of the relevant parallel adaptive mesh operators on three contemporary key levels of parallel computing: vectorization, multithreading with shared memory and multiprocessing in a distributed memory model. For each of these levels important mesh operators have been selected, for which new methods of implementation have been developed, with particular emphasis on efficient use of hardware resources. During the research new algorithms have been developed, for e.g. compressed storage scheme for discrete points and assigning globally unique identifiers in distributed environments. They can be widely used in computational sciences, beyond issues related to the management of computational meshes.

All the developed solutions were presented both theoretically and practically, taking into account the results of their application for several examples from academic and industrial practice.

Podziękowania

Serdecznie dziękuję wszystkim, którzy wspierali mnie w tej pracy, w szczególności:

— promotorowi, dr hab. inż. Krzysztofowi Banasiowi, który zaprosił mnie do współpracy i otworzył przede mną świat nauki,
— kierownikowi Katedry Modelowania i Informatyki Stosowanej WIMiIP AGH, dr hab. inż. Łukaszowi Rauchowi za mobilizację i wsparcie organizacyjne,
— dziekanowi, dr hab. inż. Tadeuszowi Telejce za wsparcie i pokrycie kosztów związanych z przebiegiem przewodu,
— kolegom z zespołu, którzy wyjaśniali mi to wszystko, na czym się nie znam.

Pracę tę dedykuję mojej Żonie, która ogarniając dom i trójkę naszych dzieci, umożliwiła jej powstanie.

Spis treści

Rozdział 1

Wprowadzenie

1.1. Wstęp

Metody obliczeniowe oparte na siatkach, w tym także metoda elementów skończonych (MES), od lat wspierają efektywną analizę złożonych zjawisk fizycznych, opisywanych za pomocą równań różniczkowych cząstkowych (RRC) w przestrzeniach jedno-, dwu- i trójwymiarowych. Istnieje wiele przypadków, w których wykorzystanie siatek niestrukturalnych, wraz z adaptacją wspartą odpowiednim oszacowaniem błędu, prowadzi do najbardziej skutecznego i efektywnego wykorzystania MES. Wymagania obliczeniowe MES z dynamicznymi siatkami adaptacyjnymi mogą być nawet o kilka rzędów wielkości mniejsze niż dla MES z gęstymi siatkami lub adaptacją jednorodną. Istnieją zadania o tak dużym stopniu złożoności, że aby zachować wymaganą dokładność wyników, mogą być one rozwiązane tylko z wykorzystaniem równoległych środowisk obliczeniowych. To w naturalny sposób prowadzi do wykorzystywania środowisk obliczeń równoległych wielkiej skali. Opracowywanie aplikacji i bibliotek dla równoległej metody elementów skończonych wymaga rozwiązania wielu problemów. Istnieją technologie oraz gotowe biblioteki wspierające tworzenie takich programów, jak również kompletne aplikacje, zarówno komercyjne, jak i „wolne i otwarte oprogramowanie" (WiOO, ang. FLOSS - free and open-source software) wspierające wykonanie w środowiskach równoległych. W ostatnich latach, w ramach każdej z tych kategorii, pojawiło się wiele propozycji, artykułów i gotowych implementacji odnoszących się do równoległej, adaptacyjnej metody elementów skończonych.

Dokładność rozwiązania MES jest implikowana przez wybrane sformułowanie słabe, przestrzeń funkcji aproksymujących oraz przestrzenną (geometryczną) dyskretyzację. Jakość rozwiązania wyraża się za pomocą pojęcia *błędu rozwiązania* określającego różnicę rozwiązania dokładnego i przybliżonego. Analiza błędu rozwiązania MES może zostać przeprowadzona w ścisły matematyczny sposób tylko dla pewnej grupy problemów. Większość rzeczywistych zagadnień zawiera różne formy nieliniowości, z których wynikają zasadnicze trudności w matematycznej analizie błę-

du, w pewnych przypadkach uniemożliwiające nawet jej przeprowadzenie. Dlatego też analiza błędu rozwiązania musi uwzględniać zmiany dyskretyzacji przestrzennej. Wymaga to, by analiza błędu była dokonywana, przynajmniej częściowo, w sposób iteracyjny. Prowadzi to do dalszej komplikacji w procesie ustalania rzetelnych technik oszacowania błędu rozwiązania MES.

Jedną z najbardziej skutecznych technik zmniejszania błędu (czyli poprawiania dokładności) rozwiązania MES jest lokalna adaptacja [10]. Istnieje kilka głównych kierunków stosowania adaptacji w czasie obliczeń MES: h–adaptacja, p–adaptacja, hp–adaptacja oraz r–adaptacja. h–adaptacja zmniejsza błąd poprzez lokalne zagęszczanie siatki przez podział elementów, co prowadzi do zwiększania liczby stopni swobody (ang. Degrees Of Freedom; DOFs). P–adaptacja zwiększa lokalnie stopnień wielomianów funkcji aproksymujących, co również prowadzi do zmniejszania błędu w danym elemencie. hp–adaptacja łączy h–adaptację i p–adaptację. r–adaptacja prowadzi do zmniejszenia błędu za pomocą przesunięcia węzłów, bez wprowadzania nowych stopni swobody. Każdy rodzaj adaptacji wprowadza pewne komplikacje i problemy, stąd efektywne zastosowanie którejkolwiek z nich nie jest zadaniem prostym. Niezależnie od komplikacji z tym związanych, stosowanie dynamicznych technik adaptacyjnych zazwyczaj znacząco obniża zapotrzebowanie na moc obliczeniową oraz pamięć RAM względem siatek adaptowanych jednorodnie lub nieadaptacyjnych, choć znacząco zwiększa stopień złożoności siatki obliczeniowej.

Równoległe środowiska obliczeniowe mogą być rozważane jako środowiska z pamięcią wspólną (ang. shared memory) lub pamięcią rozproszoną (ang. distributed memory). Coraz powszechniejsze są architektury hybrydowe, jednocześnie wspierające pamięć wspólną i rozproszoną. Wszystkie trzy rodzaje środowisk borykają się z pewnymi problemami, m.in.: synchronizacją, jednoczesnym dostępem do pamięci, alokacją zasobów, sekcjami krytycznymi i innymi. W celu rozwiązania tych i innych problemów w ciągu ostatnich lat zostały opracowane różne algorytmy, struktury danych, interfejsy i mechanizmy. Na ich podstawie zostały przyjęte pewne standardy przetwarzania komputerowego. W środowiskach z pamięcią rozproszoną koncepcja *wymiany komunikatów (ang. Message Passing)* stała się *de facto* obowiązkowym standardem, a różne biblioteki komunikacyjne dostarczają implementacji *Message Passing Interface (MPI)*. Implementacje w środowiskach z pamięcią wspólną często realizują specyfikację OpenMP w celu zarządzania wykonaniem wielowątkowym.

Niniejsza praca w swojej znacznej części skupia się na badaniu i próbach rozwiązania problemów pojawiających się przy odwzorowaniu obliczeń MES na równoległe architektury komputerowe, przy wykorzystaniu współczesnych osiągnięć obliczeń wysokiej wydajności. Nowoczesne architektury obliczeniowe sprawiają, że zarządzanie siatką obliczeniową staje się niezależnym problemem w ramach ogólnego zagadnienia adaptacji.

1.2. Symulacje adaptacyjną metodą elementów skończonych a zarządzanie siatką obliczeniową

Dla zastosowania Metody Elementów Skończonych w praktyce, kluczowa jest możliwość określenia błędu rozwiązania i jeżeli nie jest ono wystarczająco dokładne, rozwiązanie ponownie problemu w celu uzyskania większej dokładności. Jednakże, by móc rozwiązać problem dokładniej potrzebne jest lepsze jego postawienie. Zastosowanie jednorodnie gęstszej siatki jest rozwiązaniem bardzo niewydajnym, ponieważ wielkość układu równań do rozwiązania rośnie ponad-liniowo względem ilości uwzględnionych stopni swobody. W tym celu bardzo pożądane jest określenie, gdzie dokładnie jest błąd i zwiększenie precyzji rozwiązania w tym właśnie obszarze. To zagadnienie jest znane jako oszacowanie błędu *a posteriori* i adaptacja (niejednorodna). Było i jest ono przedmiotem bardzo wielu badań i zastosowań praktycznych w ciągu ostatnich lat. Istnieją różne rodzaje oszacowania błędu *a posteriori* o różnej złożoności obliczeniowej i dokładności. Można zwrócić uwagę, że wśród nich są techniki oparte na *post-processingu* i odzyskiwaniu wartości pozwalających przybliżyć błąd z większą dokładnością (np.: metoda Zienkiewicza-Zhu).

Praktyczne zastosowanie niejednorodnej adaptacji, szczególnie dla zagadnień niestacjonarnych, stawia wysokie wymagania funkcjonalne i niefunkcjonalne programom symulacyjnym MES. Muszą one utrzymywać w spójności m.in. bardzo duże siatki obliczeniowe, zmieniające swoją topologię (a czasem i geometrię) wielokrotnie w czasie jednej symulacji, zagęszczając ją, przesuwając, bądź rozrzedzając w różnych miejscach. Wymaga to rzetelnej, sprawnej i efektywnej implementacji wielu funkcjonalności dotyczących wykonywania zmian w siatce elementów skończonych w sposób poprawny, szybki i skalowalny. Okazuje się bowiem, że przeniesienie opracowanych teoretycznie metod w obszar praktycznych obliczeń naukowo-technicznych stanowi problem sam w sobie. Ogół operacji, które musi realizować siatka obliczeniowa, określa się jako zarządzanie siatką – *mesh management*. Temat ten jest przedmiotem wielu publikacji, omawianych w późniejszej części tej pracy. Wynika stąd, że zarządzanie siatką obliczeniową jest w omawianym kontekście, odrębnym problemem w zagadnieniu adaptacji. Należy jednak zwrócić uwagę, że termin *management* jest bardzo wieloznaczny, szczególnie w obszarze nauk informatycznych. Samo określenie *mesh management* oddaje intuicję kryjącą się za szeregiem funkcjonalności realizowanych przez program, ale jest nieprecyzyjne. Brak formalnej definicji, czym dokładnie takie zarządzanie jest, prowadzi do sytuacji, w której różne implementacje *zarządzania siatką* używają go do określania różnych zbiorów funkcjonalności.

1.3. Równoległe architektury obliczeniowe

Nowoczesne architektury komputerowe, nie tylko w zastosowaniach naukowo — technicznych, stanowią bardzo różnorodny zbiór urządzeń charakteryzujących się odmienną specyfikacją i posiadających różne zalety i wady. Omówienie ich wszystkich nie jest przedmiotem tej pracy, ale aby umiejscowić poruszane przez nas problemy obliczeniowe w odniesieniu do rzeczywistych maszyn obliczeniowych, można przedstawić kilka klasyfikacji funkcjonujących w naukach obliczeniowych. Wraz z końcem etapu wytwarzania coraz szybszych pojedynczych rdzeni obliczeniowych [114] nastąpiło gwałtowne ożywienie zainteresowania producentów sprzętu komputerowego różnymi technikami przetwarzania równoległego [102, 7]. Przyczyniło się to zarówno do powrotu do niegdyś sprawdzonych koncepcji, jak i poszukiwaniem nowych paradygmatów przetwarzania komputerowego.

Wprowadzona w 1972 roku taksonomia Flynna [46] definiuje podział sposobów przetwarzania komputerowego na następujące kategorie w zależności od relacji strumienia lub strumieni instrukcji oraz danych:

SISD Single Instruction, Single Data - jest to przetwarzanie sekwencyjne, np.: procesory jednordzeniowe;

SIMD Single Instruction, Multiple Data - jest to przetwarzanie równoległe realizujące te same operacje na różnych strumieniach danych; współcześnie realizowane m.in.: przez przetwarzanie wektorowe, np.: procesory wektorowe oraz karty graficzne i akceleratory;

MISD Multiple Instruction, Single Data - wiele (różnych) programów przetwarza ten sam strumień danych (redundancja); w praktyce MISD jako osobna kategoria nie funkcjonuje, poza bardzo rzadkimi i specjalistycznymi zastosowaniami;

MIMD Multiple Instruction, Multiple Data - jest to przetwarzanie wielu różnych strumieni rozkazów na różnych strumieniach danych; jest realizowany m.in. przez procesory wielordzeniowe, komputery wieloprocesorowe, klastry i gridy obliczeniowe, przetwarzanie w chmurze (cloud computing).

1.3.1. Poziomy równoległości

Podział definiowany przez taksonomię Flynna nie precyzuje, na jakim poziomie rozważany jest model przetwarzania komputerowego. Uwzględniając wielopoziomowość architektur komputerowych i mnogość dostępnych opcji można wskazać, że niezależnie od taksonomii Flynna określenie ”równoległe przetwarzanie” można aktualnie odnieść do następujących poziomów [99], uszeregowanych tu od poziomu najbardziej niejawnego (ukrytego) do poziomu najbardziej jawnego:

Równoległość na poziomie bitów i pojednyczych bajtów to poziom równoległego przetwarzania, który jest współcześnie realizowany i zapewniany bezpośrednio przez sprzęt. Układy realizujące operacje dodawania czy mnożenia liczb pojedynczej czy podwójnej precyzji albo zmiennych całkowitoliczbowych są opracowywane przez twórców sprzętu. W aktualnie używanych modelach oprogramowania nie ma żadnej kontroli nad tym poziomem, tzn. np.: programista nie ma kontroli nad tym, które układy są wykorzystywane i do czego. Jest to poziom, na którym istotne znaczenie ma wielkość słowa maszynowego (WORD). Nadal jednak poziom ten ma wpływ na całkowitą wydajność wykonania aplikacji, np.: można zrównoważyć ilość i rodzaj operacji arytmetycznych, by wykorzystać jak najwięcej układów wbudowanych w procesor naraz (np.: FMA - fused multiply-add).

Równoległość na poziomie instrukcji jest realizowana przez potoki przetwarzania wewnątrz procesora (pipelining) i w większości przypadków programista nie musi się przejmować tymi zagadnieniami. Jednakże w przypadku obliczeń wysokiej wydajności, zagadnienia takie jak superskalarność, pobieranie danych z wyprzedzeniem (prefetching), przewidywanie rozgałęzień (branch prediction), lokalność czy organizacja pamięci cache (cache associativity) mogą mieć na tyle istotne znaczenie, by uwzględniać te mechanizmy w implementacji. Na tym poziomie funkcjonują również technologie, takie jak hyper-threading.

Równoległość na poziomie wektorowym jest udostępniana za pomocą rejestrów i instrukcji wektorowych takich jak instrukcje grup SSE i AVX. Oferują one jednoczesne wykonywanie tej samej instrukcji na wielu danych, których wielkość jest wielokrotnością słowa maszynowego. Na tym poziomie możliwe jest wykorzystanie opcji wektoryzacji udostępnianych przez kompilatory, przy spełnieniu dość restrykcyjnych warunków. Oznacza to, że poziom ten jest pół-jawny. Niekiedy wymaga ręcznego włączenia odpowiednich flag kompilacji oraz dostosowania kodu źródłowego. W taksonomii Flynna odpowiada to czystemu SIMD i jest realizowane m.in.: przez akceleratory i karty graficzne.

Równoległość na poziomie wątków dotyczy realizowania wielu zadań korzystających ze wspólnej przestrzeni adresowej, przy czym mogę one realizować te same lub różne polecenia(SIMD,MIMD) na różnych danych. Technologie wspierające ten poziom to np.: realizacje standardu OpenMP, OpenCL, Intel TBB i inne realizacje wielowątkowości.

Równoległość na poziomie procesów dotyczy przetwarzania również w modelu SIMD lub MIMD, z tym że wykorzystując odrębną przestrzeń adresową dla każdego procesu. Flagową technologią tego modelu w dziedzinie nauk obliczeniowych są implementacje standardu Message Passing Interface (MPI) do wymiany komunikatów, a także komunikacja za pomocą gniazd (sockets), uwspólnionego

obszaru pamięci (shared memory region) lub dostarczanych przez system operacyjny potoków (pipe).

W odniesieniu do powyższych klasyfikacji można wskazać, że w praktyce nowoczesne architektury obliczeniowe oferują różne sposoby wykorzystania równoległości. W przypadku (multi)procesorów ogólnego przeznaczenia ich ogólną wydajność teoretyczną w operacjach zmiennoprzecinkowych na sekundę (Floating point operations per second - FLOPS) można wyrazić uproszczonym wzorem:

$$FLOPS = f[Hz] \cdot N_c \cdot V_{ops} \cdot I_{spec} \tag{1.1}$$

gdzie f oznacza częstotliwość rdzenia procesora, N_c ilość rdzeni, V_{ops} ilość operacji wektorowych realizowanych jednocześnie, a I_{spec} odpowiada za dedykowane układy sprzętowe realizujące pewne operacje jednocześnie. Aby właściwie oddać proporcje co ten wzór oznacza, przytoczmy proste obliczenie dla procesora Intel Core i7 5960X (Haswell E) taktowanego zegarem 3.0GHz, uwzględniając obliczenia podwójnej precyzji(DP):

$$3.0[GHz] \cdot 8[rdzeni] \cdot 8[wektoryzacja] \cdot 2[FMA] = 384[GFLOPS] \tag{1.2}$$

Z powyższego oszacowania wynika, że pisząc program sekwencyjny bez wykorzystania równoległości, nawet jeżeli pominiemy jakiekolwiek opóźnienia wynikające z pobierania danych z pamięci, to nie osiągniemy nawet 1% teoretycznej wydajności.

Następnie możemy uwzględnić, że większość współczesnych komputerów, również w klastrach obliczeniowych, posiada dostępne akceleratory. Zatem całkowita teoretyczna wydajność obliczeniowa komputera lub węzła obliczeniowego powinna uwzględnić oba urządzenia.

$$FLOPS_{total} = FLOPS_{CPU} + FLOPS_{ACCEL} \tag{1.3}$$

Najlepsze akceleratory dostępne w czasie pisania tej pracy osiągają teoretyczną wydajność rzędu 11 [TFLOPS] [86]. Porównując jednak dla przykładu z ww. procesorem ogólnego przeznaczenia, wybierzmy akcelerator klasy średniej-wyższej np.: GeForce GTX 770 o wydajności rzędu 3.3 [TFLOPS]. Zatem dla przykładu, zgodnie z powyższym wzorem, całkowita wydajność wyniesie

$$0.384[TFLOPS] + 3.3[TFLOPS] = 3.684[TFLOPS] \tag{1.4}$$

z czego wynika, że program napisany sekwencyjnie, bez wektoryzacji i wykorzystania akceleratora, nadal z pominięciem opóźnień pobierania danych z pamięci, osiągnie wydajność mniejszą niż 0.08% dostępnej mocy obliczeniowej. Oczywiście

w praktyce ta wydajność teoretyczna nie jest osiągalna, ale należy też uwzględnić opóźnienie związane z pobieraniem danych z pamięci, które tu pomijamy. Niezależnie od uproszczenia trend ukazany w powyższych rozważaniach jest rzeczywisty i pogłębia się coraz bardziej.

1.3.2. Bariera pamięci

Dzięki staraniom twórców sprzętu komputerowego, którzy wykorzystują różne poziomy równoległości, teoretyczna wydajność obliczeniowa komputerów nieustannie rośnie mimo braku przyspieszania pojedynczych rdzeni. Jednakże osiąganie w praktyce teoretycznej wydajności, dyskutowanej powyżej, jest niezwykle trudne ze względu na pamięć. Parametrami określającymi pamięć operacyjną są

— rozmiar[GB] - określa jaka maksymalna ilość danych może być jednocześnie przechowywana,
— przepustowość[GB/s] - określa jak dużo danych może zostać dostarczonych do procesora w jednostce czasu,
— opóźnienie[s lub cykle] - określa jak długo należy czekać od zgłoszenia zapotrzebowania na dane, do ich faktycznego dostarczenia.

W odniesieniu do przyrostu szybkości pojedynczych rdzeni szybkość pamięci wzrosła na przestrzeni ostatnich kilkudziesięciu lat bardzo nieznacznie. Podstawowym problemem są tu ograniczenia technologiczne, a w ostatnich latach również już fizyczne, dotyczące sposobu przesyłania danych. Parametrem, który jest najbardziej problematyczny, okazuje się opóźnienie (latency). Ze względu na niemożność zmniejszenia opóźnień, stosuje się różnorodne techniki mające na celu zamaskowanie czasu oczekiwania (latency hiding) na dane poprzez:

— pamięć cache i hierarchię pamięci,
— zwiększanie przepustowości pamięci,
— pobieranie danych z wyprzedzeniem (data prefetching),
— przełączanie kontekstu.

Używanie tych i innych technik spowodowało skomplikowanie modelu programowania. Kolejnym istotnym rozróżnieniem pomiędzy różnymi metodami przetwarzania komputerowego jest organizacja pamięci RAM. Historycznie istniały różne mechanizmy organizacji pamięci. Cechy takie jak jednorodność przestrzeni adresowej oraz czasu dostępu do pamięci pozwalają wskazać na kilka różnych schematów organizacji pamięci:

Wspólna, Uniform Memory Access - UMA - jednorodny dostęp do pamięci RAM. W tym modelu programowania każda komórka pamięci RAM (spośród

udostępnionej przez system) może zostać zaadresowana, i sposób dostępu do każdego adresu jest taki sam, co przekłada się na (średnio) podobny czas dostępu do każdego adresu. Programowanie wysokiej wydajności wymaga uwzględnienia dwóch efektów związanych z istnieniem pamięci cache: lokalności (wykorzystywanie danych będących już w cache) oraz kolejności dostępu (dostęp do adresów pamięci w sposób sekwencyjny jest bardziej efektywny).

Wspólna, cache coherent Non-Uniform Memory Access - ccNUMA - niejednorodny dostęp do pamięci RAM z mechanizmem spójności pamięci cache. Podobnie jak w UMA pamięć RAM jest prezentowana jako jednorodny blok adresów dla każdego procesu, ale pamięć cache dla poszczególnych rdzeni jest różna. To rozwiązanie sprawdza się bardzo dobrze, w przypadku, gdy poszczególne rdzenie realizują zadania wykorzystujące różne obszary pamięci. W przypadku przetwarzania danych w tej samej pamięci konieczne jest realizowanie mechanizmu synchronizacji pomiędzy pamięciami cache poszczególnych rdzeni (cache coherency), co wprowadza dodatkowy narzut czasowy. Dodatkowo w zależności od konkretnej architektury pewne grupy rdzeni mogą współdzielić pamięć cache wyższego poziomu (cache L2,L3), co wprowadza dodatkową niejednorodność ze względu na współdzielenie danych. Lokalność również nabiera nowego znaczenia, ponieważ poza (pożądanym) dostępem do danych w pamięci cache danego rdzenia, korzystne jest unikanie (niepożądanego) współdzielenia między pamięciami cache różnych rdzeni tych samych danych.

Wspólna, RAM NUMA - jest rodzajem NUMA, w którym podział pamięci pomiędzy rdzenie jest jeszcze głębszy niż w ccNUMA. Chociaż nadal cała pamięć RAM jest objęta jedną przestrzenią adresową, to poszczególne kości pamięci są przypisane do poszczególnych rdzeni, podobnie jak cache w ccNUMA. Możliwe jest adresowanie pamięci z kości RAM przypisanej do innego rdzenia, ale jest ono kosztowne, ponieważ jest wykonywane z wykorzystaniem magistrali płyty głównej. Rozwiązanie to jest rzadziej spotykane, ale jest stosowane m.in.: w niektórych architekturach serwerowych (tzw. blade'ach), stacjach roboczych typu symmetric multi-processor (SMP) z wieloma procesorami i w niektórych kartach graficznych i akceleratorach, które udostępniają uwspólnioną przestrzeń adresową, ale pewne jej obszary leżą "bliżej" procesora lub "bliżej" karty graficznej w sensie czasu dostępu. Konsekwencją dla modelu programowania jest wprowadzenie kolejnej komplikacji, ponieważ okazuje się, że choć można adresować całą przestrzeń, to kopiowanie pewnych danych "bliżej" wybranego urządzenia jest rzędy wielkości bardziej efektywne niż po prostu pobieranie danych z "odległego" obszaru pamięci.

Wspólna, Incoherent/weak memory - Pamięć niespójna/słaba zakłada, że obszary pamięci poszczególnych procesorów są niezsynchronizowane na poziomie

sprzętowym. Oznacza to, że procesory te nie posiadają instrukcji, które zapewniają atomowe operacje na danych w odniesieniu do zawartości pamięci sąsiadujących procesorów. W praktyce pozwala to na pominięcie procesu synchronizacji cache (przeciwnie niż w ccNUMA) aż do ponownego zapisu do głównej pamięci RAM. Chociaż programista nadal ma dostępną całą pamięć RAM w jednej przestrzeni adresowej, to ma obowiązek takiego pisania aplikacji, by brak synchronizacji nie był powodem błędów w programie. Ten model pamięci jest mało popularny, ale wykorzystywany m.in.: w procesorach PowerPC (do wersji 7) oraz ARM.

Rozproszona, skupiona - (Disjoint - tihgtly coupled) pamięć RAM jest rozłączna dla każdego procesora i przestrzeń adresowa nie jest wspólna. W przypadku, gdy różne pamięci są dostępne w ramach jednego urządzenia obliczeniowego (podłączone do wspólnej magistrali, lub bardzo wydajnej sieci) można dość swobodnie wymieniać dane, choć jest to operacja o rzędy wielkości dłuższa niż korzystanie z lokalnej pamięci RAM. Parametry takie jak opóźnienie (latency), przepustowość (bandwith) oraz pewność dostępu do danych (relability) są zależne od dedykowanej infrastruktury łączącej poszczególne elementy. W tym modelu programowania znikają problemy z synchronizacją, lokalnością i kolejnością danych, ale za to każda dana współdzielona pomiędzy procesorami musi zostać w wybranych momentach jawnie skopiowana (przesłana i odebrana). Model ten realizują m.in.: starsze karty graficzne lub wyspecjalizowane serwery obliczeniowe łączące bardzo szybką siecią poszczególne węzły obliczeniowe.

Rozproszona, luźna - (Disjoint, loosely coupled) jest to model analogiczny do powyższego, ale w którym poszczególne procesory wraz z przydzieloną im pamięcią mogą znajdować się w fizycznie innej lokalizacji. Oznacza to, że opóźnienie, przepustowość oraz pewność dostępu do danych są zależne nie tylko od samego urządzenia obliczeniowego, ale także nieznanej infrastruktury łączącej jego poszczególne elementy. Ten model jest wykorzystywany w obliczeniach w chmurze (cloud computing) oraz przy realizacji usług typu HaaS (hardware as a service) udostępniającej wirtualne maszyny obliczeniowe.

1.3.3. Wybrane poziomy równoległości

Aby wykorzystać możliwości, jakie daje równoległość we współczesnych architekturach, można wskazać cztery poziomy równoległości technologicznie dostępne dla efektywnego użytkowania zasobów obliczeniowych w programach obliczeniowych. Każdy z nich jest realizowany w którymś z powyższych schematów organizacji pamięci:

Wektoryzacja na poziomie pojedynczego rdzenia obliczeniowego w procesorze ogólnego przeznaczenia. Niegdyś dobrze znane rejestry wektorowe, wykorzystywanych już w superkomputerze "Cray-1" (Los Alamos, 1976) dziś znów jest wykorzystywany, by osiągać większą wydajność obliczeniową, także w rdzeniach ogólnego przeznaczenia [25]. Wektoryzacja wymaga lokalności i uporządkowania danych na poziomie pamięci cache, co oznacza, że może być realizowana tylko w modelu pamięci wspólnej. Wiele publikacji ukazało się w ostatnich latach dotyczących tematu wykorzystania wektoryzacji w obliczeniach wysokiej wydajności. Dotyczy to zarówno rdzeni ogólnego przeznaczenia, wektoryzacji w akceleratorach [97] i kartach graficznych [123] etc. Autorzy publikacji zazwyczaj rozważają problemy zdominowane przez obliczenia (z wysoką intensywnością obliczeniową). Zastosowanie istniejącej w procesorach wektoryzacji dla zagadnień z niską początkową intensywnością obliczeniową jest odmiennym problemem. W dzisiejszych procesorach ogólnego przeznaczenia przetwarzanie wektorowe jest wspierane przez zestawy instrukcji z rodzin SSE i AVX. Teoretycznie wykorzystanie tych instrukcji pozwala przyspieszyć wykonywane zadania kilkukrotnie.

Wielowątkowość na poziomie procesora wielordzeniowego z pamięcią wspólną jest rodzajem równoległości, który wydaje się najprostszy do wykorzystania. W praktycznym zastosowaniu najprościej uzyskać skrócenie wykonania czasu programu wykorzystując ten poziom. Jednakże efektywne wykorzystanie możliwości wielordzeniowych w całym programie jest już zadaniem trudnym, a czasem wręcz niemożliwym do osiągnięcia ze względu na cechy samego programu. Niemniej jednak istnieje szereg technologii, które udostępniają nie tylko opcje wykorzystania wielu wątków, ale również próbują rozwiązać przynajmniej część problemów spadających na osoby implementujące w modelu pamięci wspólnej. Wśród technologii nastawionych na przetwarzanie wielowątkowe typu SIMD prym wiedzie OpenMP, które w momencie pisania tej pracy jest już dostępne w wersji 4.5, a wersja 5.0 jest w przygotowaniu. W modelu przetwarzania wielowątkowego MIMD nadal wykorzystywane są pthreads, C++<thread>, boost::threads. Ten poziom równoległości jest wspierany także, przez wiele innych, mniej lub bardziej specjalistycznych technologii (np.: Intel TBB etc.).

Akceleratory i ich wykorzystanie na poziomie pojedynczego węzła obliczeniowego. Pojedynczy węzeł obliczeniowy poza (multi) procesorami może dodatkowo zawierać jeden lub więcej akceleratorów. Ich wykorzystanie wymaga użycia specjalistycznej technologii np.: OpenCL, CUDA, AM++. W przypadku aktualnych modeli pamięci, w których są dostępne akceleratory, poważnym ograniczeniem w ich stosowaniu dla zagadnień związanych z siatką obliczeniową jest duże opóźnienie w dostępie do pamięci. Ponieważ operacje siatkowe są zdominowane przez nieuporządkowane dostępy do pamięci, to krytycznym czynnikiem jest w nich

opóźnienie, a nie przepustowość. Z tego względu, wysokie opóźnienia akceleratorów spowodowały, nie były one rozważane jako docelowe urządzenia w omawianych w tej pracy rozwiązaniach, choć znajdują zastosowanie w problemach zdominowanych przez obliczenia [49, 63].

Równoległość w modelu pamięci rozproszonej na poziomie klastra, gridu lub wirtualnej maszyny obliczeniowej w chmurze. Jest to poziom przetwarzania wieloprocesowego, gdzie dodatkowo procesory nie tylko nie współdzielą wspólnego obszaru pamięci, ale dodatkowo komunikacja między nimi może odbywać się przez infrastrukturę sieciową. Z tego względu, że komunikacja taka jest bardzo kosztowna (w sensie jednostki czasu), najczęstszym paradygmatem jest przesyłanie komunikatów (*Message Passing*), z jego standardem MPI oraz licznymi implementacjami np.: MPICH, MP-MPICH, winmpich,WMPI II, PaTENT MPI, MPI-BIP, MPICH-Madeleine, LAM, HP MPI, IBM's MPI,SGI's MPI, PowerMPI, MPI/Pro i inne.

Te cztery poziomy obejmują i reprezentują standardową w momencie pisania tej pracy klastrową architekturę obliczeniową składającą się z klastra posiadającego węzły obliczeniowe z procesorami wielordzeniowymi z lokalną pamięcią, wspierającymi instrukcje wektorowe, z dodatkową opcją wykorzystania akceleratora.

Korzystanie z nowoczesnych architektur obliczeniowych w praktyce sprowadza się do korzystania z ww. różnego rodzaju równoległości w oprogramowaniu obliczeniowym. Ponieważ poziomy równoległości mają różne (nieraz sprzeczne) wymagania, trudno jest wskazać jedno rozwiązanie technologiczne pozwalające wykorzystywać każdy z nich efektywnie. Mimo to, w niniejszej pracy zostaną omówione metody, które pozwalają wykorzystać przynajmniej częściowo potencjał kryjący się w przetwarzaniu równoległym.

1.3.4. Wpływ na model programowania

Należy zauważyć, jakie konsekwencje dla modelu programowania ma dominująca równoległość. Sformułowane już 1967 prawo Amdahla [8] wskazuje, że dla każdego programu wykonywanego równolegle na p procesorach, posiadającego jakikolwiek procent czasu wykonania tylko sekwencyjnego (f), istnieje teoretyczna górna granica przyspieszenia $S_{max}(p)$, które można uzyskać:

$$S_{max}(p) \leqslant \frac{p}{1 + f \cdot (p - 1)}. \tag{1.5}$$

Co prawda Gustafson wykazał w 1988 roku, że dla rozwiązywania zadań, dla których rozmiar rośnie wraz z liczbą procesorów, każdy problem może być efektywnie zrównoleglony (prawo Gustafsona), jednak jest to silnie zależne od współczynni-

ka f oraz wzrostu udziału części sekwencyjnej f w zależności od zwiększania p w konkretnym problemie.

Analizując tę kwestię, można zauważyć pewną zbieżność pomiędzy tymi modelami teoretycznymi *wykonania* równoległego a aktualnym modelem *programowania* równoległego. Niestety główne języki programowania w dziedzinie obliczeń wysokiej wydajności praktycznie nie wspierają równoległego wykonania poprzez samą składnię języka. Wręcz przeciwnie, by uzyskać program równoległy, programista musi ręcznie wprowadzać dodatkowe dyrektywy i flagi kompilatora oraz samodzielnie dbać o unikanie błędów w wykonaniu równoległym. Aktualny model programowania *de facto* jest modelem sekwencyjnym, który został tylko wzbogacony o zazwyczaj opcjonalne możliwości wykorzystania mocy obliczeniowej udostępnianej przez komputery. Oznacza to, że domyślnie większość tworzonego kodu źródłowego programów trafia niestety do części sekwencyjnej (f) w prawie Amdahla. W pewnym sensie sytuacja ta jest zadziwiająca, ponieważ aktualnie bardzo trudno jest kupić nie-wielordzeniowy komputer bez akceleratora, ale również bardzo trudno jest stworzyć aplikację bezpośrednio wykorzystującą jego moc obliczeniową. Odzwierciedla to z jak bardzo skomplikowanym zagadnieniem mamy do czynienia. Efektem takiego stanu rzecz jest sytuacja, w której osoba tworząca program wysokiej wydajności, niezależnie od jego dziedziny, musi zawsze dodatkowo znać się także na programowaniu równoległym i ew. rozproszonym. Ponadto, ze względu na brak rdzennego, rzetelnego wsparcia wbudowanego w języki programowania, zagadnienie przenoszenia implementacji na modele równoległe i rozproszone wymaga ręcznego rozwiązywania bardzo wielu kwestii technicznych i algorytmicznych.

1.4. Zarządzanie adaptacyjną siatką obliczeniową - stan badań

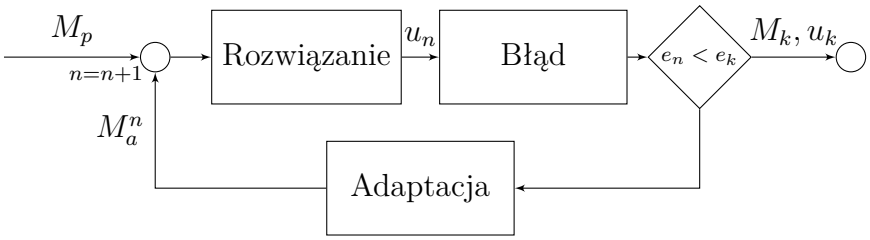
Adaptacyjne metody siatkowe, w szczególności z wykorzystaniem metody elementów skończonych, jest w wielu ośrodkach naukowych przedmiotem badań i tematem dużej ilości publikacji. Również poza badaniami naukowymi istnieją komercyjne rozwiązania omawianego zagadnienia. Na wstępie omawiania aktualnego stanu wiedzy, można wyróżnić następujący podział w dostępnych rozwiązaniach dotyczących zarządzania siatką obliczeniową:

Dostępność. Ze względu na dostępność i wdrożenie opracowanych rozwiązań:

1. Opracowania naukowe opublikowane, ale zamknięte. Opracowane algorytmy i struktury zostały zaimplementowane, ale ich realizacja nie jest udostępniona. Przykładowe wyniki zastosowania zawierają się w publikacjach np. [79, 80].
2. Opracowania naukowe opublikowane i udostępnione jako samodzielne pakiety lub biblioteki – opracowane algorytmy i struktury zostały zaimplementowane i są dostępne (w ramach określonych umów licencyjnych np. GPL) np. [69, 48]. Te rozwiązania najczęściej preferują bardzo zwartą konstrukcję kodu, z udostępnieniem dobrze zdefiniowanej funkcjonalności za pośrednictwem wąskich interfejsów,
3. Opracowania naukowe opublikowane i udostępnione jako część większego pakietu – opracowane algorytmy i struktury zostały zaimplementowane i są dostępne w ramach funkcjonalności pakietu jako całości np. StkMesh w Trilinos [53] czy libMesh [65] w PETSC [12]. Takie rozwiązania charakteryzują się często generycznym podejściem umożliwiającym użytkownikowi rozszerzanie kodu i dostosowanie go do swoich potrzeb, w tym także dołączanie własnych bibliotek.
4. Opracowania komercyjne, niedostępne do badań i analizy np. biblioteki w pakietach takich jak SolidWorks ® [124], Fluent ® [3] lub innych.

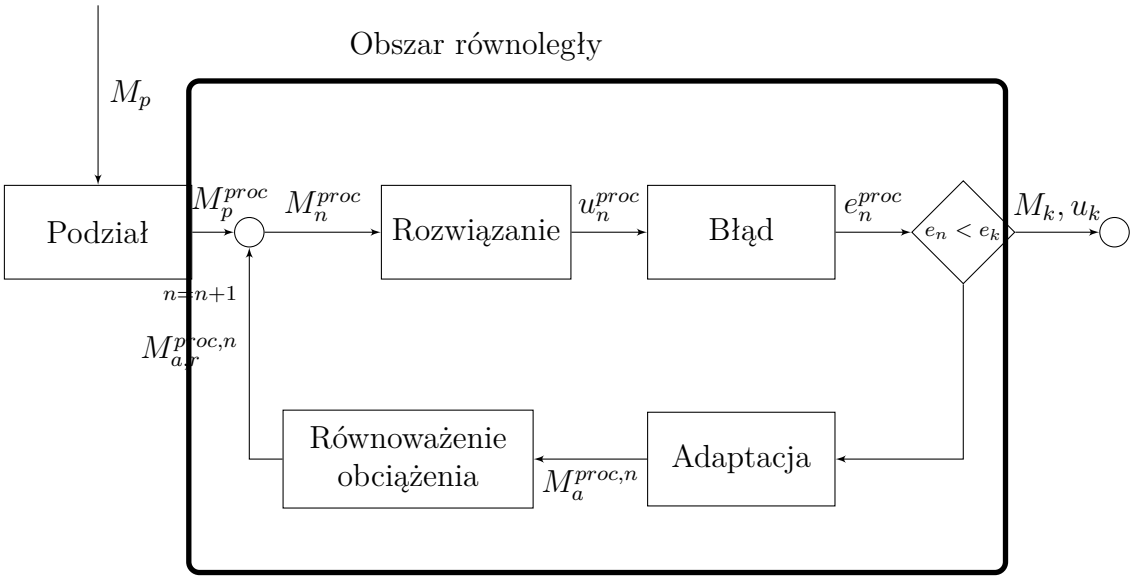
W dziedzinie analizy adaptacyjnej standardową metodyką postępowania jest rozpoczęcie modelowania od rzadkiej siatki obliczeniowej i zgrubnego rozwiązania numerycznego, a następnie – na podstawie oszacowania błędu – zastosowanie jednej ze wspomnianych już metod lokalnej adaptacji (h–, p– lub r–adaptacji) w celu poprawy rozwiązania [125] [57]. Ten podstawowy – wyjściowy dla rozważań tej pracy – schemat wykorzystujący iteracyjne oszacowanie błędu jest przedstawiony na rys. 1.1.

W celu przeprowadzenia analizy adaptacyjnej w sposób równoległy powszechnie przyjmuje się, iż zarówno rozwiązanie, obliczenie błędu, jak i adaptacja powinny zostać przeprowadzone na rozproszonej (podzielonej na obszary) siatce, wykorzystując



Rysunek 1.1: Diagram przedstawiający podstawowy schemat wykorzystania adaptacji w analizie numerycznej. Gdzie M_p – siatka początkowa, M_a^n – siatka zaadaptowana po n-tym kroku, M_k – siatka końcowa, u_n – rozwiązanie w kroku n, e_n – błąd w kroku n, e_k – akceptowalny błąd, u_k – rozwiązanie końcowe.

wiele procesorów z odrębną pamięcią [45]. Ogólny diagram koncepcyjny, przedstawiający wersję schematu 1.1 dla środowiska równoległego jest przedstawiony na rysunku 1.2.



Rysunek 1.2: Diagram przedstawiający schemat wykorzystania adaptacji w równoległych symulacjach wykorzystujących siatki obliczeniowe. Wszystkie operacje w obszarze równoległym muszą być synchronizowane między procesami. Gdzie M_p – siatka początkowa, M_p^{proc} – siatka początkowa po podziale na procesy (podobszar obliczeniowy), u_n^{proc} – rozwiązanie w kroku n w pojedynczym procesie, e_n^{proc} – błąd w kroku n w procesie, $M_a^{proc,n}$ – siatka zaadaptowana po n-tym kroku w pojedynczym procesie, $M_{a,r}^{proc,n}$ – siatka zaadaptowana po n–tym kroku w pojedynczym procesie po równoważeniu obciążenia, M_k – siatka końcowa, e_k – akceptowalny błąd, u_k – rozwiązanie końcowe.

Poprzez rozproszoną siatkę obliczeniową należy rozumieć siatkę obliczeniową podzieloną na pod-podobszary, przechowywane w różnych przestrzeniach adresowych pamięci, uwzględniając jej strukturę danych i algorytmy operujące na niej, które umożliwiają przeprowadzanie równoległych operacji podczas przeprowadzania numerycznej analizy adaptacyjnej z wykorzystaniem metod siatkowych [26]. Efektywna

i skalowalna siatka rozproszona jest konieczna do osiągnięcia wysokiej wydajności, ze względu na jej bardzo duży wpływ na ostateczną wydajność symulacji adaptacyjnej jako całości [48, 84]. Poza standardowymi operacjami związanymi z obiektami siatki takimi jak tworzenie,usuwanie czy zmiana fragmentów siatki lub informacje o klasyfikacji topologicznej, geometrycznej, łączności między obiektami musi również zapewnić wsparcie dla trzech bardzo istotnych kwestii. Po pierwsze efektywną komunikację między obiektami w siatce, również zduplikowanymi między wiele procesorów. Po drugie umożliwić migrację elementów lub całych regionów siatki między węzłami obliczeniowymi. Po trzecie umożliwić dynamiczne równoważenie obciążenia. Publikacje traktujące na powyższe tematy często skupiają się na pewnym wybranym zagadnieniu spośród wymienionych.

1.4.1. Samodzielne schematy zarządzania adaptacyjną siatką obliczeniową.

Parashar i Browne zaprezentowali DAGH (Distributed Adaptive Grid Hierarchy) [89] — rozproszoną strukturę danych siatki do równoległej nieregularnej h–adaptacji. DAGH stosuje reprezentację siatki opartą o hierarchie siatek. W przypadku rozproszonym, operacje na siatce są dokonywane lokalnie, bez angażowania komunikacji czy algorytmów synchronizacji, co jest możliwe dzięki założeniu nieregularności siatki. Równoważenie obciążenia jest dokonywane za pomocą regulowania ziarnistości tzw. bloków DAGH.

LibMesh [65] jest rozproszoną strukturą siatki opracowaną, jako część The Portable, Extensible Toolkit for Scientific Computation (PETSc) [12], w celu wspierania równoległych symulacji adaptacyjną metodą elementów skończonych z wykorzystaniem klasycznej poprawy siatki. Autorzy zdecydowanie opowiedzieli się za stosowaniem standardowej reprezentacji siatki obliczeniowej w postaci struktury danych elementy–węzły, która wspiera jednorodną h–adaptację oraz sekwencyjny podział siatki wraz z jej początkową dystrybucją między węzły obliczeniowe.

Biswas i Oliker opublikowali pracę [29], w której zaprezentowali rozproszoną strukturę siatki, przeznaczoną do adaptacyjnych obliczeń na siatkach z wykorzystaniem mechanizmów zagęszczania i rozrzedzania (ang. coarsening) siatki [87]. Sama struktura siatki składa się z wierzchołków, krawędzi i tzw. regionów przechowywanych w strukturze danych listy i dodatkowo utrzymuje – również w strukturze list – obiekty siatki, które są współdzielone na granicach podobszarów obliczeniowych, z wykorzystaniem paradygmatu *przekazywania komunikatów*. Dodatkowo, ponieważ każdy obiekt należący do siatki posiada jednoznaczny globalny identyfikator, w czasie adaptacji struktura danych jest globalnie uaktualniana w celu zapewnienia spójności danych między procesorami. Każdy obiekt siatki posiada przypisanego

właściciela w postaci procesu wybieranego losowo. Dodatkowo aplikacja wspiera dynamiczne równoważenie obciążenia, wykorzystując ParMETIS [69].

Selwood i Berzins [106] opisują ogólną strukturę danych siatki rozproszonej, wspierającą równoległą adaptację i de-adaptację. Reprezentacja siatki zawiera wszystkie poziomy siatki adaptowanej wraz z informacjami o sąsiedztwie i łączności obiektów. Równoważenie obciążenia odbywa się z wykorzystaniem pakietu Zoltan [30]. W celu zapewnienia dostępu do informacji o sąsiedztwie poprzez granice między pod-obszarami obliczeniowymi, dla każdego procesora są przechowywane wskaźniki do czworościanów granicznych, znajdujących się na sąsiednich procesorach.

W pracy [127] również jest zaprezentowana ogólna struktura rozproszonej siatki obliczeniowej nazwana PMDB (Parallel Mesh DataBase), która wspiera równoległe symulacje adaptacyjne. W PMDB dane związane z podziałem siatki są przechowywane nie na poziomie pojedynczego obiektu siatki, lecz połączenia i sąsiedztwo między procesowe są agregowane w postaci globalnych list dwukierunkowych. Struktury te dostarczają metody pozwalające odpytywać je o różne parametry, takie jak sąsiedztwo, łączność, obiekty na brzegu, a także przeprowadzać rozproszone operacje dodawania czy usuwania obiektów w siatce. Podobnie jak w innych pracach, również tutaj wprowadzono pojęcie "właściciela" obiektu, którym jest ten procesor posiadający obiekt, który posiada najniższy identyfikator. Teresco i inni [117] wzbogacili PMDB o RPM (Rensselaer Partition Model), który w jednolity sposób reprezentuje procesory heterogeniczne albo sieć węzłów obliczeniowych (lub jakąś konfiguracje obydwu) w celu zwiększenia wydajności i lepszego wykorzystania rozproszonych zasobów obliczeniowych.

Autorzy projektu FMDB (Flexible distributed Mesh DataBase) [107, 105] opracowali model podziału i zarządzania siatką obliczeniową pozwalający na dostosowanie struktur danych siatki do potrzeb użytkownika. Zdaniem twórców pozwala to na minimalizację kosztów, zarówno w sensie zapotrzebowania na pamięć, jak i czasu obliczeń. Zadaniem modelu podziału siatki jest reprezentowanie pod-obszarów obliczeniowych i zapewnienie równoległości operacjom na poziomie siatki z wykorzystaniem m.in. połączeń komunikacyjnych między procesorami. FMDB został przetestowany w użyciu z istniejącym oprogramowaniem do modelowania numerycznego.

Pakiet MeSh ToolKit (MSTK) [48] jest opisywany jako rozwiązanie, które pozwala reprezentować i manipulować niestrukturalnymi siatkami przestrzennymi o dowolnie wybranej topologii. Dzięki wielości dostępnych form reprezentacji siatki, wspartej rzetelną analizą [48], umożliwia optymalne dostosowanie siatki do wykorzystywanych algorytmów. Chociaż pakiet wspiera obliczenia na siatkach równoległych, to nie udostępnia możliwości manipulowania siatką rozproszoną. Autorzy zwracają także uwagę, że choć nie zawiera wbudowanego generatora siatek, to umożliwia łatwe

stworzenie takowego. Niestety pakiet ten nie wspiera adaptacji, choć umożliwia jej zaimplementowanie z wykorzystaniem dostarczonej funkcjonalności.

W pracy [71] został przedstawiony pakiet PARAMESH, którego autorzy skupili się na aspekcie rozwijania istniejących naukowych rozwiązań z dziedziny analizy numerycznej poprzez rozszerzenie obliczeń o możliwości równoległego wykonania z wykorzystaniem równoległej adaptacji lub jednorodnej poprawy siatki. Zaproponowane rozwiązanie wspiera także podział obszaru obliczeniowego i generację podobszarów na potrzeby obliczeń równoległych. W celu ułatwienia zrozumienia i zastosowania pakietu autorzy – w przeciwieństwie do wielu innych rozwiązań – starali się wprowadzać jak najmniej abstrakcji i wykorzystać proste i jasne konstrukcje w zakresie algorytmów i struktur danych.

Opracowany przez Burstedde, Wilcoxa i Ghattasa pakiet p4est oferuje algorytmy i struktury danych z bardzo dobrą skalowalnością [31]. Autorzy zastosowali w nim oryginalne podejście, reprezentując siatkę w postaci szeregu drzew czwórkowych (dla dwóch wymiarów) i ósemkowych (dla trzech), co w połączeniu z nadawaniem identyfikatorów obiektom na podstawie algorytmów bazujących na kodzie Mortona jest źródłem dobrych właściwości w środowiskach obliczeń wielkiej skali. Wykorzystanie koncepcji istnienia siatek w ramach innych siatek (tzw. makro-siatki) pozwoliło twórcom sprawnie zarządzać siatką w przypadku bardzo dużej ilości węzłów obliczeniowych.

Mitchell i McClain przeprowadzili rzetelne porównanie i analizę strategii adaptacyjnych dla problemów eliptycznych w przestrzeniach dwuwymiarowych [84], a swoje wnioski wykorzystali w pracach nad projektem PHAML (The Parallel Hierarchical Adaptive MultiLevel Project) [4]. Abstrahując od zawężenia dziedziny zastosowań do problemów eliptycznych w 2D, PHAML oferuje podział siatki, automatyczne równoważenie obciążenia z wykorzystaniem struktur drzewiastych, zaawansowane techniki h–,p– i hp–adaptacji, równoległą realizację poprawy siatki oraz wsparcie dla hierarchicznych metod wielosiatkowych, również w środowiskach równoległych.

The Center for Interoperable Technologies for Advanced Petascale Simulations (ITAPS) zaproponowało [42] opracowanie jednorodnego interfejsu dla narzędzi realizujących zarządzanie siatką i geometrią obliczeniową oraz innymi modułami programów obliczeniowych dla symulacji wielkiej skali. Propozycja ta dotyczy wzywań obliczeniowych realizowanych na superkomputerach z wydajnością mierzoną w petaflopsach. Zaproponowany interfejs iMesh jest już w całości lub częściowo zrealizowany w niektórych z omawianych tu projektów [116, 107].

Można zauważyć, że większość prac omawianych w publikacjach jest dostosowanych do specyficznych zastosowań siatki obliczeniowej [35, 89, 65, 29] albo wspiera tylko pewną część analizy adaptacyjnej, np. tylko etap adaptacji [13, 71, 89, 65, 90] lub jest w stanie obsługiwać tylko siatki ciągłe, bez żadnych otworów [35, 117, 71,

89, 65, 90, 106, 101, 29]. Opracowanie efektywnego i funkcjonalnego schematu zarządzania siatką rozproszoną jest zadaniem niebanalnym ze względu na złożoność strukturalną takiej siatki [73], duże wymagania funkcjonalne stawiane przez algorytmy adaptacji, synchronizacji, równoważnia obciążenia oraz udostępnienie możliwości współdziałania z algorytmami poprawiania uwarunkowania macierzy i rozwiązywania układu równań liniowych [48].

1.4.2. Realizacje schematów rozproszonego modelowania numerycznego

Autorzy wielu rozwiązań w dziedzinie równoległych symulacji metodami adaptacyjnymi, często w celu zredukowania złożoności równoległych środowisk przetwarzania komputerowego, wybierają podejście obiektowe (ang. Object Oriented apporach). Jego paradygmaty dostarczają metodyki i narzędzi do redukowania złożoności algorytmicznej i implementacyjnej poprzez zastosowanie enkapsulacji, poliformfizmu, zawierania, delegowania, abstrahowania czy wykorzystania meta-technik szablonowych. Od późnych lat 90-tych pojawiało się wiele artykułów na temat stosowania metod "OO" w dziedzinie metody elementów skończonych, w tym także wiele prac zawierających opis realizacji algorytmów obliczeniowych. Elastyczność opracowanych programów często była i jest opisywana w odniesieniu do rozszerzalności o nowe sformułowania słabe i moduły problemowe, oszacowania błędu, funkcje aproksymujące, techniki adaptacyjne, a także nowe mechanizmy rozwiązywania układów równań czy poprawy ich uwarunkowania.

Przykładem takiego programu jest FENICS Project [9], który wyróżnia architektura komponentowa skierowana na automatyczne rozwiązywanie problemu zadanego w postaci równania różniczkowego cząstkowego. Zaproponowane komponenty zapewniają narzędzia do pracy z różnymi siatkami obliczeniowymi, sformułowaniami wariacyjnymi zwykłych i cząstkowych równań różniczkowych oraz problemami algebry liniowej. Autorzy FENICS Project przyznają, że nie było ich celem osiąganie wysokiej wydajności proponowanego rozwiązania, ale wykorzystali wiele dedykowanych bibliotek wysokiej wydajności jak implementacja MPI [47], PETSC [12] , Trilinos [53], uBlas, UMFPACK [36] czy ParMetis [69].

Dla autorów szkieletu programowego Dune i jego modułu Dune-fem [38] naczelną zasadą projektową było zachowanie relacji jeden-do-jednego między matematycznym obiektami w dyskretyzowanym problemie, a jego odwzorowaniem na interfejsy programu Dune. Jak twierdzą sami autorzy [38], dzięki zaawansowanym technikom programistycznym przeprowadzone eksperymenty wykazały zarówno efektywność i dobrą stosowalność oprogramowania do dużej klasy problemów analitycznych, w tym rozwiązywania równań różniczkowych metodą elementów skończonych.

Podobne założenia, mające na celu uniknięcie nadmiernego narzutu obliczeniowego przy jednoczesnym czerpaniu korzyści z metod obiektowych zostały zaprezen-

towane przez twórców programu Deal.II [23]. Zapewniają oni, że skuteczna realizacja tego celu, wymaga właściwego oddzielenia różnych pojęć w MES: siatek, przestrzeni elementów skończonych, stopni swobody, a następnie udostępnienie użytkownikowi możliwości wybierania dowolnej kombinacji spośród nich, w tym także rodzaju elementów czy rodzaju całkowania. Twórcy podkreślają także, że bardzo ważne jest, by starać się używać standardowych wzorców programistycznych i unikać nadmiernej generalizacji oprogramowania. Co ciekawe, podają nawet negatywny przykład tej ostatniej zasady w postaci czeskiego programu OOFEM [92].

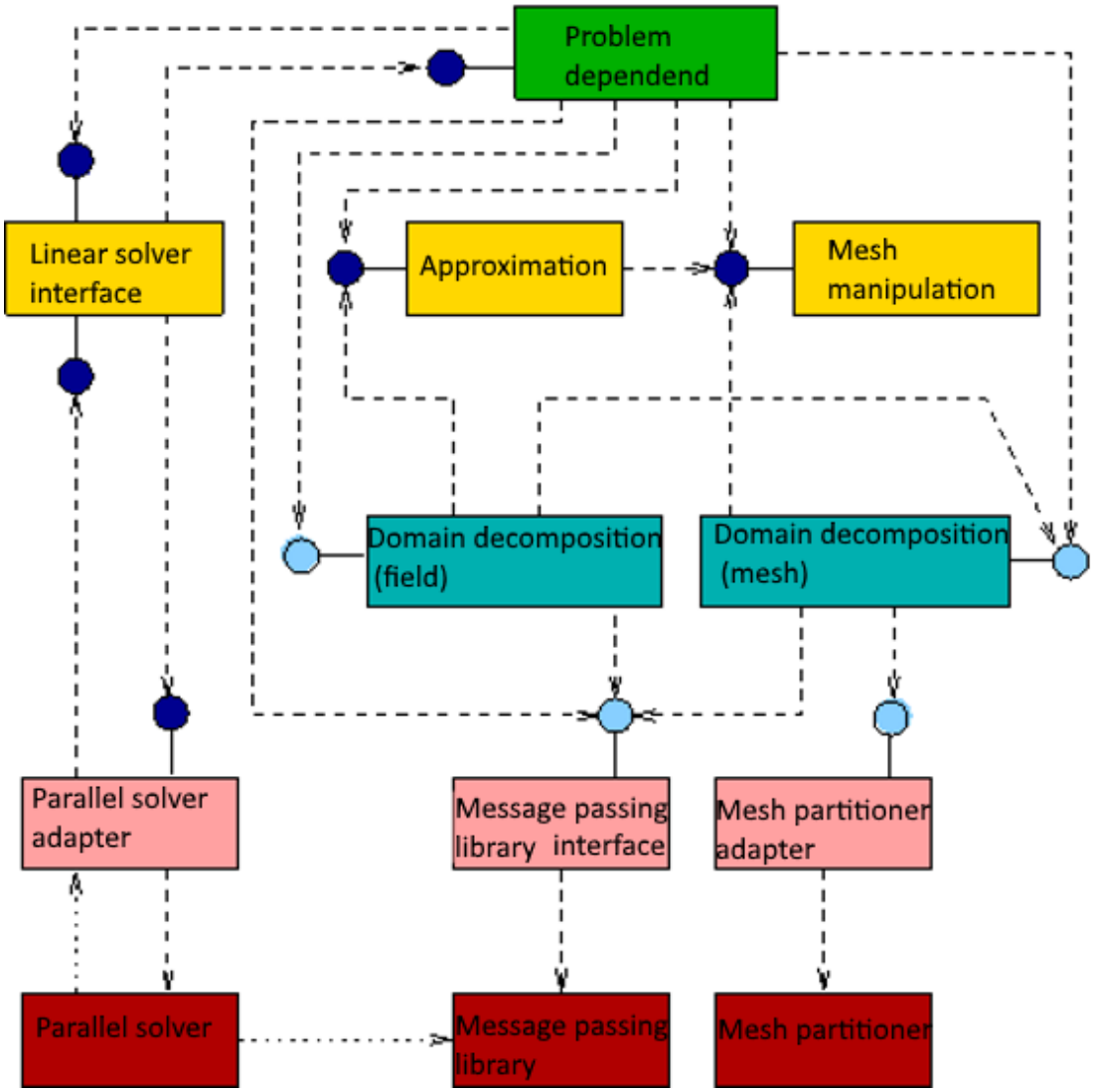
Interactive Parallel Multigrid FEM Simulator [61] jest programem, którego autorzy skupili się na dostarczaniu rozwiązań opartych o MES w czasie rzeczywistym. W opublikowanych pracach zwracają uwagę, że kluczowym aspektem jest osiąganie wymaganej dokładności rozwiązania w założonym czasie. Proponowana architektura jest oparta na dwóch głównych komponentach: serwerze symulacyjnym oraz graficznej konsoli, pracujących w modelu klient-serwer. Twórcy zapewniają, iż interfejs klienta pobierając dane o przyłożonych obciążeniach za pomocą programu graficznego, jest w stanie przesłać je do serwera obliczeniowego, odebrać rozwiązanie odkształceń obciążonych obiektów i wyświetlić efekt na ekranie – w czasie rzeczywistym. Klient odbiera nową konfigurację obiektów od serwera za każdym razem, gdy jest gotowy na wyświetlenie kolejnej ramki, a jednocześnie ciągle przesyła do serwera informacje o zmianach w obciążeniu obiektów. Zespół nadal pracuje nad rozwiązaniem pozwalającym w sposób interaktywny symulować interakcje pomiędzy wieloma obiektami. Jednocześnie przyznają, że to zadanie wymaga opracowania "sprytnego schematu wykrywania kolizji, który aktualnie jest wymagający obliczeniowo".

Podział obszaru obliczeniowego na podobszary (ang. Domain Decomposition) oraz problem dynamicznego równoważenia obciążenia także był i jest przedmiotem prac naukowych i publikacji. Między innymi biblioteki Metis i ParMetis [69] są udaną realizacją opracowanych algorytmów podziału i równoważenia obciążenia przestrzeni obliczeniowej. Wiele programów MES korzysta z nich, zamiast opracowywać własne rozwiązania. Oszacowanie błędu, szczególnie w sposób iteracyjny, oraz automatyczna adaptacja są powodem znacznego skomplikowania w procesie praktycznej realizacji zarządzaniem siatką w środowiskach równoległych. Także na ten temat pojawiały się zarówno artykuły naukowe, jak i implementacje aplikacji oraz bibliotek wysokiej wydajności [87]. Należy także podkreślić pojawianie się rozwiązań skierowanych na architektury posiadające dziesiątki i setki tysięcy rdzeni obliczeniowych [107] [105].

1.4.3. Modularność i projekt ModFEM

Badanie przedstawione w niniejszej pracy były przeprowadzane przy zaangażowaniu autora w rozwój pakietu ModFEM [79, 80] i zostały zwieńczone implementacją opracowanych rozwiązań w tym pakiecie [82, 18, 19, 76, 81, 16, 78, 93,

111, 110, 21, 109, 17]. ModFEM to generyczny, modularny szkielet programowy do równoległych obliczeń naukowych metodą elementów skończonych. Głównym założeniem projektowym jest podział całego szkieletu na moduły, połączone poprzez precyzyjnie zdefiniowane wąskie interfejsy, jak pokazano na rys. 1.3. Na szczycie architektury modularnej znajdują się moduły odpowiedzialne za modelowanie konkretnych zjawisk fizycznych, które dodatkowo mogą być łączone z innymi modułami problemowymi w super-moduły. Wszystkie poniższe moduły są dostępne do wglądu (kody,dokumentacja) w repozytorium projektu ModFEM [95], stąd ich szczegółowa zawartość nie jest tu omawiane ze względu na objętość.



Rysunek 1.3: Diagram architektury modularnej ModFEM [80].

Struktura modularna pozwala na przenoszenie kodów sekwencyjnych do środowisk równoległych z rozproszona pamięcią. Ponadto modularność wspiera rozwiązywanie problemów ze sprzężeniem wielu różnych zjawisk fizycznych oraz modelowanie wieloskalowe [78, 72, 16, 93, 111, 21, 109, 17]. Posiada on możliwość dostosowanie się

do sprzętu, na którym jest uruchamiany. Elastyczność i modularność projektu ModFEM dotyczy nie tylko warstwy problemowych (jakie zjawiska program symuluje) [18, 76, 78] i programowej (w jaki sposób dostosować program do swoich potrzeb), ale także warstwy sprzętowej [81, 77]. Istotną cechą architektury modularnej jest możliwość oddzielenia zagadnień np.: siatki i aproksymacji, a następnie badanie ich w separacji względem siebie.

1.4.4. Rodzaje schematów zarządzania siatką i istotne cechy.

Na podstawie przeglądu istniejących rozwiązań, można wyróżnić pewne cechy proponowanych schematów, które umożliwiają kategoryzację względem różnych czynników.

Równoległość w modelu pamięci wspólnej. Tutaj możemy wprowadzić rozróżnienie, ze względu na rodzaj technologii realizującej:

1. Model pamięci wspólnej z wykorzystaniem procesora wielordzeniowego lub wielu procesorów jedno- lub wielordzeniowych za pomocą technologii realizujących standard OpenMP[88] lub phtrads[85], Intel TBB[100] czy C/C++ <multithreading>[113] [79, 53].
2. Model pamięci wspólnej z wykorzystaniem akceleratora lub koprocesora z własną pamięcią np. z użyciem bibliotek CUDA, lub OpenCL [79].
3. Model hybrydowy pamięci wspólnej łączący oba powyższe np. w technologii OpenCL lub wykorzystujący więcej niż jedno rozwiązanie technologiczne naraz [108].
4. Model wykorzystujący przetwarzanie sekwencyjne dla pamięci wspólnej [48, 116].

Równoległość w model pamięci rozproszonej. Najczęściej zakłada wykorzystanie paradygmatu:

1. Przesyłania komunikatów, który jest realizowany z wykorzystaniem implementacji standardu MPI [47] (ang. Message Passing Interface) [48, 79, 31].
2. Autorskich protokołów komunikacyjnych, najczęściej opartych o wykorzystanie technologii gniazd systemowych (ang. socket).

Tworzenie geometrii W zależności od tego, w jaki sposób jest wprowadzana geometria siatki obliczeniowej, można wyróżnić następujące rozwiązania:

1. Źródłem jest geometria w postaci (pliku) CAD/CAM [56].
2. Źródłem jest geometria w postaci pliku siatki (w pewnym zdefiniowanym formacie) [48, 116, 79].
3. Rozwiązanie posiada interfejs użytkownika pozwalający na definicję geometrii obliczeniowej [124, 3, 50].

Generowanie i re–generowanie siatki W zależności od możliwości zaproponowanego schematu zarządzania siatką można wyróżnić trzy grupy rozwiązań w tej materii:

1. Wymagana jest wygenerowana siatka w postaci pliku w akceptowalnym formacie [48, 116, 79].
2. Możliwa jest generacja siatki w ramach schematu zarządzania siatką [12].
3. Możliwa jest ponowna generacja siatki (re–generacja) na podstawie dostarczonego pliku z siatką [31].

Podział siatki i dystrybucja siatki. W zależności do sposobu podziału i dystrybucji siatki między procesorami:

1. Osobny generator siatki: tworzenie pod-obszarów obliczeniowych jest delegowane do specjalnych podprogramów lub bibliotek (np. Metis, ParMetis [69]). To rozwiązanie jest wykorzystywane w większości publikowanych prac [79].
2. Mechanizm podziału jest wbudowany w program: schemat zarządzania siatką jest w stanie samodzielnie dokonać podziału siatki – takie rozwiązanie jest stosowane w generatorach siatek [56, 31]. W ramach takich schematów można wyróżnić rozwiązania o następującej kolejności czynności:
   a) Wczytanie geometrii (CAD/CAM), następnie zgrubne partycjonowanie powierzchni, rozesłanie do procesów, generacja dokładnej siatki w procesach, synchronizacja.
   b) Generacja zgrubnej siatki, partycjonowanie, rozesłanie do procesów, adaptacja siatki zgrubnej.
   c) Generacja zgrubnej siatki, generacja dokładnej siatki powierzchniowej, partycjonowanie, rozesłanie do procesów, adaptacja siatki powierzchniowej, generacja siatki objętościowej.

Operacje wejścia/wyjścia dla siatki. Standardem jest wykorzystanie systemu plików w celu zapisu i odczytu siatek:

1. Pierwszy proces czyta siatkę i następnie – po podziale – rozsyła do innych procesów [48, 79].
2. Każdy proces czyta całą siatkę, ale dokonuje obliczeń tylko na przydzielonym fragmencie [79].
3. Pewien wybrany podzbiór procesów dokonuje odczytu i podziału siatki [79].
4. Każdy proces musi posiadać swój osobny plik z siatką początkową.

Synchronizacja międzyprocesowa siatki obliczeniowej. W zależności od sposobu, w jaki schemat zarządzania organizuje synchronizację zmian w podobszarach siatki, można wyróżnić.

1. Wykorzystanie nakładania/zachodzenia na siebie obszarów siatek, powodujące duplikowanie pewnych elementów na brzegu podobszarów (ang. ghost elements) [48, 79, 31].

2. Drobnoziarnista komunikacja synchronizująca pojedyncze zmiany w obiektach siatki z wykorzystaniem opracowanego protokołu propagującego te zmiany w siatce.
3. Gruboziarnista komunikacja synchronizująca w każdej iteracji całe obszary brzegowe z innymi procesorami.
4. Opcja pierwsza w połączeniu z drugą lub trzecią.

Modyfikacje siatki. W zależności od tego, jak dokonywane są modyfikacje siatki w środowisku równoległym, można wyróżnić:

1. Sekwencyjne, w których modyfikacje struktur siatki są dokonywane w sposób sekwencyjny.
2. Równoległe, w których modyfikacje siatki są dokonywane w sposób równoległy [31].
3. Lokalnie sekwencyjne w połączeniu z równoległą aktualizacją danych między procesami [79].
4. Modyfikacje równoległe nie są możliwe po podziale siatki [48].

Identyfikowanie obiektów w siatce. Każdy schemat musi zawierać sposób jednoznacznego rozróżniania obiektów. Najczęściej wykorzystywane koncepcje to:

1. Globalny identyfikator obiektu [116] i jawne odwołania międzyprocesowe.
2. Lokalny identyfikator w połączeniu z identyfikatorem procesu–właściciela [48, 79].
3. Kod/porządek Mortona, generacja identyfikatorów wykorzystująca funkcje wypełniające przestrzeń [31].
4. Schemat Bergera-Oligera (ang. Berger–Oliger AMR Scheme) [28, 27].

Hierarchia obiektów siatki. W zależności od tego, czy schemat zarządzania siatką wykorzystuje hierarchie, a jeżeli tak, to jaką:

1. Siatki niehierarchiczne, bez wielostopniowych zależności między obiektami [48].
2. Siatki reprezentowane za pomocą struktury drzewa, szczególnie drzewa ósemkowego (Octree) [31] lub innej struktury mającej cechy hierarchiczne i wielopoziomowe, w których każdy obiekt ma przypisany pewien poziom, na którym "jest"(ang. levels/patches) [79].
3. Siatki reprezentowane przez skierowane grafy acykliczne (DAG – Direct Acyclic Graph) [116, 71]. W tej kategorii znajdują się także siatki reprezentowane za pomocą autorskich rozwiązań bazujących na grafach, takich jak Scalable Distributed Dynamic Grid (SDDG) i Distributed Adaptive Grid Hierarchy (DAGH) [89].

Reprezentacja topologiczna siatki. Opis siatki niezależny od faktycznego geometrycznego kształtu obiektów w przestrzeni, przedstawiający tylko ich wzajemne sąsiedztwo może być reprezentowany w strukturze danych jako:

1. Pełna reprezentacja topologiczna siatki [48, 107, 116, 79].
2. Zredukowana reprezentacja topologiczna siatki [65, 116, 53, 107, 48, 31].
3. Zmienna reprezentacja topologiczna siatki. Występuje wtedy, gdy w reprezentacja obiektu w siatce jest różna w zależności od umiejscowienia w przestrzeni lub hierarchii [53, 107, 48].

Podstawowa technologia implementacyjna. Wybór języka programowania zastosowanego do implementacji często pociąga za sobą możliwości, ale i ograniczenia, dotyczące dostępnych bibliotek. Dlatego też, ze względu na ten aspekt można wyróżnić podział na implementacje wykorzystujące:

1. C++ wraz z jego biblioteką standardową (przestrzeń std::) oraz innymi bibliotekami (np. boost::) [116, 53, 65, 30, 92].
2. C (najczęściej w wersji C99) [79, 31, 71, 30].
3. Fortran (najczęściej Fortran90) [71, 4, 30].

1.5. Cel i zakres pracy

1.5.1. Motywacja

Przedstawiona we wstępie (rozdział 1) charakterystyka adaptacyjnej MES oraz przegląd istniejących rozwiązań (rozdział 1.4) prowadzą do wniosku, że zarządzanie siatką adaptacyjną ma fundamentalny wpływ na realizację modelowania wykorzystującego siatki w środowiskach rozproszonych, co wynika z:

— podstaw matematycznych metody elementów skończonych i jej wymagań obliczeniowych [112, 118, 40, 125],
— rozwoju architektur komputerowych [41, 120, 115, 42].

Jednocześnie istotność zagadnienia jest podkreślana przez fakt, iż efektywne zarządzanie adaptacyjną siatką elementów skończonych z wykorzystaniem architektur równoległych, rozproszonych i hybrydowych jest aktualną tematyką badań i publikacji [98, 34, 70, 119, 104, 59, 35, 43, 122, 117, 32, 67, 13, 71, 89, 127, 87, 65, 90, 37, 106, 101, 71, 4, 30, 116, 53, 92, 74, 62, 103, 60]. Wielość publikacji w dziedzinie oraz mnogość proponowanych rozwiązań, wraz z ilością byłych, aktualnych i nowych projektów naukowo-badawczych, sektora prywatnego [124, 3] oraz publicznego [66, 42], skierowanych, częściowo lub wyłącznie, na rozwój metod zarządzania siatką adaptacyjną, szczególnie w środowiskach równoległych, wskazuje na to, że tematyka ta jest aktualna, ważna i pożądana.

Dodatkowo warto zauważyć, że szereg uznanych nagród w dziedzinie aplikacji obliczeń wysokiej wydajności [5, 2, 1] zostało przyznanych rozwiązaniom z dziedziny obliczeń numerycznych wykorzystującym siatki [96, 83, 108, 12].

Można więc stwierdzić, że szukanie efektywnych sposobów realizacji wymagań względem siatek obliczeniowych stawianych przez modelowanie numeryczne jest istotnym zadaniem naukowym. Jednocześnie odwzorowanie zagadnienia zarządzania siatką adaptacyjną na nowoczesne platformy jest aktualnym i nadal otwartym problemem, w szczególności w dziedzinie obliczeń wysokiej wydajności.

1.5.2. Definicja problemu.

Rozważanym problemem będzie *zarządzanie* siatką obliczeniową rozumiane jako sposób realizacji metod umożliwiających równoległe modelowanie adaptacyjną metodą elementów skończonych uwzględniające:

— h– adaptację oraz siatki 1-nieregularne,
— równoległość w modelu pamięci wspólnej,
— równoległość w modelu pamięci rozproszonej.

Dodatkowo uwzględnione będzie wsparcie dla wielu siatek jednocześnie, wielu pól aproksymacji na jednej siatce oraz siatek hybrydowych (różne rodzaje elementów).

1.5.3. Cele pracy

Celem poniższej pracy jest:

1. Określenie w sposób ścisły czym dokładnie jest *zarządzanie* rozproszoną siatką obliczeniową wspierającą symulacje z wykorzystaniem adaptacyjnej metody elementów skończonych.
2. Zidentyfikowanie, które elementy ww. *zarządzania* mają istotny wpływ na realizowanie modelowania numerycznego na nowoczesnych architekturach obliczeniowych oraz które miary są dla nich odpowiednie.
3. Dla wybranych zagadnień opracowanie i zweryfikowanie nowych rozwiązań posiadających pewne zalety względem dotychczas stosowanych metod.

1.5.4. Zakres pracy

Praca ta koncentruje się na zagadnieniu realnego wykorzystania opracowanych metod adaptacyjnych na współczesnych architekturach komputerowych. Z tego względu zagadnienie adaptacji w metodzie elementów skończonych pod względem teoretycznym nie jest tutaj analizowane, tylko przyjmowane za punkt wyjścia jako problem modelowy do zrealizowania. Ze względu na specyfikę operacji na siatkach obliczeniowych, które znacznie bardziej odpowiadają za *dostarczanie* danych do obliczeń niż samo *przeprowadzanie obliczeń*, co wymaga zastosowania złożonych grafowych struktur danych nieodpowiednich dla GPU, z zakresu rozważań wyłączone jest stosowanie siatek obliczeniowych na akceleratorach i kartach graficznych. Domyślnym opisem siatki będzie siatka z wierzchołkami stacjonarnymi. Poprzez adaptację będzie rozumiana h-adaptacja z wykorzystaniem 1–nieregularności, ponieważ adaptacje typu p oraz r zasadniczo nie wprowadzają większych zmian w strukturze topologiczno-geometrycznej siatki, więc w sensie funkcjonalności oferowanych przez siatkę adaptacja typu h stawia największe wymagania.

1.5.5. Zawartość pracy

Aktualny stan wiedzy dotyczący omawianego zagadnienia jest przedstawiony w rozdziale 1.4, gdzie omówione są publikacje i zastosowania z ostatnich lat. Znajduje się tam również ich zbiorcza klasyfikacja ze względu na wybrane kryteria.

W rozdziale 2 znajduje się zwięzły opis kluczowych pojęć dla adaptacji w metodzie elementów skończonych, w ramach której prowadzone są niniejsze rozważania.

Rozdział 3 opisuje istniejące w literaturze modele reprezentacji siatki obliczeniowej, które stanowią podstawę do stworzenia w rozdziale 4 ścisłego opisu, czym jest zarządzanie siatką.

Wspomniany rozdział 4 prezentuje opracowaną formalną reprezentację *zarządzania* siatką obliczeniową oraz wskazuje na odpowiednie kryteria oceny wraz z odpowiednimi miarami do nich.

W rozdziałach 5, 6, 7 przedstawione są przeprowadzone badania, analizy teoretyczne i symulacje wykonane w celu zidentyfikowania istotnych elementów zarządzania siatką w kontekście wykorzystania nowoczesnych architektur komputerowych odpowiednio na różnych poziomach równoległości. Dla każdego z tych poziomów znajduje się tam opis opracowanego rozwiązania, odnoszącego się do rezultatów badań:

— w rozdziale 5 dotyczącym wektoryzacji jest to algorytm wykorzystujący tablice decyzyjne do przyspieszenia procesu adaptacji siatki,
— w rozdziale 6 dotyczącym wielowątkowości w modelu pamięci wspólnej jest to algorytm zapisu i kompresji współrzędnych siatki obliczeniowej,
— w rozdziale 7 dotyczącym wieloprocesowości w modelu pamięci rozproszonej jest to algorytm jednoznacznego identyfikowania obiektów siatki.

Opracowane metody zostały włączone do autorskiej biblioteki programowej realizującej zarządzanie siatką. Przykłady jej praktycznych zastosowań zawiera rozdział 8.

Rozdział 9 zawiera podsumowanie całości prac.

1.5.6. Wkład w rozwój dziedziny i elementy nowatorskie w pracy

Niniejsza praca podsumowuje rezultaty osiągnięte już wcześniej i opisane w publikacjach oraz przedstawia wyniki dotąd niepublikowane:

— ścisłe zdefiniowanie operatorów, które realizują funkcjonalności adaptacyjnych siatek obliczeniowych,
— ścisłe zdefiniowanie pojęcia zarządzania siatką obliczeniową w postaci zbiorów operatorów siatki wraz z kategoryzacją dla siatek hybrydowych, adaptacyjnych i rozproszonych,
— uzyskanie reprezentatywnych profili wykorzystania operatorów w wybranych przypadkach testowych oraz rzeczywistych symulacjach dla różnych rodzajów równoległości,
— implementacja i optymalizacja metod opowiadającym operatorom dla rozproszonej adaptacyjnej siatki obliczeniowej,
— opracowanie nowego algorytmu wysokiej wydajności zapisu i kompresji współrzędnych siatki obliczeniowej,
— opracowanie nowego algorytmu do jednoznacznego identyfikowania obiektów siatki w przestrzeni bez konieczności komunikacji w modelu pamięci rozproszonej oraz bez konieczności synchronizacji w modelu pamięci wspólnej.

Rozdział 2

Adaptacja w metodzie elementów skończonych.

2.1. Metoda elementów skończonych.

Wiele naturalnych procesów można opisać na poziomie makroskopowym, bez wgłębiania się w zachowanie cząsteczek, atomów, elektronów itd. Wartości takie jak odkształcenie, gęstość, ciśnienie, temperatura, koncentracja, pole elektromagnetyczne można opisać za pomocą pól wartości skalarnych lub wektorowych, których rozkład przestrzenny i czasowy uzyskujemy przez rozwiązanie równań różniczkowych cząstkowych(RRC) [40]. Poza bardzo specyficznymi przypadkami, równania te nie posiadają znanych rozwiązań analitycznych.

Metoda Elementów Skończonych (rzadziej określana jako Metoda Elementu Skończonego, bardziej oddając oryginalne znaczenie nazwy angielskiej) pozwala znajdować przybliżone rozwiązania problemów opisywanych za pomocą RRC. Zagadnienie to było i jest poruszane w wielu specjalistycznych pracach dotyczących zarówno teoretycznego, jak i praktycznego podejścia do analizy zagadnienia m.in. [118, 33, 125, 57, 112, 39, 40, 121].

Dla różnych zagadnień eliptycznych, parabolicznych i hieperbolicznych zostały przeprowadzone i opublikowane wyprowadzenia oraz dowody wskazujące na możliwość praktycznego rozwiązywania problemów za pomocą MES. Przeważnie uwzględnia to wyprowadzenie tzw. sformułowania słabego zagadnienia, wykazanie równoważności rozwiązania słabego i klasycznego (mocnego), przy założeniu istnienia i jednoznaczności rozwiązania oryginalnego sformułowania różniczkowego. Dodatkowo wykazuje się uwzględnienie różnego rodzaju warunków brzegowych Dirichleta, Neumanna lub trzeciego rodzaju (Robina). Uwzględnia się zarówno zjawiska niezależne od czasu, jak i zmieniające się w czasie, także w sposób nieliniowy, ze szczególnym uwzględnieniem badania zbieżności różnych metod. Specyfika MES polega z jednej strony na wykorzystaniu wspomnianego wcześniej całkowego sformułowania słabego, a z drugiej na dyskretyzacji polegającej na podziale obszaru obliczeniowego na elementy skończone i aproksymacji za pomocą funkcji kawałkami wielomianowych.

Bardzo istotnym zagadnieniem w MES jest analiza błędu, gdzie wykazuje się, iż uzyskane rozwiązanie przybliżone w wybranej przestrzeni aproksymacji jest najlepszym przybliżeniem rozwiązania dokładnego w tej przestrzeni, z czego wynika, że o błędzie aproksymacji decyduje nie dobór konkretnych funkcji, ale przestrzeni, do której należą.

Wyprowadzenia te i dowody są niezwykle ważne, ponieważ stanowią fundament, bez którego tematyka tej pracy nie miałaby w ogóle podstaw. Jednakże, w zakresie teorii MES nie wprowadzamy tutaj żadnych innowacji, które wymagałyby przeprowadzania żmudnego procesu wyprowadzania opisanych już wielokrotnie zależności. Ograniczymy się do minimum, wprowadzając jedynie pojęcia konieczne do zrozumienia terminologii używanej w badaniach nad siatkami adaptacyjnymi MES.

2.2. Oszacowanie błędu rozwiązania

Wyróżnia się dwa podstawowe oszacowania błędu przybliżenia dostarczanego przez Metodę Elementów Skończonych, oba dostarczające informacji różnego rodzaju. Po pierwsze oszacowanie błędu *a-priori*, które służy do określenia zbieżności wybranej konkretnej metody elementów skończonych. Jak wskazuje nazwa, nie wymaga znajomości konkretnych wartości aproksymowanego rozwiązania w postaci stopni swobody w siatce obliczeniowej, ponieważ dotyczy ona wybranego zagadnienia jako takiego, a nie konkretnej symulacji. Po drugie oszacowanie błędu *a-posteriori*, które wymaga znajomości obliczonego rozwiązania przybliżonego. Dla konkretnego uzyskanego rozwiązania daje ono znacznie dokładniejsze oszacowanie błędu niż oszacowanie *a-priori*, a ponadto pozwala wskazać, w których fragmentach siatki obliczeniowej przybliżony błąd rozwiązania jest największy, co stanowi kluczową informację dla stosowania metod niejednorodnej adaptacji. Dla praktycznego zastosowania adaptacji oba rodzaje oszacowania błędu mają istotne znaczenie:

błąd a-priori będący podstawą dowodzenia zbieżności rozwiązania MES do rozwiązania dokładnego rozważanego zagadnienia, pozwala na matematyczną kontrolę nad poprawną realizacją procesu adaptacji, ponieważ dostarcza informacji o tym, jak błąd się powinien zmieniać wraz ze zmianą rozmiaru h elementu skończonego;

błąd a-posteriori jest konieczny do przeprowadzenia automatycznej adaptacji niejednorodnej w czasie obliczeń, ponieważ dostarcza informacji o tym, w których elementach należy zastosować adaptację, by jak najbardziej zmniejszyć i wyrównać błąd w całej siatce.

2.2.1. Oszacowanie *a-priori*

Głównym zadaniem oszacowania błędu *a-priori* jest określenie rzędu zbieżności danej metody elementów skończonych, co zazwyczaj dokonuje się poprzez przedstawienie oszacowania wybranej normy $\|\cdot\|$ różnicy nieznanego rozwiązania dokładnego u i rozwiązania przybliżonego u_h wyrażonej za pomocą rozmiaru elementu h i pewnej dodatniej stałej α:

$$\|u - u_h\| = O(h^\alpha) \tag{2.1}$$

Nawet jeżeli nie znamy dokładnej wartości wyrażenia po prawej stronie, to możemy oszacować, jak szybko maleje błąd rozwiązania wraz ze zmieszaniem rozmiaru elementu h. Łatwo zauważyć, że ponieważ α jest dodatnie, to niezależnie od niego dla $h \to 0$ widać, że $\|u - u_h\| \to 0$. Oznacza to, że zmniejszanie rozmiaru elementu zawsze da nam mniejszy błąd oraz że gdybyśmy przyjęli nieskończenie małe elementy, to otrzymalibyśmy bezbłędne rozwiązanie w danej aproksymacji. Niestety, oznacza to również, że ilość elementów skończonych rośnie odwrotnie proporcjonalnie, w kierunku nieskończoności.

W zależności od przyjętego przypadku wspomniana wcześniej norma $\|\cdot\|$ może przyjąć różną postać, najczęściej w zależności regularności zadania.

Norma H^1

Norma przestrzeni Soboleva $H^1(\Omega)$ jest definiowana jako

$$\|u - u_h\|_{H^1} = \sqrt{\int_\Omega (u' - u_h')^2 dx + \int_\Omega (u - u_h)^2 dx} \tag{2.2}$$

i pewnych przypadkach odpowiada normie energetycznej zdefiniowanej dla konkretnego problemu [112], ale jest od wybranego zagadnienia modelowego niezależna. Jak widać, mierzy ona jednocześnie błąd funkcji, jak i błąd ich pochodnych.

Norma L_2

Norma określana jako L_2 jest jedną z norm L_p definiowanych dla $1 \leqslant p \leqslant \infty$ jako

$$\|u - u_h\|_{L_p} = \sqrt[p]{\int_\Omega (u - u_h)^p dx} \tag{2.3}$$

Zazwyczaj definiuje się ją dla $p = 2$, ale w pewnych przypadkach inne wartości p również są używane, jeżeli jest to konieczne lub wygodne. Norma ta mierzy tylko błąd zależny od wartości funkcji, bez uwzględniania ich pochodnych. Czasem używa się też normy maksymalnej oznaczanej jako L_∞:

$$L_\infty = max_\Omega \|u - u_h\|. \tag{2.4}$$

2.2.2. Oszacowanie *a-posteriori*

Jak już zostało to zaznaczone wcześniej, oszacowanie błędu *a-posteriori* jest podstawą do sterowania h-adaptacją. Jednocześnie należy zaznaczyć, że w wielu praktycznych inżynierskich zastosowaniach analiza błędu dotyczy nie tylko całościowego ujęcia błędu rozwiązania, ale często błędu w konkretnym obszarze siatki lub błędu obliczenia specyficznych parametrów bazujących na obliczonym rozwiązaniu. W tym drugim przypadku metody, w których analiza błędu *a-posteriori* jest używana do sprawdzania specyficznych kryteriów dokładności, wprowadza się pojęcie tzw. oszacowań ukierunkowanych (goal-oriented adaptation; adaptacja celowa) [121]. Ze względu na zadanie, jakie ma do spełnienia oszacowanie *a-posteriori*, powinno ono być nie tylko wiarygodne w sensie obliczenia błędu bliskiemu lub proporcjonalnemu do błędu rzeczywistego, ale także nie może być zbyt kosztowne numerycznie.

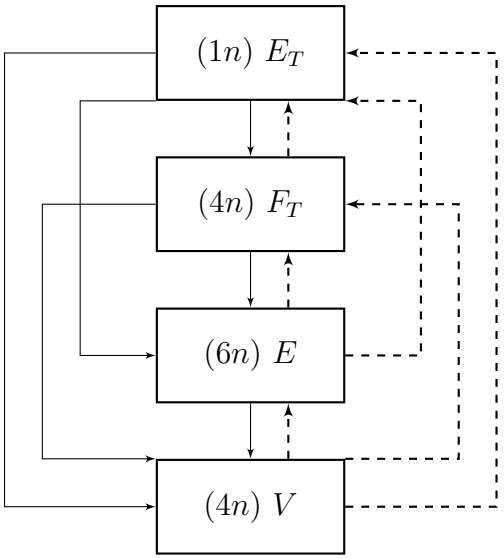
Jedną z bardziej popularnych metod jest metoda Zienkiewicza-Zhu [126]. Opiera się ona na istnieniu tzw. punktów nadzbieżności pochodnych rozwiązania MES w elementach. Zaproponowana przez nich metoda pozwala uzyskać przybliżenie błędu o rząd wielkości lepsze niż w normie L_2, przy spełnianiu założeń dotyczących istnienia punktów nadzbieżności pokrywających się często z punktami kwadratury całkowania Gaussa-Legandre'a. Zastosowanie jej w praktyce polega na stworzeniu tzw. „łat" (patches), czyli grup elementów stanowiących nośnik globalnych funkcji kształtu dla danego wierzchołka i w oparciu o wartości w punktach całkowania „odzyskaniu" gradientu rozwiązania z większą dokładnością.

2.3. Rodzaje elementów skończonych i siatka hybrydowa

W klasycznej metodzie elementów skończonych można znaleźć szereg różnych rodzajów elementów (każdy z różną ilością węzłów na krawędziach, bokach i we wnętrzu elementów) posiadających pewne wady i zalety, nadających się do różnych zastosowań. W przypadku, gdy w danej siatce występuje więcej niż jeden rodzaj elementu skończonego, to możemy mówić o siatce hybrydowej ze względu na rodzaj elementów. Rodzaje elementów skończonych mozna podzielić na pewne kategorie:

1D - element liniowy(prętowy) najczęściej wykorzystywany, gdy długość badanego obiektu znacząco przekracza jego przekrój, szczególnie gdy modelowanie uwzględnia zjawisko wygięcia pod wpływem obciążenia, wymiarem elementu skończonego jest jego długość,

2D - element płytowy,powierzchniowy: trójkątny, czworokątny są wykorzystywane do modelowania powłok, membran i innych obiektów, w których jeden wymiar jest wielokrotnie mniejszy niż dwa pozostałe, efektywniejszy obliczeniowo od modeli 3D i łatwiejszy w modelowaniu,



Rysunek 2.1: Reprezentacja elementu czworościennego jako obiektu siatki w relacji do części składowych.

3D - element objętościowy:czworościenny, szcześcienny, pryzmatyczny są to elementy objętościowe, które zasadniczo mogą służyć do modelowania różnych zjawisk, ale ze znaczącym kosztem obliczeniowym; ich złe uwarunkowanie może mieć negatywny wpływ na jakość wyników,

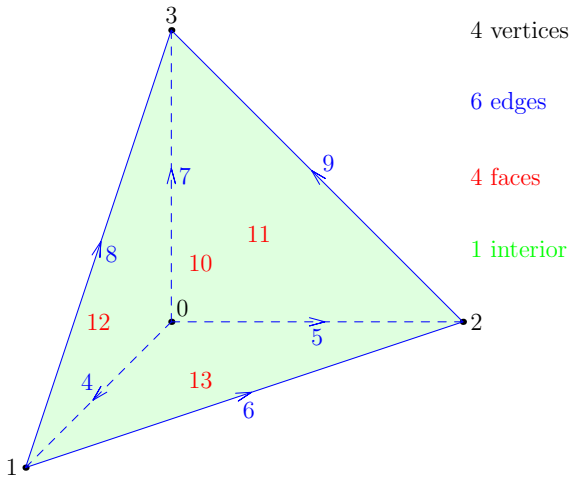
elementy subparametryczne wykorzystują interpolację geometryczną niższego rzędu względem funkcji interpolujących rozwiązanie,

elementy izoparametryczne używają tych samych funkcji, by określać zarówno rozwiązanie (wykorzystując stopnie swobody) i geometrię elementu (przy użyciu tzw. geometrycznych stopni swobody),

elementy superparametryczne wykorzystują interpolację geometryczną wyższego rzędu względem funkcji interpolujących rozwiązanie.

2.3.1. Czworościan jako podstawowy element siatki przestrzennej.

Czworościan, będący simpleksem przestrzeni trójwymiarowej, jest najczęstszym wyborem dla siatki metody elementów skończonych, który gwarantuje wypełnienie przestrzeni geometrycznej o dowolnym kształcie przy zachowaniu minimalnej ilości elementów zdegenerowanych (tzn. o bardzo złej jakości, najczęściej mierzonej relacją promienia koła wpisanego w element do opisanego na elemencie). Jest figurą prostą, składa się z czterech wierzchołków, sześciu krawędzi, czterech ścian oraz przestrzeni zamkniętej powyższymi ścianami. Na rys. 2.1 znajduje się przykład takiego elementu, wraz ze wskazaniem numeracji jego 4 wierzchołków, 6 krawędzi, 4 ścian oraz wnętrza. Ściany czworościanu są rozpięte na jego krawędziach, co do których oczywiste jest, iż można je wyznaczyć, posiadając same tylko współrzędne wierzchołków. Jest to bardzo ciekawa właściwość, której trudno szukać u innych fi-



Rysunek 2.2: Czworościan i tworzące go obiekty siatki. Pokazana jest przykładowa numeracja wierzchołków, krawędzi, ścian i wnętrza dająca w sumie 15 obiektów siatki składających się na 1 czworościan. Zaznaczone jest również skierowanie krawędzi będące zazwyczaj podstawą wyznaczania orientacji (skierowania) ścian.

gur (np.: sześcian), że czworościan można skonstruować tylko na podstawie czterech wierzchołków przestrzeni, bez posiadania jakichkolwiek innych informacji — i zawsze będzie taki sam. Ponadto czworościan foremny można umieścić w bryle sześcianu w taki sposób, że wierzchołki czworościanu i sześcianu się pokrywają, co stwarza ciekawe możliwości przechodzenia na siatki oparte na sześcianach, szczególnie jeżeli zwrócimy uwagę, że każdy sześcian można podzielić na czworościan foremny i cztery przystające czworościany prostokątne. Z punktu widzenia h—adaptacji ważna jest też właściwość, że czworościan daje się bez problemu dzielić na mniejsze czworościany i to na (nieskończenie) wiele różnych sposobów.

2.4. Adaptacja siatki obliczeniowej

Poprzez adaptację siatki obliczeniowej rozumie się zazwyczaj możliwość dynamicznego zmieniania rozmiaru elementu h oraz stopnia aproksymacji p w wybranych regionach obszaru obliczeniowego [121]. Pozwala to na zagęszczanie stopni swobody siatki MES w celu poprawy jakości aproksymacji w miejscach z dużym błędem. Zasadniczo wyróżnia się następujące rodzaje adaptacji:

h-adaptacja dąży do zmniejszania błędu poprzez zmianę rozmiaru h elementu skończonego osiąganą na drodze podziału na mniejsze elementy,

p-adaptacja dąży do zmniejszania błędu poprzez zmianę przestrzeni elementowych funkcji kształtu,

hp-adaptacja wykorzystuje techniki h-adaptacji i p-adaptacji jednocześnie

r-adaptacja dąży do zmniejszenia błędu poprzez zmianę położenia geometrycznego węzłów, bez zmieniania topologii siatki,

remeshing podobnie jak r-adaptacja, ale uwzględnia zmianę topologii siatki.

Przy czym r-adaptacja i remeshing jako jedyne nie zmieniają całkowitej liczby stopni swobody w siatce obliczeniowej. Dodatkowo można zastosować powyższe metody adaptacji w sposób anizotropowy, kiedy pewnych wybranych kierunkach dokonuje się poprawy siatki, a w innych nie.

2.4.1. H-adaptacja i podziały siatki.

W ogólności, z sytuacją siatki 1–nieregularnej spotykamy się w momencie, gdy w siatce elementów powstaje węzeł wiszący który, choć obecny w siatce, to nie posiada niezależnego globalnego stopnia swobody, tj. wartości stopni swobody w takim węźle są związane w celu zapewnienia ciągłości aproksymacji z wartościami sąsiadujących węzłów posiadających globalne stopnie swobody. Ze względu na tę zależność często określa się takie węzły jako *związane (constrained)*, a rodzaj aproksymacji uwzględniający takie węzły jako *constrained approximation*. Pojęcie 1–nieregularności odnosi się do przypadku, kiedy krawędź lub bok może posiadać tylko jeden wiszący (niezwiązany z globalnymi stopniami swobody) węzeł. Siatka 1–nieregularna powstaje w momencie, kiedy następuje nierównomierny podział siatki, tzn.: istnieje w siatce miejsce, gdzie element podzielony graniczy z elementem niepodzielonym tego samego rzędu. Warunek ciągłości rozwiązania wymusza, aby oba elementy posiadały na granicy takie samo rozwiązanie (takie same stopnie swobody). Jednak ich lokalne stopnie swobody się różnią, bo jeden jest podzielony i posiada ich więcej, a drugi nie jest podzielony i posiada stopni swobody mniej. Aby w takiej sytuacji, każdy z tych elementów posiadał takie samo rozwiązanie na wspólnej płaszczyźnie, którą się stykają, wiszący węzeł musi posiadać *związany stopnień swobody*.

Zakładając, że siatka ma być 1–nieregularna nakładamy na siatkę istotne ograniczenie. Co się dzieje w sytuacji, kiedy należy podzielić element, który jest już podzielony i jednocześnie jest 1–nieregularny? Wydaje się, iż warunek, który nałożyliśmy, bardzo mocno komplikuje rozwiązanie w tej konkretnej sytuacji. Rozważmy zatem taki przypadek. Aby zachować 1–nieregularność siatki, musimy najpierw sprawić, by element który chcemy podzielić, przestał być 1–nieregularny, a następnie wykonać normalną procedurę łamania elementu. Aby element przestał być 1–nieregularny, wszyscy jego sąsiedzi muszą być podzieleni w przynajmniej takim samym stopniu jak on sam. Zatem musimy wszystkich sąsiadów elementu, którzy tego wymagają, podzielić w normalny sposób. Jest to sytuacja, w której żądanie podziału jednego elementu pociąga za sobą podział następnych elementów, które mogą pociągnąć za sobą podział swoich sąsiadów itd. Jest to typowy przykład rekurencji: w czasie

realizowania żądania podziału elementu, wywołuje się procedurę dzielenia elementu sąsiedniego, który może wywołać procedurę dzielenia swojego elementu sąsiedniego itd. Stosując takie podejście do 1–nieregularności można zauważyć, że nie pozwala ono na lokalny, bardzo gwałtowny wzrost dokładności rozwiązania, niewspółmierny do otoczenia. 1– nieregularność wymusza na siatce, aby wzrost dokładności, czyli *de facto* podział elementu dokonywał się nie tylko „w głąb", ale także w sąsiedztwie elementu. Takie rozwiązanie stanowi zabezpieczenie ciągłości rozwiązania w przypadku nieregularności.

Rozdział 3

Modele reprezentacji siatki obliczeniowej

Zarówno ograniczenia ludzkiej percepcji, jak i ograniczenia urządzeń obliczeniowych, zmuszają do rozważania i rozwiązywania zagadnień o charakterze ciągłym w sposób dyskretny. Opierając się na teorii przedstawionej w poprzednich paragrafach, w celu znalezienia rozwiązania zagadnienia początkowo-brzegowego za pomocą MES, konieczna jest dyskretyzacja obszaru obliczeniowego. Zanim możliwe będzie dalsze omówienie siatki obliczeniowej jako istotnej części metody elementów skończonych, konieczne jest doprecyzowanie pewnych pojęć. Szereg poniższych określeń często jest używany w różnych kontekstach w dziedzinie elementów skończonych, jak i poza nią. W celu uniknięcia niedokładności, wieloznaczności lub nawet błędnego rozumienia tej części niniejszej pracy, istotne jest jednoznaczne zawężenie znaczenia używanych sformułowań. Poniższe definicje odnoszą się do siatki przestrzennej (trójwymiarowej) [105].

W klasycznym ujęciu [33] *siatka elementów skończonych* odnosi się do dyskretyzacji zawierającej informacje o:

1. geometrii obszaru obliczeniowego,
2. przestrzeni funkcji aproksymujących,
3. stopniach swobody (DOF).

Na potrzeby tej pracy, koncentrującej się na zarządzaniu siatką obliczeniową, pojęcia takie jak siatka, element etc. będą traktowane czysto geometrycznie i topologicznie, przy założeniu, że zarówno funkcje, jak i stopnie swobody – które zasadniczo nie są przedmiotem rozważań w tej pracy – mogą być z nimi w pewien określony sposób związane. W związku z tym analiza wpływu zmian w strukturze siatki pomija analizę cech związanych bezpośrednio z aproksymacją MES [112].

W klasycznej MES wyższego rzędu często wprowadza się dodatkowe węzły na krawędziach elementów skończonych, z którymi związane są zarówno tzw. geometryczne stopnie swobody, jak i stopnie swobody aproksymacji rozwiązania (DOF). Są to tzw. elementy izoparametryczne, które posiadają taką samą ilość węzłów, co stopni swobody. Powoduje to konieczność rozróżniania elementów skończonych o różnej ilości węzłów (np.: czworościan 4–węzłowy, czworościan 10–węzłowy itp.). Dla

uproszczenia rozważań przedstawione modele reprezentacji siatki zakładają elementy posiadające krawędzie prostoliniowe 2–węzłowe. Większa ilość węzłów na krawędź zmienia pewne stałe dotyczące efektywności przechowywanych struktur, natomiast nie ma wpływu na przyjęty schemat przechowywania obiektów siatki.

3.1. Pojęcia charakterystyczne dla opisu siatki

Obiekt (siatki) jest to dowolny topologiczny byt, będący częścią składową siatki jako całości, który jest obiektem geometrycznym i należy do geometrycznej przestrzeni obliczeniowej. Mogą być z nim związane stopnie swobody (ang. DOF - Degrees Of Freedom).

Łączność jest to właściwość obiektu siatki, określająca topologiczną relację do innych obiektów.

Topologia (siatki) jest to opis siatki niezależny od faktycznego geometrycznego kształtu obiektów w przestrzeni, przedstawiający tylko ich wzajemne sąsiedztwo.

Siatka (obliczeniowa) zbiór elementów skończonych zawierający ich reprezentację geometryczną i łączność topologiczną między nimi.

Geometria (siatki) określa kształty i wymiary oraz położenie przestrzenne obiektów w siatce.

Element (skończony) jest to *obiekt siatki*, będący odwzorowaniem elementu odniesienia należącego do przestrzeni 3D, zwierający ściany, krawędzie i wierzchołki.

Ściana (w siatce) jest to *obiekt siatki* będący odwzorowaniem elementu odniesienia należącego do przestrzeni dwuwymiarowej, topologicznie składający się z *krawędzi* i *wierzchołków*.

Krawędź (w siatce) jest to *obiekt siatki* odcinek, którego końce są *wierzchołkami* siatki. Składa się właśnie z tych dwóch wierzchołków.

Wierzchołek (w siatce) - *obiekt siatki* będący punktem w przestrzeni, należący do przynajmniej jednej *krawędzi* w siatce.

Kategoria(obiektów topologicznych) zbiór wszystkich obiektów siatki należących do tego samego wymiaru w sensie geometrycznym (e.g. 1D, 2D, 3D), czyli będących odwzorowaniem elementów odniesienia w danej przestrzeni.

Reprezentacja siatki określa dobór i schemat przechowywania obiektów siatki oraz ich wzajemnych łączności.

Siatka hybrydowa siatka obliczeniowa, w której jest używany więcej niż jeden element odniesienia dla elementów skończonych.

Siatka początkowa – niepodzielona siatka obliczeniowa, co do której zakłada się, że zawsze jest to siatka regularna.

Siatka rozproszona – siatka obliczeniowa podzielona na zachodzące (ang. overlapping) lub niezachodzące na siebie pod–obszary, które są przechowywane roz-

łącznie. W pracy analizowany jest przypadek siatek zachodzących na siebie. Jest to przypadek bardziej złożony, na podstawie którego można w prosty sposób przejść do przypadku siatek niezachodzących na siebie.

Ponieważ istnieje wiele różnych rodzajów elementów skończonych spełniających powyższą definicję, zostało przyjęte w tej pracy, że o ile nie zaznaczono inaczej, chodzi o trójwymiarowy czworościenny element skończony.

3.2. Podstawowy model siatki obliczeniowej.

Reprezentacja siatki obliczeniowej w programie, niezależnie od jej wewnętrznej struktury, musi być w stanie dostarczać informacji wymaganych do sformułowania i rozwiązania zadania aproksymacji. W przypadku adaptacyjnym siatka musi także dostarczać informacji i/lub być w stanie poprawnie przeprowadzać zagęszczanie i rozrzedzanie siatki. Ogólny model siatki obliczeniowej musi zatem zawierać informacje o obiektach, z którymi mogą być związane stopnie swobody: wierzchołkach, krawędziach, ścianach i elementach oraz ich wzajemnej łączności. Ogólny schemat możliwych połączeń między obiektami w siatce jest przedstawiony na rysunku 3.1, gdzie zaznaczono relacje agregacji i kompozycji poszczególnych obiektów wraz z krotnością tych relacji.

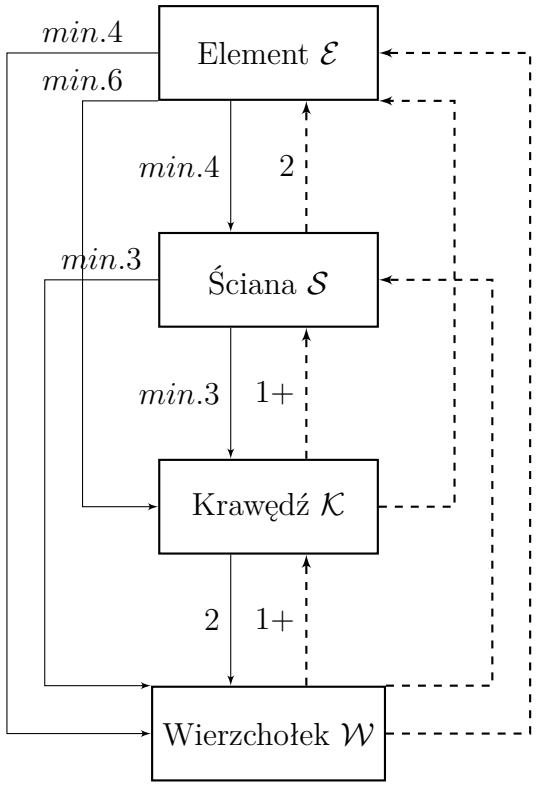
Sama struktura siatki może jawnie przechowywać wszystkie informacje o obiektach i ich łączności lub w czasie obliczeń odtwarzać żądane informacje na podstawie częściowej ich reprezentacji. Zatem w czasie tworzenia struktury siatki należy uwzględnić dwa czynniki decydujące o jej przydatności:

— zapotrzebowanie na pamięć, rozumiane jako średnia ilość pamięci w odniesieniu do przechowania jednego obiektu siatki
— efektywność, rozumiana jako złożoność obliczeniowa, potrzebna na uzyskanie od modułu zarządzania siatką żądanej informacji lub przeprowadzania wskazanej operacji

Oba te czynniki są silnie związane ze sposobem wykorzystywania siatki, w związku z czym, istnieje kilka sposobów reprezentowania siatki, które są odpowiednie do różnych zastosowań. Nawiązując do prac [48, 64, 24, 91, 105, 127, 107], można wskazać na najczęściej stosowane rodzaje reprezentacji siatki i dokonać ich klasyfikacji.

3.3. Model wydajnościowy i złożoność obliczeniowa siatki.

Analizując diagram 3.1, można dojść do wniosku o istotnej nadmiarowości uwzględnionych relacji, szczególnie gdy weźmie się pod uwagę, że pewne informacje i efektywność ich uzyskania mają znacznie większe znaczenie, a pewne mniejsze, ze względu



Rysunek 3.1: Diagram przedstawiający ogólny model siatki obliczeniowej regularnej. Liniami ciągłymi zaznaczona jest zależność agregacji ("jest zbudowany z"), a liniami przerywanymi zależność kompozycji ("zawiera się w"). Wartości przy liniach wskazują na krotność zależności.

na częstotliwość zapytań do modułu zarządzania siatką. Przykładem może tu być z jednej strony częste zapytanie o wierzchołki obiektu, konieczne by np.: obliczyć Jakobian transformacji do elementu odniesienia, a z drugiej np.: zapytanie o krawędzie obiektu, w celu sprawdzenia zgodności podziału w czasie dynamicznej adaptacji. W związku z tym można przyjąć, że miarą ogólnej efektywności siatki M z reprezentacją P, posiadającej n obiektów topologicznych, jest średnia ważona, w której złożoność obliczeniowa O_{op} operacji realizujących wymaganą funkcjonalność jest uwzględniana ze współczynnikiem f_{op} wskazującym na jej częstość :

$$O(M_P) = \sum_{op} O_{op}(n) \cdot f_{op} \tag{3.1}$$

Tak defniowana złożoność zależy nie tylko od algorytmów, ale także struktur danych. Uwzględniając wiedzę z zakresu obliczeń wysokiej wydajności, można dojść do wniosku, że odpowiednie rozłożenie danych w pamięci, może znacznie przyspieszyć wykonywanie częstych operacji – a co za tym idzie faktyczną wydajność modułu siatki i całego kodu obliczeniowego. Przy czym, samo rozlokowanie struktur danych w pamięci jest niezależne od ilości tych danych.

Złożoność obliczeniowa O_{op} dla struktur danych siatki jest tutaj analizowana pod kątem ilości operacji elementarnych związanych z łącznością w siatce. W celu takiej analizy każdy algorytm musi zostać przedstawiony w postaci podstawowych działań na strukturze danych, takich jak:

— wstawienie danej do struktury Put, dla którego $O_{op}(Put) = 1$, bez uwzględniania ew. rozszerzania globalnej struktury – zakłada się, że struktura wspiera tę operację ze złożonością O(1),
— pobranie danej ze struktury Get, dla którego $O_{op}(Get) = 1$, bez uwzględniania złożonego algorytmu przeszukiwania, tzn. kontener wykonuje operację pobrania danej pod wskazanym indeksem ze złożonością O(1),
— przypisanie wartości danej w strukturze Set, dla którego $O_{op}(Chk) = 1$, założenia podobne jak powyżej,
— porównanie dwóch danych Chk, dla którego $O_{op}(Chk) = 1$,
— znalezienie czy obiekt należy do zbioru N-elementowego Fnd, dla którego $O_{op}(Fnd) = N$.

Można zauważyć, że struktury danych wspierające takie operacje jak Put, Get, Set należą do tzw.: iteratorów, a w szczególności iteratora dostępu bezpośredniego (ang. random access iterator). Operator Chk i również Fnd wymaga, by obiekty były rozróżnialne, co w konsekwencji wymaga istnienia tzw. jednoznacznego identyfikatora. W szczególności należy zauważyć, że w środowisku z pamięcią rozproszoną koncepcja rozróżniania obiektów na poziomie globalnym jest niebanalnym zagadnieniem. Jednym z częstszych przykładów realizacji tej koncepcji jest istnienie tzw. globalnego jednoznacznego identyfikatora, który bywa realizowany za pomocą pary liczby określających lokalny identyfikator obiektu oraz pewien numer procesu, do którego należy. Problem ten jest omówiony szerzej w dalszej części pracy.

W celu spójnego i klarownego przedstawienia algorytmów wprowadźmy notację dla najczęstszych operacji dokonywanych na strukturze danych, gdzie wierzchołek, krawędź, ściana i element są oznaczane odpowiednio $\mathcal{W}$, $\mathcal{K}$, $\mathcal{S}$i $\mathcal{E}$. Poprzez oznaczenie $\{\mathcal{W}\}$, $\{\mathcal{W}_i, i = 1...6\}$ lub $\{\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3, \mathcal{W}_4, \mathcal{W}_5, \mathcal{W}_6\}$ rozumiemy zbiór wierzchołków – i analogicznie dla innych obiektów. W celu określenie relacji łączności topologicznej między obiektami w siatce będzie używany operator $(\cdot)$, np.: $\mathcal{W}(\mathcal{E})$ oznacza wierzchołek należący do elementu. W konsekwencji poprzez oznaczenie $\{\cdot(\cdot)\}$ będzie rozumiany zbiór obiektów spełniających dany operator, np.: dla czworościanu $\{\mathcal{W}(\mathcal{E})\}$ będzie tożsame ze zbiorem wszystkich wierzchołków należących do czworościanu $\mathcal{E} : \{\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3 \in \mathcal{E}\}$. Można zauważyć, że ilość operatorów określających łączność jest ograniczona i można ją przedstawić w formie tabelarycznej. Tabela 3.1 zawiera zestawienie operatorów łączności na topologicznych obiektach siatki, gdzie strzałki $\downarrow$ wskazują na operator łączności do obiektu należącego do mniejszej

przestrzeni (w sensie wymiarowości), a strzałki ↑ na operator łączności do obiektów należących do przestrzeni o większej ilości wymiarów.

W celu określenia sąsiedztwa dla danego obiektu topologicznego w postaci obiektów topologicznych tego samego rodzaju co on sam będzie używany zapis operatorowy gdzie operacja i jej argument należą do tej samej kategorii obiektów topologicznych np.: dla elementu E_1 zapis $\{\mathcal{E}(E_1)\}$ oznacza wszystkie elementy sąsiadujące bezpośrednio z elementem E_1.

	Wierzchołek	Krawędź	Ściana	Element
Elementy ( )	$\mathcal{W}(\mathcal{E})\downarrow$	$\mathcal{K}(\mathcal{E})\downarrow$	$\mathcal{S}(\mathcal{E})\downarrow$	$\mathcal{E}(\mathcal{E})$
Ściany ( )	$\mathcal{W}(\mathcal{S})\downarrow$	$\mathcal{K}(\mathcal{S})\downarrow$	$\mathcal{S}(\mathcal{S})$	$\mathcal{E}(\mathcal{S})\uparrow$
Krawędzie ( )	$\mathcal{W}(\mathcal{K})\downarrow$	$\mathcal{K}(\mathcal{K})$	$\mathcal{S}(\mathcal{K})\uparrow$	$\mathcal{E}(\mathcal{K})\uparrow$
Wierzchołki ( )	$\mathcal{W}(\mathcal{W})$	$\mathcal{K}(\mathcal{W})\uparrow$	$\mathcal{S}(\mathcal{W})\uparrow$	$\mathcal{E}(\mathcal{W})\uparrow$

Tabela 3.1: Zestawienie operatorów łączności na topologicznych obiektach siatki. Strzałki wskazują na to, czy operator określa łączność do większej (w górę) lub mniejszej (w dół) przestrzeni (w sensie wymiarowości).

W przypadku zapotrzebowania na pamięć można wprowadzić przynajmniej dwie miary. Pierwszą z nich będzie uśrednione zapotrzebowanie na pamięć potrzebną do przechowania jednego obiektu topologicznego siatki m_o. Drugą będzie ilość pamięci potrzebna do przechowania informacji o łączności między dwoma dowolnymi obiektami w siatce m_c. Zatem całkowite zapotrzebowanie na pamięć O_m dla siatki M_p, korzystającej z pewnej reprezentacji P, zawierającej n obiektów topologicznych i n_c informacji o łączności tych obiektów, dla której dane jest m_o, m_c można określić jako:

$$O_m(M_p) = n \cdot m_o + n_c \cdot m_c \tag{3.2}$$

Jednocześnie można podkreślić, że wartości m_o, m_c zależą od implementacji, w szczególności od formy i rodzaju przechowywanych danych.

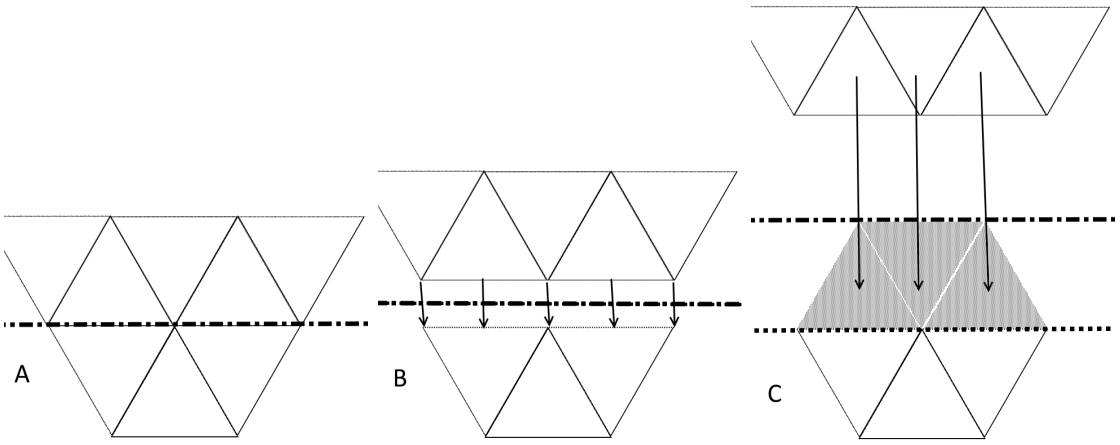
W zaproponowanym powyżej podstawowym modelu uwzględniającym zarówno złożoność obliczeniową, jak i wymagania pamięciowe reprezentacji siatki hybrydowej należy jeszcze uwzględnić narzut pamięciowy związany z duplikacją pewnych informacji oraz narzut złożoności obliczeniowej wynikający z konieczności komunikacji międzyprocesowej. Zagadnienia te zostały zanalizowane i omówione szerzej w rozdziale 4.

3.4. Siatka rozproszona i *ghost elements*.

Siatka rozproszona jest to siatka obliczeniowa podzielona na pod–obszary, w których rozwiązanie jest obliczane osobno, a następnie składane w jedną całość dla całego obszaru obliczeniowego. Najprostsze stosowane podejście zakłada replikację

całej siatki obliczeniowej w każdym pod-obszarze, z której wykorzystywany jest tylko pewien przypisany danemu obszarowi fragment. Rozwiązanie to nie skaluje się dobrze: wraz ze wzrostem ilości procesów, siatka zajmuje coraz więcej pamięci, przy jednoczesnym wykorzystaniu coraz mniejszej jej części.

Alternatywą jest forma przechowywania tylko lokalnego fragmentu siatki w pod -obszarze wraz z pewnymi elementami granicznymi pozwalającymi utrzymać spójność rozwiązania pomiędzy pod-obszarami. W takim przypadku pod-obszary są przechowywane rozłącznie. Przy czym z perspektywy każdego pod–obszaru można przyjąć, że stanowi on przy pewnych założeniach odrębny problem obliczeniowy od innych, pod warunkiem uzgodnienia rozwiązania na granicach pod-obszarów. Uzgadnianie rozwiązania na granicach podobszarów najczęściej jest realizowane na zasadzie duplikowania granicznych elementów, tworzących tzw. zakładkę (ang. overlap) w której znajdują się elementy posiadające rozwiązanie z innego pod–obszaru obliczeniowego tzw. *ghost elements* (rys. 3.2). Aby utrzymać spójność rozwiązania w takim przypadku każdy element, do którego należy wierzchołek, musi być w nakładce. Osobnym zagadnieniem jest zdefiniowanie głębokości takiej zakładki. Konieczne jest istnienie przynajmniej 1-elementowego zachodzenia na siebie pod-obszarów, ale mogą one mieć większą głębokość. Stosowane są również rozwiązania, w których przechowywany pod-obszar jest reprezentowany z pełną dokładnością, a reszta pod-obszarów jest w danym procesie reprezentowana przez siatkę mocno rozrzedzoną [22].



Rysunek 3.2: Podział na podobszary i dodawanie *ghost elements*. A: Przerywana linia określa granicę podziału na pod-obszary. B: Rozdzielenie powoduje zduplikowanie (zaznaczone strzałkami) obiektów siatki niższych wymiarów (w 3D: wierzchołków, krawędzi i ścian). C: 1-elementowa zakładka jest tworzona poprzez kopiowanie elementów sąsiednich (zaznaczone strzałkami) z innego pod-obszaru - *ghost elements* (w 3D elementy objętościowe). Ich kopiowanie wymaga skopiowania również wszystkich pozostałych obiektów składowych *ghost elements* nie istniejących jeszcze w docelowym pod-obszarze.

3.5. Klasyfikacja i rodzaje reprezentacji siatek.

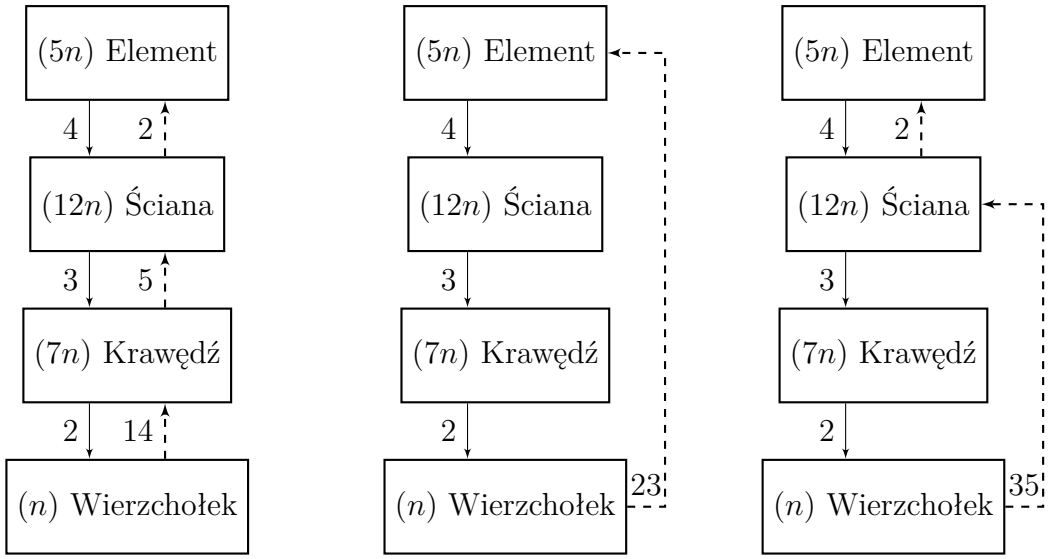
Reprezentacje siatki zawierające wszystkie rodzaje obiektów topologicznych dla każdego wymiaru są określane jako *pełne* reprezentacje. Zawierają one zawsze jakąś postać reprezentatywną dla wierzchołków, krawędzi, ścian i elementów (naturalnie dla trzech wymiarów; dla dwóch analogicznie). Nie ma jednak wymogu przechowywania wszystkich połączeń jawnie. Jednocześnie pełne reprezentacje siatki mogą przechowywać "mieszane" siatki, tj. siatki łączące obiekty topologiczne z różnych wymiarów, które nie stanowią jednorodnej siatki obliczeniowej, np. samodzielne ściany (trójkąty) niebędące częścią żadnego elementu przestrzennego itp. Zapotrzebowanie pamięciowe pełnej reprezentacji siatki zależy przede wszystkim od jawnie przechowywanych połączeń między obiektami, natomiast efektywność przede wszystkim zależy od wymagań obliczeniowych odtwarzania łączności nieprzechowywanych w siatce, a w drugiej kolejności od organizacji struktury danych oraz sposobu zarządzania pamięcią.

Jak to zostało już zaznaczone wcześniej, w zależności od rodzaju (lub rodzajów) elementu odniesienia oraz przyjętego przykładu siatki jako reprezentatywnego, w rozważaniach tego typu konkretne stałe liczby w wybranych reprezentacjach mogą się zmieniać. W celu uproszczenia i ujednorodnienia omawianych tu zagadnień za modelową siatkę przyjmujemy uniwersalną trójwymiarową siatkę czworościenną zajmującą pewien obszar w przestrzeni, co do której nie zakłada się regularnego kształtu ani jej wypukłości. Zakładamy natomiast, że stosunek powierzchni do objętości nie jest znany, ale że w miarę wzrastania rozmiaru siatki dąży do stosunku charakterystycznego dla kuli.

Z przedstawionej wcześniej analizy literatury oraz własnych badań, wynika, że pewne rodzaje reprezentacji siatki są opisywane i używane w praktycznych zastosowaniach. W celu odróżnienia ich od siebie można im przypisać symbole: $P_1, P_2, ...$ dla pełnych reprezentacji oraz $R_1, R_2, ...$ dla zredukowanych reprezentacji siatki. Poniżej znajdują się opisy poszczególnych ich typów.

3.5.1. Pełne reprezentacje siatki.

1. Reprezentacja P_1. Pełna jednopoziomowa dwukierunkowa łączność. Przedstawiona na diagramie 3.3 jest pełną reprezentacją, zawierającą wszystkie rodzaje obiektów topologicznych siatki i ich wzajemną łączność w zasięgu jednego poziomu (k–poziomowa łączność jest relacją pomiędzy obiektami topologicznymi różniącymi się wymiarem o k). Taka struktura może reprezentować siatki o różnych wymiarach (1D,2D,3D). Jak zostało to już przedstawione na rysunku 3.3, dla siatki czworościennej posiadającej n wierzchołków dobrym przybliżeniem ilości krawędzi, ścian i elementów będzie odpowiednio $7n$, $12n$ i $5n$. Zakładając,

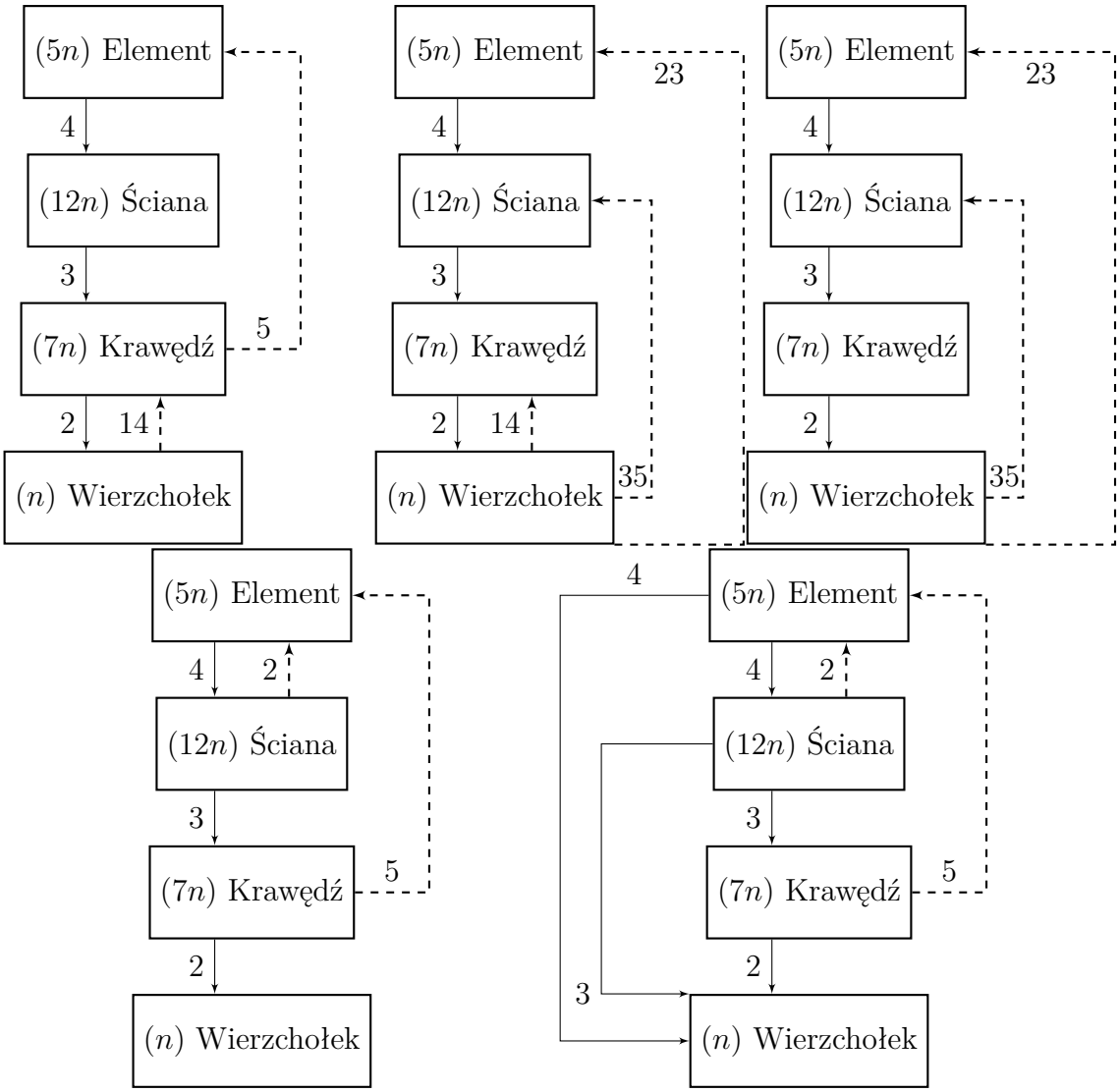


Rysunek 3.3: Pełna reprezentacja siatki typu P_1, P_2 i P_3. Statystyczna ilość obiektów podana w nawiasach obok nazwy. Obok linii statystyczna ilość połączeń między obiektami.

że średnio każdy obiekt potrzebuje m_o pamięci, całkowite zapotrzebowanie na pamięć przez wszystkie obiekty wynosi $(nm_o + 7nm_o + 12nm_o + 5nm_o) = 25nm_o$. Jednocześnie zakładając, że dla każdego połączenia potrzeba m_c pamięci , a dla każdego wierzchołka, których jest n, potrzeba przechować łączność do średnio 14 krawędzi $(14nm_c)$, oraz dla każdej krawędzi z $7n$ krawędzi siatki potrzeba przechować połączenie do 2 wierzchołków i 5 ścian, co daje w sumie $(12n)(2+5)m_c = 49nm_c$. Podobnie, połączenia ze ścian potrzebują $(12n)(3+2)m_c = 60nm_c$, a dla elementów $(5n)(4)m_c = 20nm_c$. Sumarycznie całkowite zapotrzebowanie na pamięć wynosi $(25m_o + 143m_c)n$.

2. Reprezentacja P_2. Jednopoziomowa łączność zstępna z łącznością wstępną wierzchołek –element. Ta pełna reprezentacja siatki przedstawiona jest na diagramie 3.3 jako P_2. Zawiera wszystkie obiekty topologiczne, ale jawnie przechowywane są tylko ich połączenia zstępujące wraz z jednym rodzajem łączności wstępującej wierzchołek–element. W celu uzyskania dostępu do łączności z obiektem wyższego wymiaru należy dokonać serii zstępujących operacji w sposób cykliczny, wykorzystując dodatkową relację wierzchołek–element. W praktyce oznacza, to że np. operacja $\mathcal{S}(\mathcal{K})$ sprowadza się do przeprowadzenia cyklu operacji $\mathcal{S}(\mathcal{K}) = \mathcal{S}(\mathcal{E}(\mathcal{W}(\mathcal{K})))$. Stąd też, można określić ten rodzaj reprezentacji jako reprezentacja cykliczna. Analizując wymagania pamięciowe: przechowywanie wszystkich rodzajów obiektów topologicznych to koszt $25nm_o$, natomiast zredukowanie reprezentacji łączności wymaga tylko $((4)(5) + 3(12) + 2(7) + 23)nm_c = 93nm_c$, skąd całkowite zapotrzebowanie na pamięć wynosi $(25m_o + 93m_c)n$.

3. Reprezentacja P_3. Jednopoziomowa łączność zstępna z łącznością wstępną ściana –element. Ten schemat jest podobny do schematu $P2$, z tym że cykliczność dotyczy tylko zależności wierzchołek–krawędź–ściana. Relacja element–ściana i ściana–element są trzymane jawnie, ze względu, na to, że szereg algorytmów siatek adaptacyjnych wymaga znajomości tej zależności. Wymagania pamięciowe są podobne co w przypadku P_2, z tym że należy uwzględnić dodatkowe połączenie wstępujące ściana–element, co w efekcie daje $(25m_o + 129m_c)n$.
4. Reprezentacja P_4. Jednopoziomowa łączność zstępna z łącznością wstępną dla relacji krawędź–element. Reprezentacja siatki P_4 stawia w centralnym punkcie obiekt reprezentujący topologiczne krawędzie. Koszt pamięciowy kształtuje się podobnie jak poprzednio, z tym że po uwzględnieniu dodatkowych połączeń wynosi $(25m_o + 119m_c)n$.
5. Reprezentacja P_5. Jednopoziomowa łączność zstępna z łącznością wstępną dla wierzchołków. Reprezentacja ta jest szczególnie nakierowana na wykorzystanie wierzchołków. Dzięki temu, że wierzchołek posiada jawnie przechowywane łączności do wszystkich innych obiektów topologicznych, koszt obliczeniowy pobierania tych informacji jest rzędu $O(1)$. Koszt przechowywania tej reprezentacji poza łącznościami zstępującymi $70nm_c$wymaga uwzględnienia wszystkich łączności wstępujących dla wierzchołka, co daje $72nm_c$, co ostatecznie daje $(25m_o + 142m_c)n$. Można zauważyć, że jest to wartość zbliżona do reprezentacji P_1. Przy porównywalnym koszcie przechowania, reprezentacja P_5 oferuje jednak lepszą efektywność w przypadku pobierania sąsiadujących obiektów i tworzeniu nowych.
6. Reprezentacja P_6. Jednopoziomowa łączność zstępna z częściową łącznością wstępną dla wierzchołków P_6. Zaprezentowana na rysunku 3.4 reprezentacja jest rodzajem zredukowanej reprezentacji $P5$. Jej wymagania pamięciowe wynoszą $(25m_o + 128m_c)n$.
7. Reprezentacja P_7. Pełna jednopoziomowa łączność zstępująca z łącznością wstępującą ściana–element i krawędź–element. W tej reprezentacji rolę dominującego punktu odniesienia przyjmuje element, dzięki wprowadzeniu dodatkowych łączności wstępujących. Koszt przechowania tej reprezentacji wynosi $25nm_o$ za łączności zstępujące oraz $(7)5m_c + 12(2)m_c$ za łączności wstępujące, co w efekcie daje $(25m_o + 59m_c)n$.
8. Reprezentacja P_8. Niepełna wielopoziomowa łączność P_8 jest bogatą reprezentacją siatki, w której każdy obiekt jawnie przechowuje wierzchołki, które go tworzą. Jest to poszerzona wersja reprezentacji P_7, zatem koszt przechowywania można będzie wynosił $(25m_o + 115m_c)n$.



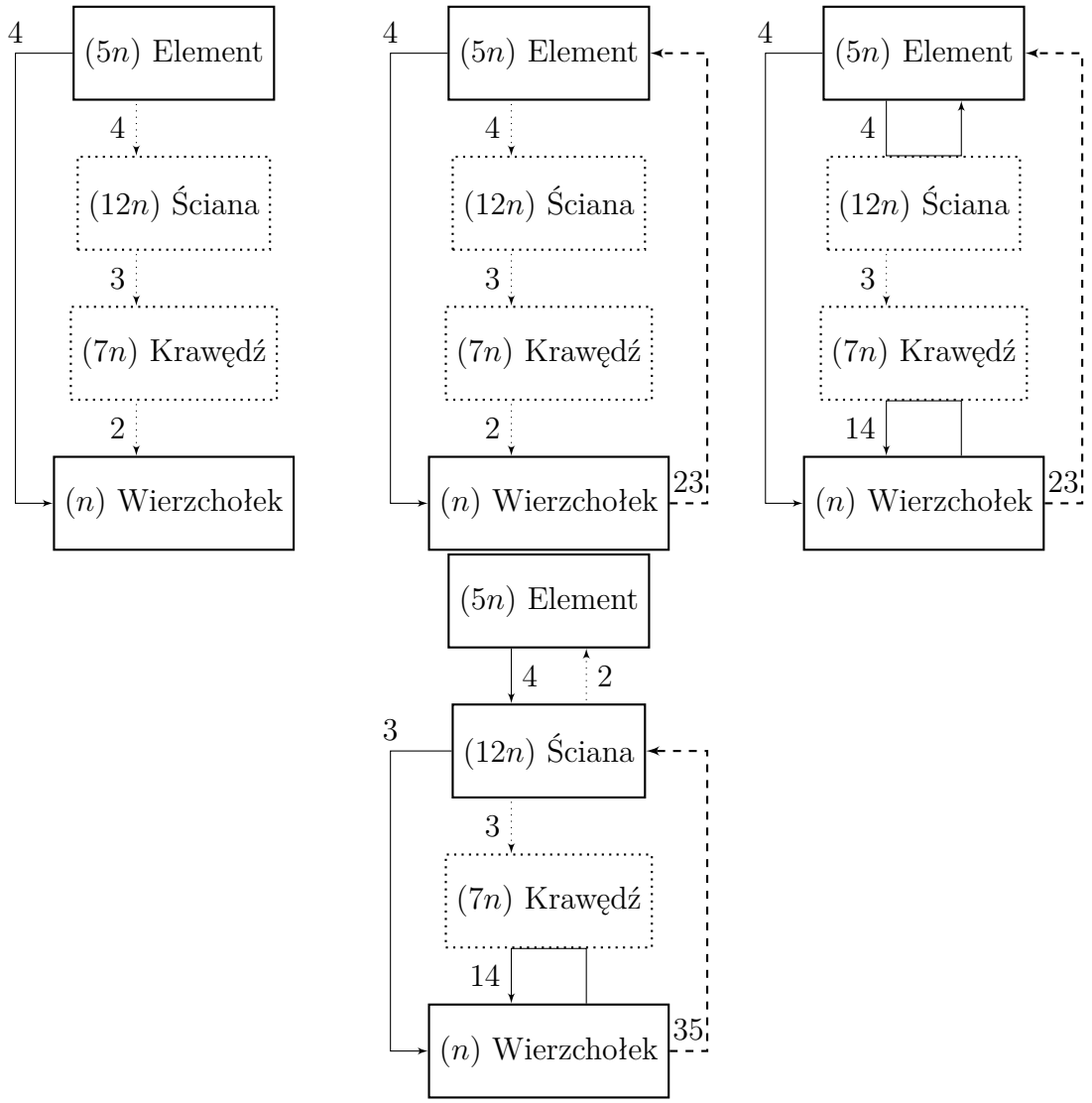
Rysunek 3.4: Pełna reprezentacja siatki typu P_4, P_5 i P_6 (górny rząd) oraz P_7 i P_8 (dolny rząd). Statystyczna ilość obiektów podana w nawiasach obok nazwy. Obok linii statystyczna ilość połączeń między obiektami.

3.5.2. Zredukowane reprezentacje siatki.

Struktura danych siatki, w której część obiektów ma charakter tymczasowy jest określana jako *zredukowana*. W takim przypadku struktura danych musi przechowywać przynajmniej obiekty najwyższego (element) i najniższego rzędu (wierzchołki). W zależności od aplikacji możliwe jest jednak pominięcie krawędzi lub ścian w jawnej, trwałej reprezentacji siatki. Obiekty te mogą nie być potrzebne w ogóle lub mogą być tworzone "w locie" na żądanie. Zredukowana reprezentacja siatki pozwala znacznie zmniejszyć wymagania pamięciowe, a przy odpowiedniej organizacji operacji na siatce, dodatkowy koszt obliczeniowy związany z odtwarzaniem struktur na żądanie może być niewielkim wymaganiem względem zysków pamięciowych. Ponadto, zredukowana reprezentacja siatki zwalnia z obowiązku synchronizowania nietrwałych

obiektów w siatce (np.: rozproszonej). Tymczasowe obiekty topologiczne powstające na bazie zsynchronizowanych struktur trwałych ograniczają ilość synchronizacji.

1. Zredukowana reprezentacja R_0. Zredukowana reprezentacja siatki zawierająca tylko łączność elementy–wierzchołki R_0 (przedstawiona na rysunku 3.5) jest to klasyczny zapis siatki, do dziś bardzo popularna reprezentacja w zapisie plikowym siatek. Pozwala na uzyskanie wszelkich połączeń w siatce (co prawda w sposób dość kosztowny, ale jednak) zachowując prostotę i małe wymagania pamięciowe.



Rysunek 3.5: Zredukowana reprezentacja siatki typu typu R_0, R_1 i R_2 (u góry) oraz R_4 (u dołu). Obiekty topologiczne zaznaczone linią kropkowaną nie są przechowywane w pamięci w sposób stały. Statystyczna ilość obiektów podana w nawiasach obok nazwy. Obok linii statystyczna ilość połączeń między obiektami.

2. Zredukowana reprezentacja R_1. Symetryczna łączność wierzchołek–element R_1 została przedstawiona na rys. 3.5. Jest to jedna bardziej popularnych repre-

zentacji zredukowanych, zawierająca elementy i wierzchołki oraz informacje o sąsiedztwie wierzchołek–element $\mathcal{W}(\mathcal{E})$ i element–wierzchołek $\mathcal{E}(\mathcal{W})$. Dzięki ograniczonemu przechowywaniu krawędzi i ścian można znacznie obniżyć pamięciowe koszty przechowywania struktury danych. Obiekty topologiczne nieprzechowywane w sposób trwały, muszą być tworzone "na żądanie" w czasie wykonywania algorytmów, które ich wymagają. W wielu opracowanych sposobach przechowywania siatki przyjmuje się pewne szablony, pozwalające na podstawie wierzchołków elementu odtwarzać krawędzie lub ściany. Istotą jest tutaj kryterium kolejności wierzchołków. Dla danego elementu czworościennego $\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3$, spełniającego założenia odnośnie do dodatniej objętości (np. iloczyn wektorów od $\mathcal{W}_0$ do $\mathcal{W}_1$ oraz $\mathcal{W}_0 do \mathcal{W}_3$ jest wektorem skierowanym do wewnątrz), ściany tego elementu $\{\mathcal{S}(\mathcal{E})\} = \{\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$ można przedstawić jako $\{\mathcal{S}(\mathcal{E})\} = \{\{\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_2\}, \{\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_3\}, \{\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3\}, \{\mathcal{W}_2, \mathcal{W}_0, \mathcal{W}_3\}\}$. Jednym z zagadnień wymagających szczególnej uwagi w przypadku obiektów tymczasowych jest ich zgodność z obiektami trwałymi w siatce, szczególnie w sensie ciągłości rozwiązania. Można zauważyć, że wykorzystanie szablonowego podejścia odtwarzania ścian dla dwóch sąsiednich elementów, może w efekcie doprowadzić do powstania dwóch reprezentacji tej samej ściany, różniących się tylko jej skierowaniem (rozumianym w sensie iloczynu wektorowego wektorów rozpinających). Taka sytuacja jest niedopuszczalna i prowadzi do błędnego rozwiązania. W celu jej uniknięcia można podjąć pewne działania, które zostały opisane m.in.: w [6]. Wymagania pamięciowe tej reprezentacji siatki dla (n wierzchołków i odpowiadającym im $5n$ elementom) wynosi $(6nm_o)$, a przechowanie łączności między nimi wymaga $(5n)(4)m_c + (n)(23)m_c = 43nm_c$, co w efekcie daje całkowite wymaganie rzędu $(6m_o + 43m_c)n$.

3. Zredukowana reprezentacja R_2. Jednopoziomowa łączność zstępna z łącznością wstępną wierzchołek–element R_2. Ta reprezentacja siatki jest jedną z wielu możliwych wariacji na temat reprezentacji R_1. Tylko wierzchołki i elementy są jawnie przechowywane w strukturze danych, wraz z ich wzajemną dwukierunkową łącznością. Jednocześnie przechowywane są pewne dodatkowe informacje, mające na celu bardziej efektywne realizowanie operacji związanych z ulotnymi obiektami w siatce. Elementy przechowują informację na temat innych sąsiadujących elementów (z którymi graniczyłyby poprzez ścianę, gdyby ta była przechowywana). Wierzchołki również przechowują informację nt. sąsiednich wierzchołków, tj. takich, z którymi dzielą hipotetyczną krawędź. Wymagania pamięciowe dla takiej struktury siatki są odpowiednio większe niż dla zwykłej reprezentacji R_1 $((6m_o + 43m_c)n)$ i są większe odpowiednio o 4 sąsiedztwa czworościanu oraz – średnio – 14 wierzchołków sąsiadujących dla każdego wierzchołka w siatce.

Stąd, całkowity koszt pamięciowy wynosi $(6m_o + 77m_c)n$. Reprezentacja R_2 jest przedstawiona na rysunku 3.5.

4. Zredukowana reprezentacja R_3. Jednopoziomowa reprezentacja obustronna R_3 bez krawędzi. W tej reprezentacji siatki przechowywane są wierzchołki, ściany i elementy. Motywacją przemawiającą za tą formą reprezentacji siatki jest fakt, iż przedstawienie ścian, jako obiektów trwałych wspiera wiele algorytmów (ściany często są używane do definiowania warunków brzegowych, łączności między siatkami itd.). Ponadto koszt poprawnego utworzenia tymczasowej ściany jest znacznie większy niż dla krawędzi. Elementy są tutaj definiowane za pomocą ścian, a ściany za pomocą wierzchołków. Wymagania pamięciowe dla tego danych rodzaju reprezentacji wynoszą $5nm_o + 12nm_o + nm_o = 18nm_o$, a dla łączności między obiektami $((5n)(4) + (12n)(3) + (n)(35) + (12n)(2))m_c = 115nm_c$.
5. Zredukowana reprezentacja R_4. Reprezentacja R_4 jest pewnym rodzajem reprezentacji R_3, w którym kwestia łączności została rozwiązana podobnie jak w reprezentacji R_2. Do przechowywanych połączeń analogicznie jak w R_3 dodano łączność między wierzchołkiem i jego sąsiadami–wierzchołkami, co zostało przedstawione na rysunku 3.5. Wymagania pamięciowe dla tego rodzaju reprezentacji wynoszą odpowiednio $5nm_o + 12nm_o = 18nm_o$ dla obiektów i $((5n)(4) + (12n)(3) + (n)(35) + (12n)(2) + (14n)(1))m_c = 129nm_c$ dla łączności miedzy nimi, co w efekcie daje $(18m_o + 129m_c)n$.

3.5.3. Hierarchia obiektów w siatce.

Przedstawiona powyżej klasyfikacja siatek odnosi się ogólnie do zarządzania siatką obliczeniową, niekoniecznie adaptacyjną. W przypadku siatki adaptacyjnej można mówić o pełnej lub zredukowanej reprezentacji w odniesieniu do hierarchii siatek (elementów) powstałej w wyniku adaptacji. W takim przypadku musi zostać uwzględniony w modelu następny czynnik odpowiadający za przechowywanie i odtwarzanie informacji o hierarchii podziałów. Można przyjąć, że choć odnosi się tak samo do przechowywania informacji o obiektach topologicznych, to łączności między tymi obiektami mają charakter wertykalny - w przeciwieństwie do horyzontalnych łączności w aktualnej siatce. Należy zauważyć, że wertykalne połączenia między obiektami topologicznymi w hierarchii siatki adaptacyjnej wymuszają przechowywanie (przynajmniej częściowe) obiektów i łączności, które de facto nie należą do aktualnej aktywnej siatki obliczeniowej. Powstaje tutaj problem: czy i jak przechowywać te – w pewnym sensie nieaktualne – informacje. Istnieją dwa najczęściej stosowane podejścia.

— Osobne przechowywanie informacji o hierarchii obiektów w siatce adaptacyjnej: obiekty niewykorzystywane aktualnie oraz łączności hierarchiczne są przecho-

wywane w osobnych strukturach danych (niejako poza siatką). Wadą takiego rozwiązania jest konieczność utrzymywania osobnych kontenerów z danymi lub dynamicznego alokowania dodatkowej pamięci, co prowadzi do wzrostu złożoności operacji i logiki kodu. Jednocześnie pozwala to ograniczyć przechowywanie informacji o hierarchii wertykalnej tylko do obiektów, które potrzebują je przechowywać (większość obiektów w siatce to obiekty końcowe drzewa hierarchii – liście drzewa – które nie podlegają dalszemu podziałowi).

— Informacje o hierarchii obiektów w siatce adaptacyjnej są traktowane jako integralna część składowa obiektów siatki (m.in.: podejście obiektowe wymusza tę konwencję). Zaletami jest większa prostota implementacji, przechowywanie wszystkich informacji o obiekcie w jednym miejscu oraz mniejsza ilość kontenerów (a czego wynika mniej alokacji i de-alokacji pamięci). Największą wadą tego podejścia jest rozmiar pojedynczego obiektu, który z założenia zawiera od razu przestrzeń na informacje dotyczące podziału hierarchicznego – która to może nie być nigdy wykorzystana.

3.5.4. Algorytmy siatki adaptacyjnej.

Niezależnie od wybranej reprezentacji, każdy mechanizm zarządzania siatką musi realizować pewne funkcje konieczne do poprawnego wspierania obliczeń z adaptacją. Algorytmy prezentowane poniżej stanowią abstrakcję funkcjonalności oferowanej przez różne interfejsy programowe i realizowanej za pomocą różnych struktur danych i adekwatnych dla nich operatorów $\mathcal{W}, \mathcal{K}, \mathcal{S}, \mathcal{E}$, Put, Get, Set, Chk, Fnd wspomnianych wcześniej.

Algorytm podziału elementu w siatce obliczeniowej

Pierwszym przedstawionym algorytmem jest algorytm 1 podziału pojedynczego elementu w siatce obliczeniowej. Ponieważ nie zakłada żadnych informacji specyficznych odnośnie do rodzaju tego elementu, można go określić jako algorytm uniwersalny lub odpowiedni dla siatki hybrydowej. Wykorzystane zostały pewne pojęcia takie jak generacji (poziomu) obiektu siatki Gen, których formalna definicja znajduje się w rozdziale 4. Zasadniczo składa się on z trzech etapów.

1. Pierwszy etap to sprawdzenie, czy podział elementu jest w ogóle legalny w sensie przyjętych założeń. Z technicznego punktu widzenia zawsze możemy podzielić element, ale to nie znaczy, że zawsze powinniśmy od razu robić, nawet gdy mamy informację o relatywnie dużym błędzie w danym miejscu. Jeżeli podział elementu naruszyłby np.: zasadę 1-nieregularności siatki obliczeniowej, to przeprowadzenie takiej operacji wprowadzi nieznany w skutkach błąd dotyczący jakości obliczanego rozwiązania. Stanie się tak, ponieważ złamiemy założone zasady zgodności rozwiązania pomiędzy elementami skończonymi (tzw. problem *conforming ap-*

proximation w czasie podziału nieregularnego). Dlatego też, zanim podzielimy element, wymuszamy spełniania przyjętych założeń, np.: poprzez podział elementów nie ze względu na zbyt duży błąd, ale ze względu na utrzymanie założenia 1-nieregularności.

2. Drugi etap to zapewnienie istnienia wszystkich obiektów siatki, do których będą się odnosić nowo-powstałe elementy potomne, w tym również tych, z których będą się bezpośrednio składać. W przypadku podziału nieregularnego dużym problemem okazuje się graniczenie ze sobą elementów o różnej wielkości i orientacji w przestrzeni. Otóż najczęściej występuje sytuacja, w której niektóre wymagane obiekty już istnieją, a niektóre jeszcze nie. Siatka obliczeniowa musi być w stanie udostępniać te informacje w krótkim czasie, bez żmudnego przeszukiwania struktur danych. Co więcej, jeżeli obiekty wymagane już istnieją, należy uwzględnić, że ich orientacja, czy uporządkowanie są już ustalone i nowy element musi dostosować się do nich, a nie jak w przypadku tworzenia nowych obiektów, kiedy to można kolejność konstruowania czy orientację wybrać sobie zgodnie z przyjętym schematem. I tutaj pojawia się kolejny problem, ponieważ, jeżeli nie zawsze jesteśmy w stanie wymusić jednorodny schemat kolejności czy orientacji elementów, to znaczy, że musimy również zacząć sprawdzać, czy któryś z naszych sąsiadów nie jest takim specyficznym przypadkiem, który wyłamał się z przyjętej konwencji w sposób wymuszony otoczeniem. Prowadzi nas to do kroku trzeciego.
3. Etap trzeci to faktyczne stworzenie nowych mniejszych elementów w miejsce elementu-rodzica oraz ustalenie i ustawienie poprawnych informacji dotyczących ich łączności. Etap ten jest bardzo podatny na błędy, ponieważ istnienie bardzo wiele scenariuszy, które należy przewidzieć i odpowiednio zrealizować, by otrzymać poprawną strukturę topologiczną siatki, szczególnie gdy rozwiązanie to ma być wydajne obliczeniowo. Przykładowo, uwzględniając najprostszy czworościenny element 3D, który posiadając tylko dwie odmiany orientacji w przestrzeni, daje się podzielić na 3 różne sposoby. Czyli w sumie mamy 6 różnych sposobów podziału jednego elementu. Jeżeli uwzględnimy, że sąsiaduje on z 4 innymi elementami, z których każdy również może dzielić się na 6 różnych sposobów, otrzymujemy $6^5 = 7776$ scenariuszy podziału uwzględniających orientację otoczenia. Dodatkowo należy uwzględnić, że sąsiedzi mogą być lub nie być podzieleni. W przypadku siatki hybrydowej należy tę ilość scenariuszy podziału pomnożyć odpowiednio dla możliwości sąsiadowania z elementami różnych rodzajów. Daje to zawrotne ilości kombinacji do rozpatrzenia, których z oczywistych przyczyn, wymuszają opracowanie rozwiązań automatycznych.

Algorytm 1: Algorytm adaptacji elementu siatki hybrydowej (3D)

Dane: element skończony E, łączności dla tego elementu I_E
sposób podziału P,
ilość obiektów potomnych n_P
Rezultat: elementy potomne $\{e_0, ..., e_{n_P}\}$
informacje o łączności obiektów $\{I_0, ..., I_{n_P}\}$

W przypadku siatki 1—nieregularnej należy uwzględnić i sprawdzić warunek 1—nieregularności: $Gen(e)$ = generacja (poziom) obiektu e ;
for *element e sąsiadujący (również pośrednio tj. przez wspólną krawędź) z E* **do**
  if $|Gen(E) - Gen(e)| \geqslant 1$ **then**
    Wykonaj rekurencyjnie adaptację obiektu e;
  end
end
Gdy warunek 1—nieregularności jest spełniony można wykonać adaptację:
for *ściany f należące do elementu E* **do**
  if *ściana f jest niepodzielona* **then**
    Podziel ścianę f;
    Zakłada się, że podział ściany pociąga za sobą także podział jej części składowych.
  end
end
for $n <= n_P$ **do**
  Stwórz nowy obiekt potomny e_n;
  Przypisz do obiektu e_n poprawne informacje o łączności horyzontalnej i wertykalnej I_{e_n} między obiektami;
end

Algorytm de-adaptacji elementu w siatce obliczeniowej

Drugi rozważany algorytm, to algorytm 2 teoretycznie odwrotny do poprzedniego, czyli algorytm de-adaptacji elementu skończonego trójwymiarowej siatki hybrydowej. Polega on na tym, że daną grupę mniejszych elementów (np.: wcześniej powstałych w wyniku podziału jednego elementu) zamienia na pojedynczy element skończony. Rozwiązanie tego zagadnienia, gdy siatka nie posiada informacji o hierarchicznej strukturze pionowej tj. o relacjach potomek-rodzic, jest w ogóle osobnym zagadnieniem, które nie jest tutaj rozważane. Zakładamy, że siatka posiada lub jest w stanie odtworzyć informację o takich relacjach. De-adaptacji używa się w miejscach, gdzie błąd jest mały, by zmniejszyć zagęszczenie stopni swobody i tym samym umożliwić po pierwsze zmniejszenie całkowitego rozmiaru zadania, a po dru-

gie zwiększenie ilości stopni swobody w miejscu, gdzie jest to konieczne ze względu na duży błąd. W tym algorytmie problematyczne zagadnienia nie tylko nie znikają, ale zmieniają swój charakter, przez co rozwiązania, które gwarantowały poprawne działanie podziału hierarchicznego, już nie wystarczają. Oznacza to, że siatka, która poprawnie przeprowadza adaptację, niekoniecznie jest w stanie przeprowadzić proces odwrotny. Analogicznie jak w pierwszym algorytmie podstawowe etapy wykonania można przedstawić w trzech krokach.

1. Etap pierwszy to sprawdzenie, czy i tym razem pożądana operacja jest legalna w ramach przyjętych założeń. Warunek ten daje się prosto wyrazić słowami, ale jego zapewnienie w ramach siatki obliczeniowej jest skomplikowanym procesem. Pierwszy problem tkwi w tym, że w ramach aproksymacji i topologii związanej, pewne wierzchołki nie mogą zostać tak po prostu usunięte, ponieważ zależą od nich inne wierzchołki, które w takim wypadku stałyby się niewiadomymi o nieznanej wartości rozwiązania. W praktyce wymagane jest sprawdzenie, czy żaden z obiektów siatki sąsiadujących z de-adaptowanym elementem nie posiada węzłów związanych. Przy odpowiednich założeniach można ten przypadek sprowadzić do sprawdzenia, czy elementy zawierające krawędzie de-adaptowanego elementu nie są elementami z węzłami zawieszonymi.
2. Etap drugi. Usuwanie niepotrzebnych obiektów składowych. Rozważmy najprostszy przypadek, jakim jest pojedynczy element czworościenny, podzielony uprzednio na mniejsze elementy potomne. Zauważmy, że dojście do tego etapu już wymagało od siatki udostępnienia informacji dot. rodzica mniejszych elementów. Otóż de-adaptacja tego elementu pociągnie za sobą pożądany proces usuwania obiektów siatki, które nie są już dłużej potrzebne. W szczególności chodzi o ściany, krawędzie i wierzchołki elementów potomnych, ale nie wszystkie: tylko te, które nie będą już dłużej używane. Otóż stwierdzenie, że dane obiekty nie są już nigdzie indziej używane, nie jest proste, ze względu na nieregularny rozkład elementów skończonych w przestrzeni. Oznacza to, że nie wystarczy sprawdzić, czy bezpośredni sąsiedzi nie korzystają z tych obiektów; należy przeszukać całe pobliskie otoczenie lub utrzymywać w siatce aktualny informacje o tym, który element używa którego obiektu, co jest bardzo niewydajne w sensie zajętości pamięci oraz konieczności ciągłego uaktualniania danych.
3. Etap trzeci. Upewniwszy się, że usunięte zostaną tylko obiekty niepotrzebne, można ostatecznie usunąć same elementy, które zostają zastąpione elementem - rodzicem.

Algorytm 2: Algorytm de-adaptacji elementu siatki hybrydowej (3D)

Dane: element skończony E, informacje o jego łączności I_E
elementy potomne $\{e_0, ..., e_{n_P}\}$
informacje o łączności obiektów $\{I_0, ..., I_{n_P}\}$

W przypadku siatki 1–nieregularnej należy uwzględnić i sprawdzić warunek 1–nieregularności: $Gen(E)$ = generacja (poziom) obiektu E ;
for *element e zawierający dowolną krawędź elementów* $\{e_0, ..., e_{n_P}\}$ **do**
  if $|Gen(E) - Gen(e)| \geqslant 1$ **then**
    Pomiń de–adaptację obiektu E;
  end
end
Gdy warunek 1–nieregularności jest spełniony można wykonać de–adaptację:
for *ściana* f *należąca do któregokolwiek z elementów* $\{e_0, ..., e_{n_P}\}$ **do**
  element e = sąsiad z drugiej strony f ;
  if $Gen(E) < Gen(e)$ **then**
    Dokonaj de–adaptacji ściany f;
    Zakłada się, że de–adaptacja ściany pociąga za sobą także de–adaptację jej części składowych.
  end
end
for $n <= n_P$ **do**
  Usuń obiekt potomny e_n;
end

Rozdział 4

Formalne sformułowanie problemu

4.1. Definicja formalna *zarządzania* adaptacyjną siatką obliczeniową

Opierając się na przedstawionym wcześniej w rozdziale 3 modelu siatki obliczeniowej, warto zauważyć, że koncepcja podstawowych *operatorów* pozwalających na definiowanie algorytmów w sposób niezależny od implementacji siatki, stwarza podstawę do precyzyjnego i formalnego zdefiniowania, czego oczekuje się od schematu *zarządzania* siatką obliczeniową. W konsekwencji pojawia się możliwość, aby wykorzystując zbiór operatorów, które udostępnia dana implementacja siatki, przypisać jej pewną klasyfikację, ze względu na to, jakie algorytmy mogą być dla tej siatki wykonywane, a jakie nie.

4.1.1. Operatory siatki jako podstawa realizacji algorytmów

Jak to zostało omówione w rozdziale 3.3, operatory stanowią dogodną bazę do definiowania algorytmów wykorzystujących siatki obliczeniowe. Jednakże w przypadku siatek adaptacyjnych, próba wykorzystania tylko tych operatorów do opisu algorytmów siatki adaptacyjnej, prowadzi do niejednoznaczności i sprzeczności. Jednocześnie 1–nieregularność siatki oraz sąsiadowanie ze sobą elementów należących do różnych generacji/poziomów siatki zmienia znaczenie pewnych operatorów. Ponadto uwzględniając nieregularną adaptację i de-adaptację siatki pojawia się konieczność zaistnienia nowych operatorów dotychczas niepotrzebnych. Wynika stąd, iż podstawowy zestaw operatorów łączności topologicznej $\mathcal{W}(\cdot), \mathcal{K}(\cdot), \mathcal{S}(\cdot), \mathcal{E}(\cdot)$ przedstawiony w tabeli 4.1 jest niewystarczający i wymaga rozszerzenia. Sama tabela 4.1 wskazuje tylko jakie operatory są możliwe do zrealizowania i ich wzajemną zależność w sensie wymiarowości. Dlatego też dla każdego z tych operatorów poniżej jest przedstawiona jego rozszerzona wersja.

Dla każdego z ww. operatorów można wskazać zbiór będący jego przeciwdziedziną, zawierający wszystkie obiekty danej kategorii topologicznej. Suma tych zbiorów

zawiera wszystkie obiekty topologiczne w danej siatce obliczeniowej. Przyjmując, że poprzez $dim(t)$ oznaczamy wymiarowość obiektu topologicznego siatki t oraz:

$$\begin{aligned} \boldsymbol{\mathcal{W}} &= \{w| \quad dim(w) = 0\} \\ \boldsymbol{\mathcal{K}} &= \{k| \quad dim(k) = 1\} \\ \boldsymbol{\mathcal{S}} &= \{s| \quad dim(s) = 2\} \\ \boldsymbol{\mathcal{E}} &= \{e| \quad dim(e) = 3\} \end{aligned} \tag{4.1}$$

można zdefiniować operatory siatki, z uwzględnieniem różnych kategorii elementów topologicznych:

$\mathcal{W}(E)$ **- wierzchołki elementu** E posiadającego n_e węzłów

$$\begin{aligned} \mathcal{W}_E: \quad & \boldsymbol{\mathcal{E}} \rightarrow \boldsymbol{\mathcal{W}}^{n_e} \\ & \mathcal{W}_E(E) = \{w | w \in \boldsymbol{\mathcal{W}} \wedge w \in E\} \end{aligned} \tag{4.2}$$

$\mathcal{W}(S)$ **- wierzchołki ściany** S posiadającej n_s węzłów

$$\begin{aligned} \mathcal{W}_S: \quad & \boldsymbol{\mathcal{S}} \rightarrow \boldsymbol{\mathcal{W}}^{n_s} \\ & \mathcal{W}_S(S) = \{w | w \in \boldsymbol{\mathcal{W}} \wedge w \in S\} \end{aligned} \tag{4.3}$$

$\mathcal{W}(K)$ **- wierzchołki krawędzi** K posiadającej n_k węzłów

$$\begin{aligned} \mathcal{W}_K: \quad & \boldsymbol{\mathcal{K}} \rightarrow \boldsymbol{\mathcal{W}}^{n_k} \\ & \mathcal{W}_K(K) = \{w | w \in \boldsymbol{\mathcal{W}} \wedge w \in K\} \end{aligned} \tag{4.4}$$

$\mathcal{W}(W)$ **- wierzchołki sąsiednie wierzchołka** W , gdzie sąsiedztwo oznacza istnienie wspólnych krawędzi

$$\begin{aligned} \mathcal{W}_W: \quad & \boldsymbol{\mathcal{W}} \rightarrow \boldsymbol{\mathcal{W}}^{n} \\ & \mathcal{W}_W(W) = \{w | w \in \boldsymbol{\mathcal{W}} \wedge \exists k \in \boldsymbol{\mathcal{K}} : \mathcal{W}(k) = \{w, W\}\} \end{aligned} \tag{4.5}$$

I dalej analogicznie dla krawędzi, ścian i elementów:

$\mathcal{K}(E)$ **- krawędzie elementu** E

$$\begin{aligned} \mathcal{K}_E: \quad & \boldsymbol{\mathcal{E}} \rightarrow \boldsymbol{\mathcal{K}}^{n_e} \\ & \mathcal{K}_E(E) = \{k | k \in \boldsymbol{\mathcal{K}} \wedge k \in E\} \end{aligned} \tag{4.6}$$

$\mathcal{K}(S)$ **- krawędzie ściany** S

$$\begin{aligned} \mathcal{K}_S: \quad & \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{K}}^{n_s} \\ & \mathcal{K}_S(S) = \{k | k \in \boldsymbol{\mathcal{K}} \wedge k \in S\} \end{aligned} \tag{4.7}$$

$\mathcal{K}(K)$ **- krawędzie sąsiednie krawędzi** K , gdzie sąsiedztwo oznacza istnienie dokładnie jednego wspólnego wierzchołku

$$\begin{aligned} \mathcal{K}_K: \quad & \boldsymbol{\mathcal{K}} \to \boldsymbol{\mathcal{K}}^{n} \\ & \mathcal{K}_K(K) = \{k | \, |\mathcal{W}(K) \cap \mathcal{W}(k)| = 1\} \end{aligned} \tag{4.8}$$

$\mathcal{K}(W)$ **- krawędzie zawierające wierzchołek** W

$$\begin{aligned} \mathcal{K}_W: \quad & \boldsymbol{\mathcal{W}} \to \boldsymbol{\mathcal{K}}^{n} \\ & \mathcal{K}_W(W) = \{k | \mathcal{W}(k) \ni W\} \end{aligned} \tag{4.9}$$

$\mathcal{S}(E)$ **- ściany elementu** E

$$\begin{aligned} \mathcal{S}_E: \quad & \boldsymbol{\mathcal{E}} \to \boldsymbol{\mathcal{S}}^{n_e} \\ & \mathcal{S}_E(E) = \{s | s \in \boldsymbol{\mathcal{S}} \wedge s \in E\} \end{aligned} \tag{4.10}$$

$\mathcal{S}(S)$ **- ściany sąsiednie ściany** S gdzie sąsiedztwo oznacza istnienie jednej wspólnej krawędzi

$$\begin{aligned} \mathcal{S}_S: \quad & \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{S}}^{n} \\ & \mathcal{S}_S(S) = \{s | s \in \boldsymbol{\mathcal{S}} \wedge \exists k \in \boldsymbol{\mathcal{K}} : |\mathcal{K}(S) \cap \mathcal{K}(s)| = 1\} \end{aligned} \tag{4.11}$$

$\mathcal{S}(K)$ **- ściany zawierające krawędź** K

$$\begin{aligned} \mathcal{S}_K: \quad & \boldsymbol{\mathcal{K}} \to \boldsymbol{\mathcal{S}}^{n} \\ & \mathcal{S}_K(K) = \{s | \mathcal{K}(s) \ni K\} \end{aligned} \tag{4.12}$$

$\mathcal{S}(W)$ **- ściany zawierające wierzchołek** W

$$\begin{aligned} \mathcal{S}_W: \quad & \boldsymbol{\mathcal{W}} \to \boldsymbol{\mathcal{S}}^{n} \\ & \mathcal{S}_W(W) = \{s | \mathcal{W}(s) \ni W\} \end{aligned} \tag{4.13}$$

$\mathcal{E}(E)$ **- elementy sąsiednie elementu** E , gdzie sąsiedztwo oznacza istnienie wspólnej ściany

$$\begin{aligned} \mathcal{E}_E: \quad & \boldsymbol{\mathcal{E}} \to \boldsymbol{\mathcal{E}}^{|\mathcal{S}(E)|} \\ & \mathcal{E}_E(E) = \{e | \, |\mathcal{S}(E) \cap \mathcal{S}(e)| = 1\} \end{aligned} \tag{4.14}$$

$\mathcal{E}(S)$ **- elementy zawierające ścianę** S

$$\begin{aligned} \mathcal{E}_S: \quad & \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{E}}^2 \\ & \mathcal{E}_S(S) = \{e | \mathcal{S}(e) \ni S\} \end{aligned} \tag{4.15}$$

$\mathcal{E}(K)$ **- elementy zawierające krawędź** K

$$\begin{aligned} \mathcal{E}_K: \quad & \boldsymbol{\mathcal{K}} \to \boldsymbol{\mathcal{E}}^n \\ & \mathcal{E}_K(K) = \{e | \mathcal{K}(e) \ni K\} \end{aligned} \tag{4.16}$$

$\mathcal{E}(W)$ **- elementy zawierające wierzchołek** W

$$\begin{aligned} \mathcal{E}_W: \quad & \boldsymbol{\mathcal{W}} \to \boldsymbol{\mathcal{E}}^n \\ & \mathcal{E}_W(W) = \{e | \mathcal{W}(e) \ni W\} \end{aligned} \tag{4.17}$$

	Wierzchołki ( )	Krawędzie ( )	Ściany ( )	Elementy ( )
$E \in \boldsymbol{\mathcal{E}}$	$\mathcal{W}(E)\downarrow$	$\mathcal{K}(E)\downarrow$	$\mathcal{S}(E)\downarrow$	$\mathcal{E}(E)$
$S \in \boldsymbol{\mathcal{S}}$	$\mathcal{W}(S)\downarrow$	$\mathcal{K}(S)\downarrow$	$\mathcal{S}(S)$	$\mathcal{E}(S)\uparrow$
$K \in \boldsymbol{\mathcal{K}}$	$\mathcal{W}(K)\downarrow$	$\mathcal{K}(K)$	$\mathcal{S}(K)\uparrow$	$\mathcal{E}(K)\uparrow$
$W \in \boldsymbol{\mathcal{W}}$	$\mathcal{W}(W)$	$\mathcal{K}(W)\uparrow$	$\mathcal{S}(W)\uparrow$	$\mathcal{E}(W)\uparrow$

Tabela 4.1: Operatory na topologicznych obiektach siatki bez uwzględnienia niejednorodnej adaptacji i de-adaptacji. Gdzie E, S, K, W należą odpowiednio do zbioru elementów, ścian, krawędzi i wierzchołków. Strzałki wskazują na to, czy operator określa łączność do większej (w górę) lub mniejszej (w dół) przestrzeni (w sensie wymiarowości)

.

Wielokrotnie użyta w 4.2-4.17 relacja przynależności *a należy do A* ($a \in A$) w sensie topologicznym (nie w znaczeniu geometrycznym) jest dobrze sformułowana i jednoznaczna. Jednakże użycie jej w przypadku siatki adaptacyjnej z 1–nieregularnością by określać sąsiadujące elementy, ściany i krawędzie, należące do różnych generacji, powoduje niespełnianie warunku topologicznego sąsiedztwa, mimo iż w sposób geometryczny obiekty te ze sobą graniczą. Co więcej, dotychczasowe operatory nie pozwalają też rozróżnić, do jakiej generacji należą rozważane fragmenty siatki. Wy-

nika stąd, iż operatory te wymagają zmiany albo konieczne jest rozszerzenie zestawu operatorów.

W przypadku architektur rozproszonych i dekompozycji domeny, wymagającej podziału całego obszaru obliczeniowego na pod-obszary, zmiany wymaga zdefiniowanie kompletnych zbiorów elementów $\boldsymbol{\mathcal{E}}$, ścian $\boldsymbol{\mathcal{S}}$, krawędzi $\boldsymbol{\mathcal{K}}$ i wierzchołków $\boldsymbol{\mathcal{W}}$. O ile w przypadku jedno-procesowym, definicja tych zbiorów w sensie topologicznym jest prosta, gdyż należą do nich wszystkie odpowiadające im istniejące instancje danego topologicznego fragmentu siatki, o tyle w analizowanym przypadku rozproszonej siatki obliczeniowej z tzw. *ghost elements* tworzącymi zachodzące na siebie pod-obszary(*e.g 1–element overlap*) musi ona uwzględnić zarówno duplikowanie pewnych fragmentów siatki w sensie globalnym, jak i niekompletność tejże w sensie lokalnym. Zatem, aby operatory były dobrze zdefiniowane, wymagana jest zmiana definicji tych zbiorów. Wraz z tą zmianą dalszej modyfikacji musi ulec definicja sąsiedztwa, ponieważ pojawia się nowy rodzaj sąsiadowania poprzez granicę pod-obszaru (ang. subdomain boundary).

Konsekwencją powyższych spostrzeżeń jest rozszerzony i zmodyfikowany zbiór operatorów, który jest dobrze zdefiniowany dla problemów adaptacyjnych i uwzględnia specyfikę rozproszonych architektur obliczeniowych.

4.1.2. Zbiór operatorów dla rozproszonej adaptacyjnej siatki obliczeniowej

Aby zestaw operatorów przedstawiony w tabeli 4.1 umożliwiał faktyczne przeprowadzania obliczeń na siatce, konieczny jest operator, który na podstawie informacji topologicznej udostępni informacje o geometrycznych własnościach obiektu topologicznego. Możemy zatem wprowadzić operator udostępniający położenie geometryczne obiektu $\mathcal{G}_{eo}(\cdot)$ oraz jego rozmiar charakterystyczny $h(\cdot)$:

$$\begin{aligned} &\forall t \in T \qquad \mathcal{G}_{eo}(t) : t \to \mathbb{R}^3 \\ &\forall t \in T \qquad h(t) : t \to \mathbb{R} \end{aligned} \tag{4.18}$$

Operatory siatki adaptacyjnej

Aby rozróżnić grupy obiektów topologicznych należących do tej samej kategorii, ale do różnych generacji siatki adaptacyjnej, można wprowadzić dodatkowy operator określający *generację* obiektu topologicznego t należącego do zbioru obiektów tej samej kategorii T:

$$\mathcal{G}(t) = n \quad \Leftrightarrow \quad \exists T_p \subset T \wedge T_p = \{t_p | t \subset Int\, t_p\} \wedge |T_p| = n, \quad \forall t \in T$$

gdzie $t \subset Int\, t_p$ oznacza geometryczną przynależność fragmentu siatki do wnętrza innego fragmentu siatki będącego w tej samej kategorii topologicznej. Należy zauważyć, że operator $\mathcal{G}(\cdot)$ wprowadza poważną nieścisłość, ponieważ dla wierzchołków siatki zawsze będzie zwracał generację 0, ze względu na to, że nowe wierzchołki nie są częścią wcześniej istniejących wierzchołków. Można uzupełnić definicję o dodatkowe rozróżnienie:

$$\mathcal{G}(t) = n \Leftrightarrow \begin{cases} t \in \{\mathcal{W}\}: & \exists e_n \in \mathcal{E}\{t\} : \mathcal{G}(e_n) = n \wedge \forall e \in \mathcal{E}\{t\}\, \mathcal{G}(e_n) \geqslant \mathcal{G}(e) \\ t \notin \{\mathcal{W}\}: & \exists T_r \subset T \wedge T_r = \{t_r | Int\, t_r \supset t\} \wedge |T_r| = n \end{cases} \tag{4.19}$$

Powyższa definicja jest spójna i nie wymaga informacji związanych z historią adaptacji i de-adaptacji przeprowadzanych siatce. Uwzględniając istnienie takiej historii, można posłużyć się prostszymi sformułowaniami, określającymi czym jest generacja obiektu w siatce, ale takie podejście wymusza założenie o dostępności takich informacji.

Przeprowadzanie adaptacji i de-adaptacji wymaga nie tylko rozróżniania generacji, ale także wprowadzenia relacji rodzic-potomek pomiędzy obiektami siatki. Pierwszym, prostszym jej rodzajem jest sytuacja mająca miejsce, gdy obiekty należą do tej samej kategorii topologicznej. Można określić operatory pozwalające wskazać zbiór wszystkich potomków $\mathcal{P}'(\cdot)$ i rodziców $\mathcal{R}'(\cdot)$, niezależnie od ich generacji, co w efekcie daje całą rodzinę obiektu siatki $\mathcal{F}'(\cdot)$:

$$\begin{aligned} \mathcal{P}'(t) =& \{t_p | t_p \subset Int\, t\} \\ \mathcal{R}'(t) =& \{t_r | t \subset Int\, t_r\} \\ \mathcal{F}'(t) =& \{t_r | t_r \in \mathcal{P}' \vee t_r \in \mathcal{R}'\} \end{aligned} \tag{4.20}$$

Następnie można zdefiniować operatory bezpośredniego potomka $\mathcal{P}(\cdot)$, rodzica $\mathcal{R}(\cdot)$ i rodziny $\mathcal{F}(\cdot)$ w następujący sposób:

$$\begin{aligned} \mathcal{P}(t) = t_p \quad &\Leftrightarrow \quad \mathcal{G}(t_p) = \mathcal{G}(t) - 1 \wedge t_p \in \mathcal{P}' \\ \mathcal{R}(t) = t_r \quad &\Leftrightarrow \quad \mathcal{G}(t_r) = \mathcal{G}(t) + 1 \wedge t_r \in \mathcal{R}' \\ \mathcal{R}(t) = t_r \quad &\Rightarrow \quad t \in \{\mathcal{P}(t_r)\} \\ \mathcal{P}(t) = t_p \quad &\Rightarrow \quad \mathcal{R}(t_p) = t \\ \mathcal{F}(t) =& \{t_r | t_r \in \mathcal{P} \vee t_r \in \mathcal{R}\} \end{aligned} \tag{4.21}$$

Na tej podstawie można ostatecznie zdefiniować operatory adaptacji i de-adaptacji dla siatki 1–nieregularnej. Dla danego obiektu t schemat adaptacji $\mathcal{A}_i$ na i elementów potomnych i analogiczny schemat de-adaptacji $\mathcal{D}_i$ z wykorzystaniem i potomków :

$$\begin{aligned}
\mathcal{A}_i: &\quad T \to T^i \\
\mathcal{A}_i(t) =&\{p_t | p_t = \mathcal{P}(t) \wedge \bigcup_{t=1}^{i} Int\, p_t = t \wedge \bigcap_{t=1}^{i} Int\, p_t = \emptyset\} \qquad t, p_t \in T \\
\mathcal{D}_i: &\quad T^i \to T \\
\mathcal{D}_i(t_1, .., t_i) =&t \Leftrightarrow t = \mathcal{R}(t_1, ..., t_i) \wedge Int\, t = \bigcup_{t=1}^{i} Int\, t_t \qquad t, t_1, ..., t_i \in T
\end{aligned} \tag{4.22}$$

Operatory siatki rozproszonej

Moduł siatki umożliwiający obliczenia w P pod-obszarach (gdzie $P \in \mathbb{N}$) musi posiadać rozróżnienie na *lokalny* zbiór topologicznych obiektów siatki $T_i, i \in \mathbb{N}^+$ nad którym zostanie obliczone rozwiązanie lokalne i jego *globalny* odpowiednik T:

$$T = \bigcup_{p=0}^{P} T_p$$

W lokalnym zbiorze topologicznym T_p można wyróżnić

— część należącą ekskluzywnie do danego pod-obszaru $\mathring{T}_p$
— i część współdzieloną z innymi obszarami $\bar{T}_p$, czyli pewien rodzaj domknięcia powyższego obszaru. .

Istnienie części brzegowej zapewnia ciągłość domeny obliczeniowej w całym obszarze, lecz sprawia, że moc zbiorów obiektów rozproszonych w pod-obszarach jest większa niż moc ich sumy. Różnica ta implikuje narzut obliczeniowy związany z wykorzystaniem środowisk rozproszonych. Zatem zbiór wszystkich obiektów topologicznych tej samej kategorii jest sumą zbiorów tych obiektów w poszczególnych P pod-obszarach:

$$\begin{aligned}
\mathcal{E} &= \bigcup_{p=0}^{P} \bar{\mathcal{E}}_p \cup \mathring{\mathcal{E}}_p \qquad |\mathcal{E}| < \sum_{p=0}^{P} |\bar{\mathcal{E}}_p| + |\mathring{\mathcal{E}}_p| \\
\mathcal{S} &= \bigcup_{p=0}^{P} \bar{\mathcal{S}}_p \cup \mathring{\mathcal{S}}_p \qquad |\mathcal{S}| < \sum_{p=0}^{P} |\bar{\mathcal{S}}_p| + |\mathring{\mathcal{S}}_p| \\
\mathcal{K} &= \bigcup_{p=0}^{P} \bar{\mathcal{K}}_p \cup \mathring{\mathcal{K}}_p \qquad |\mathcal{K}| < \sum_{p=0}^{P} |\bar{\mathcal{K}}_p| + |\mathring{\mathcal{K}}_p| \\
\mathcal{W} &= \bigcup_{p=0}^{P} \bar{\mathcal{W}}_p \cup \mathring{\mathcal{W}}_p \qquad |\mathcal{W}| < \sum_{p=0}^{P} |\bar{\mathcal{W}}_p| + |\mathring{\mathcal{W}}_p|
\end{aligned} \tag{4.23}$$

Niestety należy zauważyć, iż dla dowolnej kategorii T:

$$|\bigcup_{p=0}^{P} \mathring{T}_i| < |T| \tag{4.24}$$

Z 4.23 i 4.24 wynika, że całkowity zbiór obiektów zapewniających ciągłość rozwiązania $\bar{T}$ nie jest dotychczas znany. Następstwem tego jest konieczność przeszukiwania wszystkich pod-obszarów T_p w celu znalezienia i wymiany informacji pozwalających utrzymać ciągłość rozwiązania. Jak to było zaznaczone w rozdziale 1.4, by rozwiązać ten problem, wprowadzano modyfikację polegającą na odgórnym przypisaniu obiektów granicznych do wybranych pod-obszarów zgodnie z przyjętym kluczem. Jeden z pod-obszarów posiadających obiekt graniczny stawał się właścicielem obiektu (ang. owner), gdy spełniony był pewien arbitralny warunek (np.: związany z numeracją pod-obszarów. Można zatem wprowadzić dla siatki rozproszonej operator własności $\mathcal{O}$ (ang. ownership) jednoznacznie przypisujący obiekt topologiczny do pod-obszaru. W konsekwencji obszar graniczny pod-obszaru p dla obiektów topologicznych klasy T można zdefiniować jako:

$$\begin{aligned} \bar{T}_p =& \bar{T}_p^E \cup \bar{T}_p^I \\ \bar{T}_p^I =& \{t_p | t_p \in \bar{T}_p \wedge \mathcal{O}(t_p)\} \\ \bar{T}_p^E =& \{t_p | t_p \in \bar{T}_p \wedge t_p \notin \bar{T}_p^I\} \end{aligned} \tag{4.25}$$

Gdzie $\bar{T}_p^I$ określa granicę wewnętrzną, a $\bar{T}_p^E$ granicę zewnętrzną pod-obszaru obliczeniowego. Z definicji tej wynika, iż pod-obszar T_p w sensie topologicznym składa się teraz z trzech rozłączonych podzbiorów określających odpowiednio jego wnętrze $\mathring{T}_p$, wewnętrzny obszar graniczny $\bar{T}_p^I$, dla którego spełniony jest warunek własności $\mathcal{O}$, oraz zewnętrzny obszar graniczny $\bar{T}_p^E$ dla którego nie jest on spełniony:

$$T_p = \mathring{T} \cup \bar{T}_p^I \cup \bar{T}_p^E \tag{4.26}$$

Z 4.23 i 4.25 wynika, że:

$$\begin{aligned} \mathcal{E} &= \bigcup_{p=0}^{P} \bar{\mathcal{E}}_p^I \cup \mathring{\mathcal{E}}_p \qquad |\mathcal{E}| = \sum_{p=0}^{P} |\bar{\mathcal{E}}_p^I| + |\mathring{\mathcal{E}}_p| \\ \mathcal{S} &= \bigcup_{p=0}^{P} \bar{\mathcal{S}}_p^I \cup \mathring{\mathcal{S}}_p \qquad |\mathcal{S}| = \sum_{p=0}^{P} |\bar{\mathcal{S}}_p^I| + |\mathring{\mathcal{S}}_p| \\ \mathcal{K} &= \bigcup_{p=0}^{P} \bar{\mathcal{K}}_p^I \cup \mathring{\mathcal{K}}_p \qquad |\mathcal{K}| = \sum_{p=0}^{P} |\bar{\mathcal{K}}_p^I| + |\mathring{\mathcal{K}}_p| \\ \mathcal{W} &= \bigcup_{p=0}^{P} \bar{\mathcal{W}}_p^I \cup \mathring{\mathcal{W}}_p \qquad |\mathcal{W}| = \sum_{p=0}^{P} |\bar{\mathcal{W}}_p^I| + |\mathring{\mathcal{W}}_p| \end{aligned} \tag{4.27}$$

I w przeciwieństwie do 4.24:

$$|\bigcup_{p=0}^{P} \mathring{T}_i \cup \bar{T}_i^I| = |T| \tag{4.28}$$

Z czego wynika, że korzystając z takiej definicji, dla każdego obiektu topologicznego t_i można zawsze wskazać pod-obszar, w którym ten obiekt się znajduje, oraz że skala narzutu obliczeniowego będzie zależeć od stałej c_{narz} określającej szerokość obszarów granicznych

$$c_{narz} = \sum_{p=1}^{P} |\bar{T}_p^E|$$

Poza tym w obliczeniach rozproszonych wymagane jest istnienie operatora pozwalającego jednoznacznie identyfikować te same geometrycznie obiekty w różnych pod-obszarach. Zwyczajowo identyfikator ten określa się mianem *Guid* (z ang. Globally Unique IDentifier)

$$\begin{aligned} Guid: \quad & T \to \mathbb{N} \\ & t_i = t_j \Leftrightarrow Guid(t_i) = Guid(t_j) \qquad \forall t_i, t_j \in T \end{aligned} \tag{4.29}$$

należy podkreślić, że dla rozproszonej siatki, musi on spełniać następującą identyczność przynajmniej w każdej parze pod-obszarów p_i, p_j zawierających obiekt t:

$$\forall p_i, p_j : i, j \in 1, .., P \qquad Guid_{p_i}(t) = Guid_{p_j}(t) \qquad \forall t : t \in T_i \wedge t \in T_j \tag{4.30}$$

Podział siatki pomiędzy pod-obszary oraz adaptacyjne równoważenie obciążenia wymagają także operatorów związanych z przesyłem fragmentów siatki pomiędzy obszarami. Ze względu na możliwość duplikowania pewnych fragmentów siatki można wskazać, że operator transferu $\mathcal{T}$ między pod-obszarami może przyjmować semantykę *kopiowania* $\mathcal{T}_k$ (ang. copy semantics) albo *przemieszczania* (ang. move semantics) $\mathcal{T}_p$. Można zatem zdefiniować efekt działania tego operatora pomiędzy pod-obszarami p_i, p_j jako:

$$\begin{aligned} \mathcal{T}_k: \quad & T \times P \to T \times P^2 \\ & \mathcal{T}_k^i(t,j) = \{t | t \in T_i \wedge t \in T_j \quad \forall t \in T\} \\ \mathcal{T}_p: \quad & T \times P \to T \times P \\ & \mathcal{T}_p^i(t,j) = \{t | t \notin T_i \wedge t \in T_j \quad \forall t \in T\} \end{aligned} \tag{4.31}$$

Trzeba też podkreślić, że dla poprawnie zdefiniowanych operatorów $Guid_p(\cdot), \mathcal{T}_k(\cdot), \mathcal{T}_p(\cdot)$ zachodzi zależność:

$$\begin{aligned} &\forall t \in T \qquad \mathcal{T}_k^i(t,j) \Rightarrow Guid_i(t) = Guid_j(t) \\ &\forall t \in T \qquad \mathcal{T}_p^i(t,j) \Rightarrow Guid_i(t) = \emptyset \wedge Guid_j(t) \neq \emptyset \end{aligned} \tag{4.32}$$

4.1.3. Zarządzanie jako realizacja zbioru operatorów siatki

Konkludując rozważania z punktu 4.1.2: zarządzanie siatką obliczeniową M korzystającą z reprezentacji (omówionej w rozdziale 3) P jest sposobem realizacji (jawnego lub nie) zbioru operatorów $\mathcal{Z}$ nad siatką obliczeniową $M_P = \{\mathcal{E}, \mathcal{S}, \mathcal{K}, \mathcal{W}\}$, pozwalającym wykonywać jej wybrane algorytmy. W zależności od obszerności tego zbioru można dokonać klasyfikacji schematów zarządzania siatką:

hybrydowa siatka obliczeniowa

$$\begin{aligned} \mathcal{Z}_h = \{&\mathcal{G}_{eo}(E), h(E), \mathcal{W}(E), \mathcal{K}(E), \mathcal{S}(E), \mathcal{E}(E), \\ &\mathcal{G}_{eo}(S), h(S), \mathcal{W}(S), \mathcal{K}(S), \mathcal{S}(S), \mathcal{E}(S), \\ &\mathcal{G}_{eo}(K), h(K), \mathcal{W}(K), \mathcal{K}(K), \mathcal{S}(K), \mathcal{E}(K), \\ &\mathcal{G}_{eo}(W), h(W), \mathcal{W}(W), \mathcal{K}(W), \mathcal{S}(W), \mathcal{E}(W)\} \end{aligned} \tag{4.33}$$

siatka adaptacyjna 1–nieregularna

$$\begin{aligned} \mathcal{Z}_a = \{&\mathcal{G}(E), \mathcal{R}(E), \mathcal{P}(E), \mathcal{F}(E), \mathcal{R}'(E), \mathcal{P}'(E), \mathcal{F}'(E), \mathcal{A}(E,i), \mathcal{D}(E_1, ..., E_i, i), \\ &\mathcal{G}(S), \mathcal{R}(S), \mathcal{P}(S), \mathcal{F}(S), \mathcal{R}'(S), \mathcal{P}'(S), \mathcal{F}'(S), \mathcal{A}(S,i), \mathcal{D}(S_1, ..., S_i, i), \\ &\mathcal{G}(K), \mathcal{R}(K), \mathcal{P}(K), \mathcal{F}(K), \mathcal{R}'(K), \mathcal{P}'(K), \mathcal{F}'(K), \mathcal{A}(K,i), \mathcal{D}(K_1, ..., K_i, i), \\ &\mathcal{G}(W), \mathcal{R}(W), \mathcal{F}(W), \mathcal{R}'(W), \mathcal{F}'(W)\} \end{aligned} \tag{4.34}$$

siatka rozproszona na p podobszarów

$$\begin{aligned} \mathcal{Z}_r = \{&\mathring{\mathcal{E}}, \bar{\mathcal{E}}_p^I, \bar{\mathcal{E}}_p^E, Guid(E), \mathcal{T}_k(E,j), \mathcal{T}_p(E,j), \mathcal{O}(E), \\ &\mathring{\mathcal{S}}, \bar{\mathcal{S}}_p^I, \bar{\mathcal{S}}_p^E, Guid(S), \mathcal{T}_k(S,j), \mathcal{T}_p(S,j), \mathcal{O}(S), \\ &\mathring{\mathcal{K}}, \bar{\mathcal{K}}_p^I, \bar{\mathcal{K}}_p^E, Guid(K), \mathcal{T}_k(K,j), \mathcal{T}_p(K,j), \mathcal{O}(K), \\ &\mathring{\mathcal{W}}, \bar{\mathcal{W}}_p^I, \bar{\mathcal{W}}_p^E, Guid(W), \mathcal{T}_k(W,j), \mathcal{T}_p(W,j), \mathcal{O}(W), \} \end{aligned} \tag{4.35}$$

Zatem omawiane przez nas tytułowe zagadnienie *zarządzania rozproszoną siatką obliczeniową na nowoczesnych architekturach komputerowych* formalnie można

przedstawić jako zbiór operatorów

$$
\begin{aligned}
\mathcal{Z}_t =& \mathcal{Z}_h \cup \mathcal{Z}_a \cup \mathcal{Z}_r, \qquad czyli \\
\mathcal{Z}_t =& \{\mathcal{G}_{eo}(E), h(E), \mathcal{W}(E), \mathcal{K}(E), \mathcal{S}(E), \mathcal{E}(E), \\
& \mathcal{G}_{eo}(S), h(S), \mathcal{W}(S), \mathcal{K}(S), \mathcal{S}(S), \mathcal{E}(S), \\
& \mathcal{G}_{eo}(K), h(K), \mathcal{W}(K), \mathcal{K}(K), \mathcal{S}(K), \mathcal{E}(K), \\
& \mathcal{G}_{eo}(W), h(W), \mathcal{W}(W), \mathcal{K}(W), \mathcal{S}(W), \mathcal{E}(W), \\
& \mathcal{G}(E), \mathcal{R}(E), \mathcal{P}(E), \mathcal{F}(E), \mathcal{R}'(E), \mathcal{P}'(E), \mathcal{F}'(E), \mathcal{A}(E,i), \mathcal{D}(E_1, ..., E_i, i), \\
& \mathcal{G}(S), \mathcal{R}(S), \mathcal{P}(S), \mathcal{F}(S), \mathcal{R}'(S), \mathcal{P}'(S), \mathcal{F}'(S), \mathcal{A}(S,i), \mathcal{D}(S_1, ..., S_i, i), \\
& \mathcal{G}(K), \mathcal{R}(K), \mathcal{P}(K), \mathcal{F}(K), \mathcal{R}'(K), \mathcal{P}'(K), \mathcal{F}'(K), \mathcal{A}(K,i), \mathcal{D}(K_1, ..., K_i, i), \\
& \mathcal{G}(W), \mathcal{R}(W), \mathcal{F}(W), \mathcal{R}'(W), \mathcal{F}'(W), \\
& \mathring{\mathcal{E}}, \bar{\mathcal{E}}_p^I, \bar{\mathcal{E}}_p^E, Guid(E), \mathcal{T}_k(E,j), \mathcal{T}_p(E,j), \\
& \mathring{\mathcal{S}}, \bar{\mathcal{S}}_p^I, \bar{\mathcal{S}}_p^E, Guid(S), \mathcal{T}_k(S,j), \mathcal{T}_p(S,j), \\
& \mathring{\mathcal{K}}, \bar{\mathcal{K}}_p^I, \bar{\mathcal{K}}_p^E, Guid(K), \mathcal{T}_k(K,j), \mathcal{T}_p(K,j), \\
& \mathring{\mathcal{W}}, \bar{\mathcal{W}}_p^I, \bar{\mathcal{W}}_p^E, Guid(W), \mathcal{T}_k(W,j), \mathcal{T}_p(W,j), \\
& \mathcal{O}(E), \mathcal{O}(S), \mathcal{O}(K), \mathcal{O}(W)\}
\end{aligned}
\tag{4.36}
$$

na siatce obliczeniowej w symulacjach adaptacyjną metodą elementów skończonych oraz jego implementację.

Warto zauważyć, że dzięki formalnym definicjom 4.23 — 4.32 nie jest wymagane dalsze różnicowanie i uzupełnianie zestawów operatorów w celu opisania zarządzania dla np.: *adaptacyjnej siatki hybrydowej* lub *adaptacyjnej siatki rozproszonej* itp. itd.

4.2. Kryterium oceny jakości zarządzania siatką obliczeniową

Miara, którą możemy przypisać metodzie zarządzania siatką, jest dwojaka. Z jednej strony na pewno jest to ilość i rodzaj dostarczanych operatorów. One decydują o potencjalnej przydatności siatki obliczeniowej dla praktycznego użytku. Tutaj docelowy zakres rozważań był ściśle określony od początku pracy i w punkcie 4.1.2 został przedstawiony formalnie. Zatem drugim kryterium jest jakość realizacji tychże operatorów. W tym przypadku inżynieria oprogramowania wskazuje na sześć kluczowych cech określających, jak *dobre* jest przedstawione rozwiązanie. Są to wg Sommerville'a:

poprawność - poprawność siatki jest zapewniona przez poprawną realizację dostarczanych operatorów,

łatwość dokonywania zmian - wynika z separacji i jednoznaczności operatorów,
niezawodność - rozumiana jako pewność, wynika z formalnych definicji,
bezpieczeństwo - rozumiane w kontekście kryptografii oraz nie stwarzania zagrożeń jest tutaj pomijane,
łatwość stosowania - wynika z separacji i jednoznaczności operatorów,
wydajność - wynika z jakości realizacji operatorów.

Zatem, dla tak postawionego problemu, najbardziej informatywną miarą jakości zarządzania siatką jest *wydajność*. Jest to zgodne z intuicją, że podstawowym powodem dla szerokiego zainteresowania omawianymi zagadnieniami – jak to zostało przedstawione w rozdziale 1.4 – jest uzyskanie możliwości modelowania zjawisk o większej złożoności obliczeniowej (skalowalność słaba) lub modelowania zjawisk szybciej (skalowalność mocna). U podstaw, oba te kierunki wynikają ze zbyt małych zasobów obliczeniowych, zarówno w sensie wymaganej pamięci (GB RAM), jak i mocy obliczeniowej (Gflop). Stąd, wykorzystanie nowoczesnych architektur obliczeniowych powinno być jak najbardziej wydajne, ponieważ właśnie poszukiwanie wydajności jest powodem ich użytkowania.

Ponieważ schematy zarządzania siatką obliczeniową nie są samodzielnymi programami, lecz współrealizują symulacje wraz z innymi bibliotekami realizującymi aproksymację metodą elementów skończonych, należy oddzielić, które fragmenty oprogramowania symulacyjnego podlegają ocenie wydajnościowej jako cześć zarządzania siatką, a które nie. I tutaj można skorzystać z przedstawionej definicji zarządzania siatką obliczeniową i wskazać, że fragmentami podlegającymi ocenie wydajnościowej są te realizujące operatory dostarczane przez siatkę obliczeniową. Ponieważ operatory są ściśle zdefiniowane, łatwo można wskazać zarówno teoretyczną wydajność operatora, jak i zmierzyć faktyczny udział w całkowitym czasie symulacji.

Łącząc powyższe kryteria oraz modele siatki obliczeniowej przedstawione w rozdziale 3 otrzymujemy, iż:

— dla wybranego zbioru $\mathcal{Z}$
— zawierającego operatory o charakteryzujące się złożonością $O_o(n)$ i średnią częstotliwością f_o
— nad siatką obliczeniową M
— z określoną reprezentacją P
— posiadającej n obiektów w danej kategorii topologicznej

za miarodajne kryterium oceny można przyjąć:

1. Teoretyczną wydajność obliczeniową:

$$O(M_P) = \sum(\forall o \in \mathcal{Z})(O_o(n) \cdot f_o) \tag{4.37}$$

2. Wydajność pamięciową:

$$O_m(M_P) = nm_o + n_c m_c \tag{4.38}$$

Kryteria te posłużyły do oceny wpływu na wydajność ogólną poszczególnych operatorów siatki obliczeniowej, co zostało omówione w następnych rozdziałach.

Rozdział 5

Wektoryzacja na poziomie rdzenia obliczeniowego

W kolejnych rozdziałach analizowana jest implementacja operatorów siatki, ze szczególnym uwzględnieniem odwzorowania na architekturę sprzętu obliczeniowego. Kolejno badane są możliwości wektoryzacji, zrównoleglenia w modelu pamięci wspólnej i pamięci rozproszonej. Dla każdego z powyższych modeli obliczeń zwrócona jest uwaga na specyficzne problemy odwzorowania na architekturę. Dla niektórych problemów przedstawione są nowatorskie rozwiązania, o wartości teoretycznej i praktycznej.

5.1. Przyjęte miary dotyczące obliczeń

Na potrzeby analizy skalowania opracowanych rozwiązań i ich implementacji użyto następujących standardowych miar dot. przyspieszenia, skalowalności i efektywności tychże:

Przyspieszenie dla p procesów/wątków $S(p) = \frac{T_1}{T_p}$, gdzie T_1 jest czasem rozwiązania dla 1 procesu/wątku,a T_p jest czasem rozwiązania dla p procesów/wątków.

Efektywność dla p procesów/wątków efektywność zrównolglenia $E(p) = \frac{S(p)}{p}$.

Przyspiesznie skalowane dla p procesów/wątków jest zdefiniowane jako

$$S^P(p) = \frac{T_{II}(1, pW_0)}{T_{II}(p, pW_0)} = \frac{p * T_{II}(1, W_0)}{T_{II}(p, pW_0)}$$

gdzie,
W_0 - liczba operacji lub praca do wykonania (ang. workload)
$T_{II}(1, pW_0)$ - czas rozwiązania zadania o rozmiarze $p * W_0$ na 1 procesie/wątku
$T_{II}(p, pW_0)$ - czas rozwiązania równoległego zadania o rozmiarze $p * W_0$ używając p procesów/wątków.

Efektywność skalowana dla p procesów/wątków efektywność skalowalna jest definiowana jako $E^P(p) = \frac{S^P(p)}{p}$.

O ile samo pojęcie skalowalności może mieć różne znaczenia, to w dziedzinie obliczeń równoległych i rozproszonych wyróżnia się dwa rodzaje skalowalności:

Skalowalność silna dotyczy przyspieszania obliczeń przy stałym rozmiarze zadania, ale zwiększającej się liczbie zasobów. Prawo Amdahl'a [8] dotyczy skalowalności silnej. Występowanie skalowalności silnej przeważnie implikuje skalowalność słabą. Idealnym przypadkiem skalowalności silnej jest sytuacja, gdy uruchamiając to samo zadanie na p razy większych zasobach, uzyskujemy rozwiązanie p razy szybciej. Odpowiada to sytuacji, gdy przyspieszenie $S(p)$ jest funkcją liniową.

Skalowalność słaba dotyczy utrzymania stałego czasu obliczeń w miarę proporcjonalnego zwiększania jednocześnie wielkości zadania i udostępnianych zasobów. Prawo Gustavson'a [52] dotyczy skalowalności słabej. Idealnym przypadkiem skalowalności słabej jest sytuacja, gdy wykonanie pracy W_p, (która jest p razy większa niż praca W_0) na p razy większych zasobach niż W_0 zajmuje tyle samo czasu. Np.: dzieje się tak, gdy przyspieszanie skalowalne obliczeń $S^P(p)$ jest funkcją liniową.

5.2. Wykorzystanie możliwości przetwarzania wektorowego

Ze względu na ograniczenie zakresu badań, wektoryzacja była badana w odniesieniu do rdzeni ogólnego przeznaczenia. Istnieje szereg dość restrykcyjnych warunków, które muszą zostać spełnione, by kompilator mógł wykorzystać wektoryzację sprzętową na takich rdzeniach [58, 54, 25]. Są to:

— dane muszą być przechowywane w pamięci w ciągłych blokach,
— rozmiar danych (w Bajtach) musi być wielokrotnością wielkości słowa maszynowego (WORD, zazwyczaj 16B, ale zależy od konkretnej architektury),
— dane muszą być typu wbudowanego,
— dane całkowitoliczbowe (integer) muszą się mieścić w rejestrze 128 bitowym, a zmiennoprzecinkowe (floating point) w rejestrze 256 bitowym,
— pętla przetwarzająca dane musi posiadać prosty (trivial) warunek końca pętli,
— pętla nie może być ręcznie zoptymalizowana poprzez technikę rozwijania pętli (roll-out),
— zmienna sterująca pętlą nie może być podatna na błąd przepełnienia (overflow),
— wewnątrz wektoryzowanego bloku nie mogą się znajdować instrukcje warunkowe, skoki, przerwania etc.,
— dane przetwarzane w bloku muszą być adresowane bezpośrednio,
— indeksowanie danych nie może uwzględniać zmiennych nieznanych na etapie kompilacji.

Ponadto, aby wykorzystać potencjał wekoryzacji dla bardziej skomplikowanych operacji, konieczne jest przedstawienie ich w prostych formach uwzględnionych w zestawach np.: SSE i AVX. Przy czym należy zaznaczyć, że w zależności od konkretnego zestawu funkcji dopuszczalne operacje ulegające wektoryzacji są różne.

W ramach zarządzania siatką obliczeniową, operacją, która przetwarza większe ilości danych na raz, a przez to jest potencjalnie najlepszym kandydatem do wektoryzacji, jest adaptacja siatki obliczeniowej. Wykorzystując algorytm 1, przeprowadzono testy sprawdzające efektywność wektoryzacji. Wykazały one, że problemem na pewno pozostaje długość, lub raczej niedostateczna długość, wewnętrznej pętli algorytmu. Efektem tego wektoryzacja albo nie była przez kompilator stosowana w ogóle, lub szczątkowo. W celu pokonania tej przeszkody, została opracowana kolejna wersja algorytmu.

Przedstawiony wcześniej algorytm 1 adaptacji pojedynczego elementu siatki może być wykonywany w długiej pętli, która spełnia część warunków wektoryzacji dotyczących organizacji kodu, co zostało przedstawione w algorytmie 3. Kluczowym zabiegiem jest tutaj wykonywanie poszczególnych kroków algorytmu w odniesieniu nie do jednego elementu, ale wszystkich naraz. Powoduje to przejście z modelu pojedynczej pętli zawierającej trzy etapy na dwie pętle zawierające odpowiednio etap pierwszy oraz drugi i trzeci łącznie. Oczywiście wymaga to dostarczenia odpowiednich struktur danych. Dla uproszczenia zakładamy, że wszystkie elementy są dzielone w ten sam sposób, co w praktyce implementacyjnej nie jest prawdą, ale bardzo skomplikowałoby zapis algorytmu. Zwektoryzowana wersja algorytmu 1 przedstawiona w algorytmie 3 napotyka na szereg problemów na poszczególnych etapach:

1. Etap pierwszy jest etapem przygotowania danych. Sprawdzana jest legalność podziału, ze względu na przyjęte założenia odnośnie do nieregularności — w tym przypadku jest to warunek 1-nieregularności. Wszystkie elementy, które dodatkowo muszą zostać podzielone, są dodawane na koniec kolekcji z elementami do podziału. Oznacza to potencjalny wzrost całkowitej liczby dzielonych elementów. Długość pętli z nieznanej na początku, ale zawsze ograniczonej liczby sąsiadów (w praktyce bardzo rzadko więcej niż 30), została znacząco wydłużona liniowo względem liczby elementów, co jest pożądane dla wektoryzacji, choć nadal nie jest znana. Niestety, ponieważ etap ten w najbardziej zagnieżdżonej pętli zakłada sprawdzanie warunków logicznych, to nie jest dobrym kandydatem do wektoryzacji.
2. Etap drugi. Tutaj również głównym argumentem za potencjalnym zastosowaniem wektoryzacji jest długość pętli. Niestety warunkowe wykonywanie w środku pętli podziału ściany absolutnie wyklucza pierwszą pod-pętlę z wektoryzacji. Natomiast druga pod-pętla wydaje się bardziej obiecująca, ponieważ zawsze znana jest jej długość. Problemem jest tutaj tworzenie obiektów potomnych. Wymaga

ono albo dynamicznej alokacji pamięci albo synchonicznego przydzielania identyfikatorów, co wprowadza do tej pętli zależność przenoszoną w pętli, co również wyklucza wektoryzację.

Algorytm 3: Dwuetapowy algorytm adaptacji wybranej kolekcji elementów siatki hybrydowej (3D).

Dane: zbiór N elementów do podziału $\boldsymbol{E} = \{E_0, E_1, E_2, ..., E_N\}$,
łączności dla tych elementów $\boldsymbol{I} = \{I_0, I_1, I_2, ..., I_N\}$
sposób podziału P,
ilość obiektów potomnych pojedynczego elementu n_P
Rezultat: zbiór elementów potomnych
$\boldsymbol{e} = \{\{e_0^0, ..., e_{n_P}^0\}, \{e_0^1, ..., e_{n_P}^1\}, ..., \{e_0^N, ..., e_{n_P}^N\}\}$
informacje o łączności stworzonych obiektów
$\{\{I_0^0, ..., I_{n_P}^0\}, \{I_0^1, ..., I_{n_P}^1\}, ..., \{I_0^N, ..., I_{n_P}^N\}\}$

```
Etap I
Gen(e) = generacja (poziom) obiektu e ;
for (i = 0; i < |E|; ++ i) do
    for element e sąsiadujący (również pośrednio) z E_i do
        if |Gen(E_i) − Gen(e)| ⩾ 1 then
            Put_end(E, e) ;
        end
    end
end
Etap II
for (i = 0; i < |E|; ++ i) do
    for ściany f należącej do elementu E_i do
        if ściana f jest niepodzielona then
            Podziel ścianę f;
            Zakłada się, że podział ściany pociąga za sobą także podział jej
            części składowych.
        end
    end
    for n <= n_P do
        Stwórz nowy obiekt potomny e_n^i;
        Przypisz do obiektu e_n^i poprawne informacje o łączności horyzontalnej
        i wertykalnej I_e^i między obiektami;
    end
end
```

Niestety ze względu na nieliniową strukturę powszechne są niebezpośrednie odwołania do danych, które nie spełniają warunku bezpośredniej indeksacji pamięci. By rozwiązać ten problem, opracowana została nowa reprezentacja algorytmu adaptacji, dostosowana do wymagań wektoryzacji.

5.3. Adaptacja z wykorzystaniem wektoryzacji

Wstępna analiza problemu wykorzystania wektoryzacji wykazała, że podstawowymi problemami blokującymi możliwość wektoryzacji jest kwestia zarządzania pamięcią, instrukcje warunkowe oraz pośrednie indeksowanie danych. W celu usunięcia tych problemów algorytm został przeprojektowany. Niemniej jednak, by było to możliwe, należało najpierw wprowadzić pewne elementy konieczne, by to zrealizować.

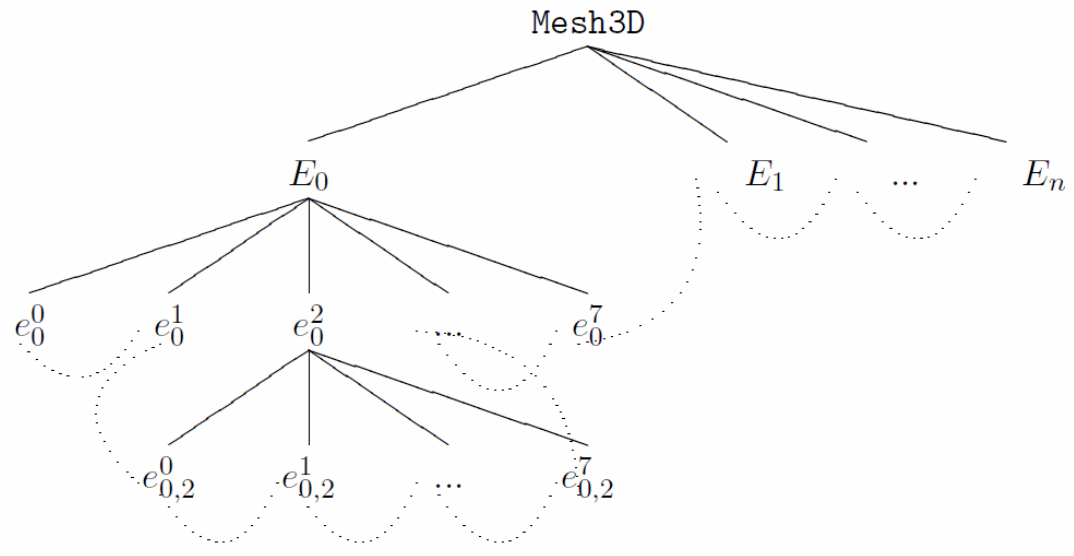
5.3.1. Zarządzanie pamięcią dla wektoryzacji w siatkach hybrydowych

We współczesnych systemach zarządzanie pamięcią jest bezpieczne wielowątkowo (*thread safe*), ale niestety na razie nie jest wektoryzowalne w żaden sposób. W związku z tym musiało zostać usunięte poza obszar wektoryzacji. Dodatkowo specyfika instrukcji wektorowych zakłada ciągłość obszaru pamięci. W celu zrealizowania tych i innych wymagań został opracowany schemat zarządzania pamięcią [19]. Schemat ten wykorzystuje znaną koncepcję tzw. *memory pooling*, która w głównym zarysie polega na przejęciu od bibliotek kompilatora pewnego dużego obszaru pamięci i samodzielnego nim zarządzania, bez delegowania operacji przydzielania i zwalniania do funkcji standardowych takich jak `new,delete,malloc,free` itp. Sam schemat przechowywania w pamięci danych dla obiektów siatki nie jest szczególnie problematyczny dla siatek o małej różnorodności obiektów, a bardziej konkretnie: o obiektach mających ten sam rozmiar w bajtach. Problematyczne jest przechowywanie dużej ilości obiektów o różnorodnych rozmiarach w ramach tej samej kategorii obiektów siatki, co występuje w siatce hybrydowej. Należy uwzględnić jednorodne indeksowanie obiektów tej samej kategorii, co jest niezwykle pożądane. Jednocześnie, szczególnie ze względu na ograniczenie rozmiaru rejestrów wektorowych, obiekty przechowywane w pamięci powinny być upakowane tak gęsto jak to możliwe. W tym celu wprowadzono rozróżnienie na dwa rodzaje obiektów w siatce adaptacyjnej: obiekty podzielone, które przechowują pewne dodatkowe informacje dotyczące podziału i potomków, oraz obiekty niepodzielone zajmujące wielokrotnie mniej miejsca w pamięci. W celu utrzymania ciągłości obszarów pamięci również wymaganej przez wektoryzację wprowadzono specyficzny porządek elementów, który jest pokazany na rysunku 5.1. Widać, że w wyniku podziału elementów, kolejność iterowania po elementach nie jest zmieniona i zachowuje lokalność. Może zostać zdefiniowany pewien porządek liniowy na zbiorze wszystkich elementów początkowych i potomnych (powstałych w wyniku adaptacji). Wprowadźmy następujące oznaczenia: E_n lub e^n – n-ty element początkowy, natomiast e^n_k k–ty syn n–tego elementu początkowego w pierwszej generacji. Sposób iterowania pokazany na rys. 5.1 odpowiada porządkowi liniowemu:

$$(\boldsymbol{E}, \preceq) : \forall e^n_k, e^m_l \in E : e^n_k \preceq e^m_l \wedge e^m_l \succeq e^n_k \tag{5.1}$$

Przy czym operator określający ten porządek jest zdefiniowany w sposób następujący:

$$e_k^n \preceq e_l^m \iff$$
$$(n = m \wedge k \leqslant l) \vee (n \neq m \wedge n \leqslant m)$$

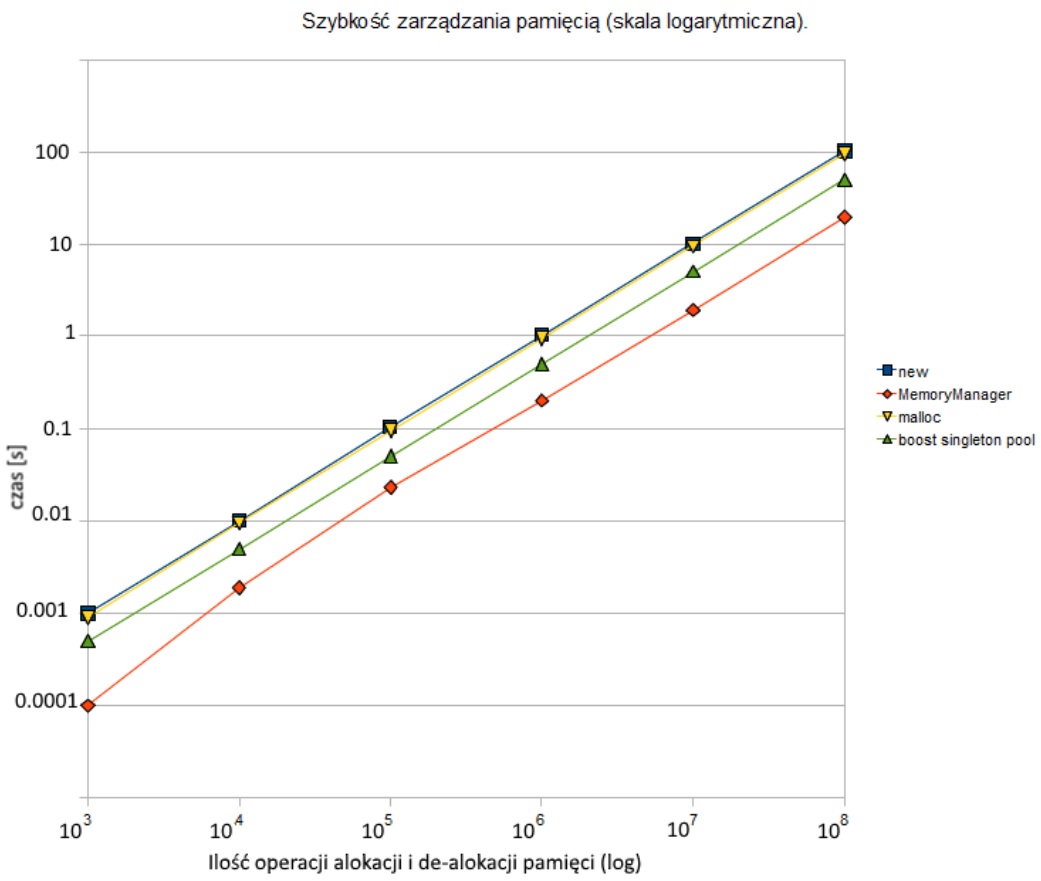


Rysunek 5.1: Naturalny porządek iterowania po elementach w siatce adaptacyjnej.

O efektywności opracowanej struktury świadczą wyniki zaprezentowane na rysunkach 5.2, 5.3 zebrane w czasie przeprowadzania testów opracowanego rozwiązania. Na rys. 5.2 widać porównanie z innymi rozwiązaniami w czasie wykonywania różnej ilości operacji alokacji i de-alokacji pamięci odpowiadającym tworzeniu i usuwaniu obiektów z siatki. Na rys. widać czas potrzebny na przejście przez wszystkie elementy w siatce i odczytanie identyfikatora obiektów składowych każdego z nich. Jak widać zarówno porównanie do standardowych rozwiązań, jak i sam pomiar średniego czasu (w nanosekundach) wskazują, że uzyskane rozwiązanie pozwala na szybszy dostęp do danych oraz że średni czas dostępu do danych siatki odpowiada rzędowi wielkości, jaki jest charakterystyczny w przypadku efektywnego wykorzystania hierarchii pamięci procesora (w przypadku pamięci cache poziomu L1 wynosi on kilka nanosekund).

5.4. Algorytm adaptacji z wykorzystaniem wektoryzacji

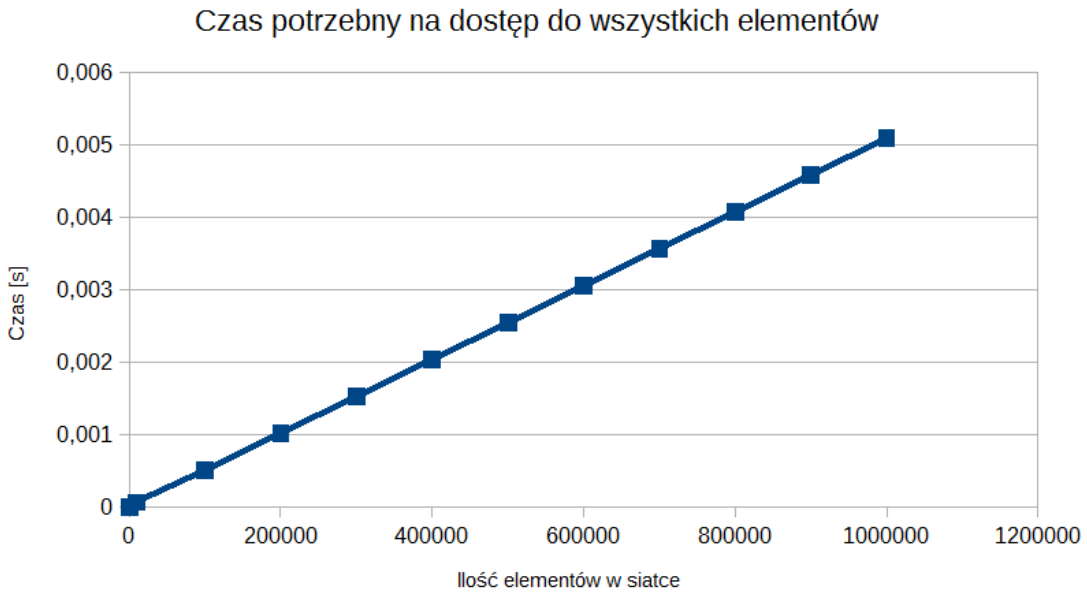
Wykorzystanie własnego schematu zarządzania pamięcią umożliwiło wprowadzenie modyfikacji w Algorytmie 3 polegających na wyłączeniu fragmentów związanych z zarządzeniem pamięcią przed obszar podlegający wektoryzacji. W ten sposób



Rysunek 5.2: Szybkość zarządzania pamięcią względem standardowych rozwiązań. Dla różnych ilości operacji alokacji i de-alokacji pamięci (new,delete albo malloc,free) mierzono czas całkowity czas wykonania żądanych operacji.

pierwsze etapy algorytmu stały się typowym przygotowaniem danych wejściowych oraz miejsca na dane wyjściowe. Co więcej, stosując omówiony wyżej sposób zarządzania pamięcią do każdej kategorii obiektów siatki z osobna, możliwe stało się rozdzielenie operacji na różnych typach obiektów, co otworzyło szersze możliwości wykonania wielowątkowego. Przedstawiony poniżej Algorytm 4 posiada już trzy etapy. W pierwszym etapie zbieramy całkowite dane o ilości obiektów, które powstaną, uwzględniając wymagania nieregularności. Dotyczy to wszystkich typów obiektów w siatce obliczeniowej z osobna. W etapie drugim rezerwujemy pamięć, bazując na informacjach zebranych w etapie pierwszym. W etapie trzecim ostatecznie otrzymaliśmy pętle bez jawnych instrukcji warunkowych i operacji pamięciowych, która przynajmniej teoretycznie nadaje się do wektoryzacji.

Nie poddające się wektoryzacji indeksowanie pośrednie, zostało zastąpione wykorzystaniem odpowiednio dobranych dla wektoryzacji tzw. tablic logicznych. Wykorzystanie tego mechanizmu wymaga wprowadzenia do algorytmu dodatkowego etapu przygotowania danych. Następnym problemem była duża ilość instrukcji warunko-



Rysunek 5.3: Czas dostępu do elementów w czasie iterowania po strukturach siatki. Przy dużych ilościach obiektów, lokalny schemat odwołań skutkuje czasem przejścia z jednego obiektu siatki do drugiego na poziomie 5 nanosekund (średnio).

wych odpowiedzialnych za odpowiednie wykrywanie i rozpatrywanie szczególnych przypadków (dość częstych w przypadku siatki 1–nieregularnej). Instrukcje warunkowe zostały usunięte z głównej pętli algorytmu i zastąpione odpowiednim doborem tablic logicznych na etapie przygotowania danych. Spowodowało to przejście z prostego schematu paru tablic logicznych na bardziej złożone wykorzystanie układu tablic decyzyjnych. Nowe podejście spowodowało przesunięcie wielokrotnego sprawdzania różnych przypadków na jednorazowy dobór odpowiedniej tablicy decyzyjnej.

Ostateczna wersja algorytmu jest zmodyfikowanym algorytmem 4. Algorytm 5 przedstawia ostateczną wersję opracowanego rozwiązania. Można zauważyć, iż ze względu na ograniczoną ilość szczególnych przypadków adaptacji pewne grupy elementów będą posiadały identyczne tablice A i K. Uwzględniając podział ze względu na tę własność, etap pierwszy algorytmu przygotowujący dane sprowadza się do wypełnienia tablic P i K. W praktyce tablice te przechowują dane dotyczące:

P = [[obiekty potomne],[łączność obiektów potomnych]]
E = [typ elementu, [komponenty elementu], [łączność komponentów], parametry rozpatrywanego przypadku]
A = [typ podziału, [liczba obiektów potomnych każdej kategorii]]
K = [łączność (schemat sąsiadów)]

Algorytm 4: Algorytm adaptacji wybranej kolekcji elementów siatki hybrydowej (3D) z wykorzystaniem własnego schematu zarządzania pamięcią.

Dane: zbiór N elementów do podziału $\boldsymbol{E} = \{E_0, E_1, E_2, ..., E_N\}$,
łączności dla tych elementów $\boldsymbol{I} = \{I_0, I_1, I_2, ..., I_N\}$
sposób podziału P,
ilość obiektów potomnych pojedynczego elementu n_P,
zbiór ścian do podziału $\boldsymbol{F} = \emptyset$,
zbiór krawędzi do podziału $\boldsymbol{K} = \emptyset$
Rezultat: zbiór elementów potomnych
$\boldsymbol{e} = \{\{e_0^0, ..., e_{n_P}^0\}, \{e_0^1, ..., e_{n_P}^1\}, ..., \{e_0^N, ..., e_{n_P}^N\}\}$
informacje o łączności stworzonych obiektów
$\{\{I_0^0, ..., I_{n_P}^0\}, \{I_0^1, ..., I_{n_P}^1\}, ..., \{I_0^N, ..., I_{n_P}^N\}\}$

Etap I

```
for (i = 0; i < |E|; ++i) do
    for element e sąsiadujący (również pośrednio) z E_i do
        if |Gen(E_i) − Gen(e)| > 1 then
            Put_end(E, e) ;
        end
    end
    for niepodzielona ściana f w elemencie E_i do
        Put_end(F, f) ;
        for niepodzielona krawędź k w ścianie f do
            Put_end(K, k) ;
        end
    end
end
```

Etap II

Zarezerwuj ciągłą pamięć dla $|\boldsymbol{K}|$ nowych krawędzi, $|\boldsymbol{F}|$ nowych ścian, $|\boldsymbol{E}|$ nowych elementów ;

Etap III (osobno dla $\boldsymbol{K}, \boldsymbol{F}, \boldsymbol{E}$, poniżej dla elementów:)

```
for (i = 0; i < |E|; ++i) do
    for n <= n_P do
        Przypisz do obiektu e_n^i poprawne informacje o łączności horyzontalnej
        i wertykalnej I_e^i między obiektami;
    end
end
```

5.5. Pomiary wydajności i wnioski

Wykresy na rysunkach 5.4 i 5.5 przedstawiają porównanie uzyskanych wyników bez wykorzystania wektoryzacji i z wykorzystaniem opracowanego algorytmu 5. Do testów wykorzystano komputer z procesorem Intel siódmej generacji Core I7-3930K, Hexa Core, 3.20GHz, pamięć 4x8GB DDR3 1866MHz Dual Channel, system operacyjny Ubuntu 14.04, kernel 3.14. Wynika z nich, iż algorytm 5 przyspieszył realizację adaptacji, a analiza kodu obliczeniowego wykazała zastosowanie instrukcji wekto-

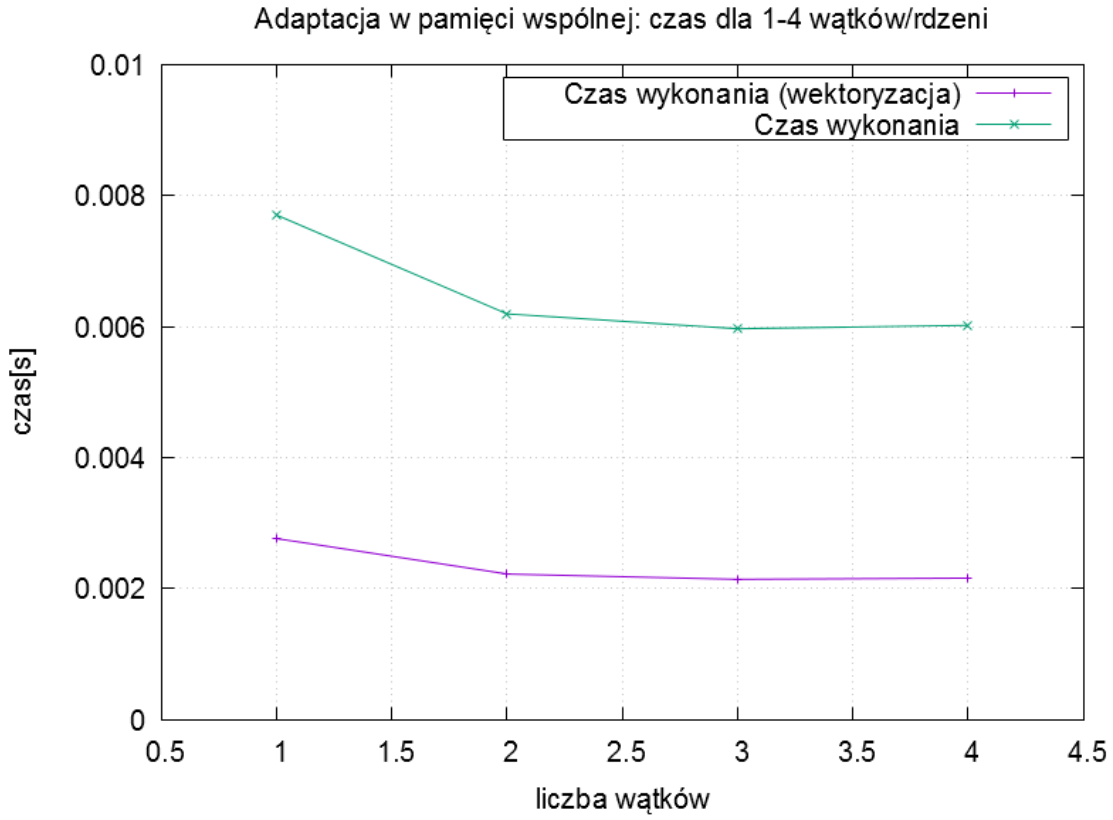
Algorytm 5: Algorytm równoległej adaptacji siatki hybrydowej z wykorzystaniem tablic decyzyjnych.

Dane: Jak w algorytmie 4
Rezultat: Jak w algorytmie 4

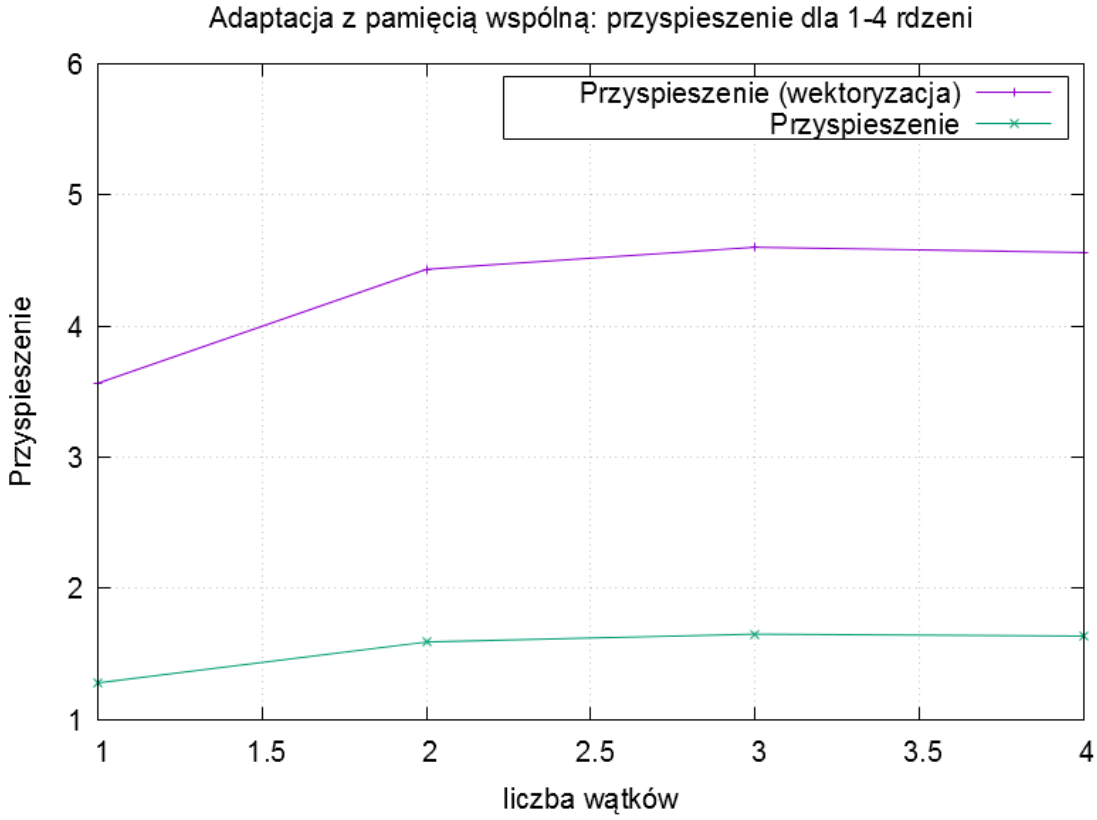
Etap I jak w algorytmie 4.
Etap II jak w algorytmie 4.
for *(*$i = 0; i < |\boldsymbol{E}|; ++ i$*)* **do**
Przygotuj dane do adaptacji elementu e.
1. Przygotuj tablicę P z miejscem na dane dotyczące obiektów potomnych i ich łączności;
2. Przygotuj dane w tablicy E dotyczące elementu e i jego wierzchołków i komponentów oraz łączności tych obiektów;
3. Wybierz tablicę A zawierającą dane dotyczące rodzaju wybranej adaptacji;
4. Wybierz tablicę K zawierającą dane dotyczące kontekstu i otoczenia elementu e;

Przeprowadź adaptację elementu e na podstawie przygotowanych danych.
$P = A(E) + K(E)$;
$E = E + A(E)$;
end

rowych. Wykres przedstawiony na rys. 5.4 pokazuje, że dzięki ich wykorzystaniu możliwe było trzykrotne zmniejszenie czasu wykonania względem wcześniejszych rezultatów. Efektem jest uzyskanie przyspieszenia przedstawionego na rys. 5.5, które wzrosło z poziomu 1.5-2 do zakresu 4-5. Chociaż, uzyskana efektywność przyspieszenia jest znacznie lepsza niż w wersji bez wektoryzacji, to należy zaznaczyć, że dla problemów z dużą intensywnością obliczeniową wielowątkowość w połączeniu z wektoryzacją pozwala uzyskać lepsze rezultaty. Mimo to można podkreślić, że wektoryzacja dostarczana przez nowoczesne procesory ogólnego przeznaczenia w postaci grup instrukcji SSE i AVX może zostać skutecznie wykorzystana do zwiększenia efektywności wykonania również względem zagadnień o mniejszej intensywności obliczeniowej. Chociaż sama wektoryzacja jest już w procesorach i tylko „czeka" by być wykorzystana — i w tym sensie jest darmowa, to nakład pracy związany z dostosowaniem się do niej dla zadań nie zdominowanych przez obliczenia jest bardzo duży. Istotnym aspektem opracowanego schematu jest wykorzystanie logiki tablicowej w postaci tablic decyzyjnych, co umożliwia dostosowanie wymagań funkcjonalnych do warunków wektoryzacji.



Rysunek 5.4: Wykres czasu wykonania zadania adaptacji z wykorzystaniem wektoryzacji i bez niej.



Rysunek 5.5: Wykres przyspieszenia wielowątkowego dla zadania adaptacji z wektoryzacją i bez.

Rozdział 6

Wielowątkowość i pamięć wspólna

Posiadając ścisłe sformułowanie problemu, omówione w rozdziale 4, można zbadać, które operatory siatki obliczeniowej mają istotny wpływ na jej wydajność, ponieważ zgodnie z punktem 4.2 jest to kryterium jakości realizacji zarządzania siatką obliczeniową. W tym celu przeprowadzono szereg czynności dla reprezentatywnych przypadków akademickich i z praktyki przemysłowej.

6.1. Profilowanie wpływu operatorów siatki na całkowity czas symulacji

Do badania częstości i charakterystyki używania operatorów siatki przez oprogramowanie symulacyjne został wybrany program ModFEM. Program ten posiada otwarte źródła, dobrze udokumentowaną strukturę wewnętrzną [15, 14, 94, 95] oraz był dobrze znany autorowi, co bardzo pomaga uniknąć błędnych wyników spowodowanych poprzez nieumiejętne wykorzystywanie dostarczanych komponentów programowych.

W celu określenia istotnych punktów wymagających optymalizacji zostało przeprowadzone profilowanie na trzech przykładach modelowania:

— przepływ Lid Driven Cavity (LDC), które jest klasycznym zadaniem testowym dla programów symulacyjnych modelujących numerycznie przepływy płynów nieściśliwych .
— Modelowanie rzeczywistego procesu spawania , który w przeciwieństwie do LDC jest procesem niestacjonarnym, zatem lepiej obrazuje dynamiczną adaptację i de-adaptację.
— Modelowanie zjawiska konwekcji-dyfuzji, ale z uwzględnieniem znajomości rozwiązania dokładnego (problem Laplace'a).

Wykorzystane zostały następujące narzędzia profilujące, o różnych sposobach działania:

Parallel Studio XE Composer Edition for C++ Linux 2014 jest to duże narzędzie firmy Intel do profilowania aplikacji pod różnym kontem. Poza wieloma

Własne	Funkcja
2 742 302	mmr_el_node_coor
2 677 478	select_mesh(int)
2 622 046	mmr_node_status
918 328	mmr_el_type
799 940	mmr_get_next_act_face
574 840	mmr_el_hsize
561 306	mmr_fa_bc
518 962	mmr_fa_neig
437 710	mmr_add_face
426 594	mmr_get_next_act_elem
226 818	mmr_el_status
183 268	mmr_add_elem
172 202	mmr_el_groupID
169 620	mmr_fa_status
126 232	mmr_fa_eq_neig
123 484	mmr_get_next_face_all
122 224	mmr_el_fa_nodes
99 840	mmr_get_next_node_all
65 640	mmr_add_node
56 158	mmr_fa_area
47 830	mmr_fa_type
26 004	mmr_get_next_elem_all
1 286	mmr_init_read
1 047	mmr_init_mesh2
520	mmr_get_max_node_id
494	writeOutputStream()
392	mmr_get_nr_face
382	mmr_get_nr_node
357	mmr_finish_read
276	mmr_get_nr_elem
264	mmr_test_mesh
245	mmr_set_max_gen
232	mmr_get_max_gen_diff
170	__static_initialization_an...
160	mmr_module_introduce
138	mmr_get_nr_edge
128	mmr_get_max_elem_id
127	mmr_set_max_gen_diff
125	mmr_get_max_node_max
124	mmr_get_max_edge_id
117	_GLOBAL__sub_I_mmv_...
24	mmr_get_max_face_id
12	mmr_get_max_gen

Rysunek 6.1: Zrzut ekranu z profilera cachegrind. Przedstawiona tabela podająca koszt wywołania funkcji (nazwa po prawej) w cyklach procesora (po lewej). Na samej górze funkcja `mmr_el_node_coor` odpowiadająca za realizacje operatora $\mathcal{G}_{eo}(\mathcal{W}(\mathcal{E}))$.

opcjami podglądu i analizy zebranych danych największą jego zaletą jest możliwość korzystania z liczników sprzętowych na komputerach z procesorami tej samej firmy w celu zbierania dokładnych, a nie przybliżonych danych dotyczą-

cych wykonania programu. Jest to duży pakiet i wymaga sporych zasobów do działania (np.: generuje pliki z informacjami o wykonaniu, które zajmują kilkanaście/kilkadziesiąt GB).

gprof jest standardowy narzędziem do profilowania na systemach Unix i Linux, którego wsparcie jest wbudowane w kompilator (gcc). Działa w trybie wbudowanym tzn. instrukcje zbierające informacje o wykonaniu są wkompilowane w wykonywany program. W czasie działania programu jest generowany plik binary zawierający informacje diagnostyczne, a sam program gprof jest używany do przeglądania i analizy tego pliku. Istnieją również różne interfejsy graficzne dostosowane do prezentowania zebranych informacji w postaci map,grafów itp.

callgrind jest narzędziem należącym do programu valgirnd, służącym do mapowania i zliczania wywołań funkcji wraz z czasem ich trwania. Analizowany program jest wykonywany jako pod-proces programu valgirnd, który przejmuje rolę pośrednika pomiędzy programem, a system operacyjnym. Pozwala to na kontrolę i analizę zachowania programu. Efektem ubocznym jest bardzo duże spowolnienie wykonywania programu, ponieważ wszystkie instrukcje są przechwytywane i analizowane. W związku z tym rozwiązanie to jest bardzo nieefektywne przy badaniu zachowania dużych programów lub dużych zadań obliczeniowych.

cachegrind również jest narzędziem należącym do programu valgrind i działa na tej samej zasadzie co callgrind. Różnicą jest to, że cachegrind analizuje wykorzystanie pamięci cache, w tym zlicza stosunek trafień i chybień dla różnych poziomów tej pamięci, wskazując instrukcje w programie, charakteryzujące się niewydajnym korzystaniem z pamięci.

Przykładowy rezultat profilowania można zobaczyć na rysunkach 6.2, 6.1. Narzędzia dostarczyły dane w postaci czasów i procentowych udziałów dla poszczególnych metod w całkowitym czasie trwania symulacji. Następnie na ich podstawie zidentyfikowano metody, które w praktyce odpowiadają omówionym w pkt 4.1.2 operatorom realizującym zarządzanie siatką. W konsekwencji zidentyfikowania istotnych operatorów przeprowadzono dalsze analizy poprzez umieszczenie w kodach źródłowych bezpośrednich pomiarów czasu dla tych operatorów. Efektem przeprowadzonych analiz drzew wywołania oraz czasów dostarczonych przez programy profilujace, oraz innych jawnych pomiarów czasów różnych przypadków w kodzie źródłowym jest następujący wniosek: metodą siatki, która ma największy wpływ na całkowity czas działania programu (w sensie procentowym) niezależnie od docelowej architektury obliczeniowej jest ta odpowiadająca operatorowi pobierania współrzędnych węzłów dla elementu skończonego $\mathcal{G}_{eo}(\mathcal{W}(E))$.

6.1.1. Wnioski z profilowania

W wyniku otrzymanych danych została przeprowadzona dalsza analiza, które wskazała, że:

— Największym problemem optymalizacyjnym w metodzie odpowiadającej operatorowi $\mathcal{G}_{eo}(\mathcal{W}(Element))$ jest kwestia dostępu do pamięci. Wywołanie tej metody wymaga zapisania w podanej tablicy współrzędnych wierzchołków, które praktycznie zawsze nie sąsiadują ze sobą w pamięci. Co więcej, im większa siatka, tym maksymalna odległość między wierzchołkami i ich współrzędnymi staje się większa i choć można tę sytuację poprawiać poprzez przenumerowanie węzłów i inne techniki (np. indeksowanie za pomocą krzywych wypełniających przestrzeń), to nie da się wyeliminować tego problemu.

— Kolejnym problemem optymalizacyjnym jest zbytnia obiektowość kodu obliczeniowego. Paradoksalnie, uznanie obiektów siatki (elementy, ściany, krawędzie, wierzchołki), za osobne klasy obiektów sprawdza się bardzo dobrze w pracach nad programem, ale sprawdza się źle w przypadku kiedy program już działa. Wywoływanie dla każdego elementu osobno metody realizującej operator $\mathcal{G}_{eo}(\mathcal{W}(Element))$ w dość dużej odległości czasowej (jak na standardy procesora tj. więcej niż $10^{-8}s$) pociąga za sobą konieczność ciągłego na nowo ściągania do procesora danych np.: o współrzędnych i co gorsza - wielokrotnie tych samych danych, co bardzo obniża wskaźnik intensywności obliczeniowej AI.

Powyższe wnioski oznaczają, że w praktyce statystycznie mając n_{el} elementów, z których każdy ma vt wierzchołków, w sumie wszystkie elementy przechowują informację o $n_{el} \cdot vt$ wierzchołkach. Każdy wierzchołek ma trzy współrzędne podwójnej precyzji (8 bajtów każda), zatem musimy pobrać $n_{el} * n_{el}vt * 3 * 8B$ danych do procesora. Co dla przykładowej siatki czworościennej mającej 10 tysięcy elementów wynosi 960kB danych żądanych przez procesor, choć same współrzędne zajmują w pamięci średnio 40kB. Warto tutaj zauważyć, że patrząc na problem pod kątem specyfikacji sprzętu, to wielkość pamięci cache procesora w architekturze SandyBridge ma rozmiar 64kB dla poziomu L1, a dla poziomu L2 256kB na rdzeń, co oznacza, że różnica w ilości danych ma tutaj istotne znaczenie. Jednocześnie, mamy pewność, że dane te nie będą liniowo ułożone w pamięci. W praktyce można zauważyć, że w najlepszym statystycznie przypadku wierzchołki elementu są zlokalizowane wokół dwóch adresów w pamięci. Można by zatem założyć, że kiedy iterujemy po elementach dużej siatki w sposób rozsądny (dobra numeracja + dobry przydział pamięci), to połowa potrzebnych danych jest już w L2 lub L3 (na pewno nie w L1, ponieważ pomiędzy punktami wołania operatora $\mathcal{G}_{eo}(\mathcal{W}(Element))$ istnieje dużo innych operacji, które zastąpią w pamięci cache L1 linie z danymi pobranymi wcześniej). Co oznacza, że dla 40kB danych o współrzędnych potrzebujemy pobrać tak naprawdę w

sumie 480kB, co daje współczynnik wykorzystania danych pobranych do procesora na poziomie

$$AI(\mathcal{G}_{eo}(\mathcal{W}(Element))) = \frac{1}{10}, \tag{6.1}$$

co skutecznie uniemożliwia osiągnięcie w tym przypadku wysokiej wydajności.

6.2. Testy syntetyczne operatorów adaptacji dla węzła obliczeniowego

W celu pełniejszego wykorzystania możliwości procesorów wielordzeniowych został opracowany model adaptacji wielowątkowej. U podstaw tego opracowania leżą koncepcje opisywane już w innych miejscach [18, 19] w szczególności model zarządzania pamięcią omówiony wcześniej w rozdziale 5. Został opracowany schemat zarządzania bankami pamięci w taki sposób, by nie blokować wątków tworzących nowe obiekty. W celu zweryfikowania tego modelu zostały opracowane i przeprowadzone testy. W ramach testów porównywano najlepszy opracowany działający kod, z następującymi mechanizmami synchronizacji:

— Sekcja krytyczna (critical) – standardowy, najwolniejszy, najbezpieczniejszy sposób synchronizacji. Sekcja krytyczna jest rozwiązaniem blokującym.
— Operacje atomowe (konkretnie: *omp atomic capture : x=b+=c;*) – zastosowanie tego rozwiązania wymagało wprowadzenia pewnych modyfikacji w kodzie, a powinno być znacznie szybsze, ponieważ na architekturze testowej (procesor Intel Core i7 siódmej generacji) ta operacja atomowa jest realizowana na poziomie jednej instrukcji procesora.
— Transakcyjność. Kod został tak skonstruowany, że niezależnie od sposobu synchronizacji, ich ilość została bardzo drastycznie zredukowana (praktycznie do 3-4 operacji atomowych na wątek na jeden krok adaptacji). W związku z tym można było przyjąć dość ciekawe założenie, że statystycznie szansa na spowodowanie błędu jest bardzo mała, a do tego błąd ten może być sprawdzony i poprawiony (podobne podejście jest używane np.: w pewnych sytuacjach w jądrach systemów operacyjnych), co wymaga sprawdzenia sumy kontrolnej i wypadku wykrycia błędu należy dokonać wycofania błędnych zmian i ponownego wprowadzenia poprawnych wartości.

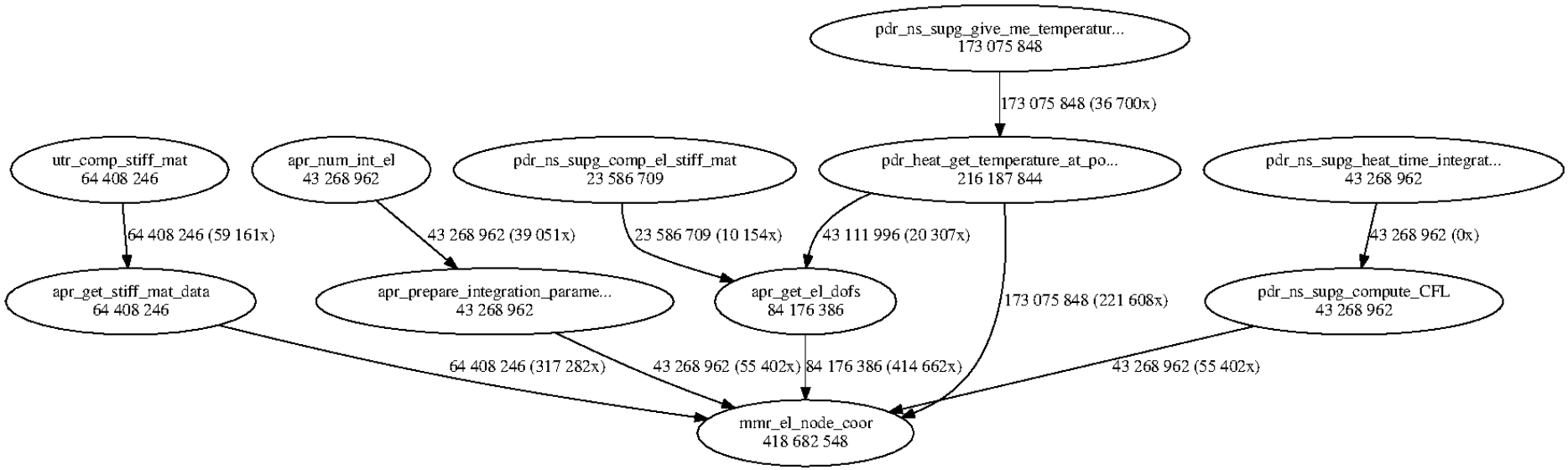
W wyniku testów odkryto pewne ciekawe prawidłowości. Podsumowanie testów znajduje się na rysunkach 6.3, 6.4 i 6.5.

6.2.1. Wnioski z testów syntetycznych

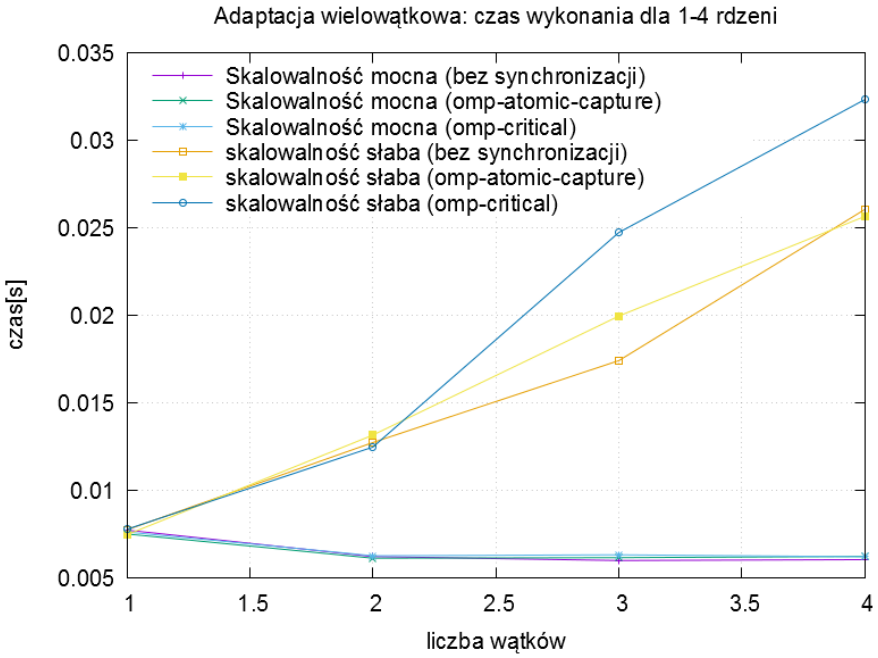
Wyniki testów są częściowo zaskakujące. Otóż w przypadku problemu, jakim jest adaptacja, efektywność zrównoleglenia tej operacji jest zbieżna, praktycznie do tej samej wartości wynoszącej w przybliżeniu 0.35 (rysunek 6.5), z wyjątkiem jedynie zauważalnie gorszego zachowania w przypadku z sekcją krytyczną, gdzie efektywność zrównoleglenia schodzi do poziomu ok 0.31 (rysunek 6.5). Należy przy tym zauważyć, że granicą testu było 10^6 elementów do zaadaptowania, czyli ok $9 \cdot 10^6$ po adaptacji co stanowi odpowiednie obciążenie, uwzględniając model z pamięcią wspólną przeznaczony na jeden węzeł. Jeżeli przyjrzymy się dokładnie wynikom, to:

— Efektywność operacji atomowej i jej braku jest praktycznie taka sama (rysunek 6.5).
— Sekcja krytyczna ma gorsze wyniki czasowe niż dwa pozostałe sposoby synchronizacji (rysunek 6.3).
— Użycie operacji atomowych zostało poprawnie zaprojektowane i wykonane (rysunek 6.5, 6.4).
— Program w opcji bez synchronizacji nie spowodował błędu ani razu (testy były w sumie powtarzane kilka tysięcy razy).

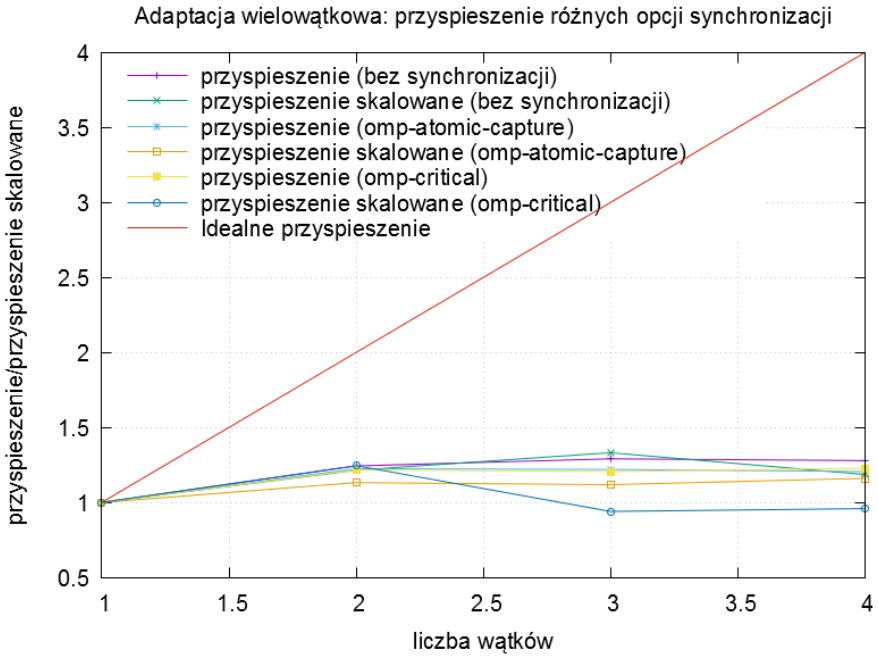
Przyglądając się wynikom, należy zauważyć, że dobór opcji synchronizacji ma *pewne znaczenie*, ale w ramach ściśle określonych granic wyznaczanych przez wykazaną efektywność. Wynika z tego, że problem synchronizacji w tym konkretnym przypadku jest *wysoce drugorzędny* w kwestii opracowania kodu adaptacji *wysokiej wydajności*. Można postawić wniosek, że dopóki nie zmniejszy się ilości danych, ich lokalności i objętości (np.: kosztem większej ilości obliczeń w siatce) opracowywanie nowych sposób realizacji wielowątkowej ma charakter drugorzędny względem wydajności dostępu do danych. W wyniku analizy teoretycznej, praktycznej i doświadczalnej doszedłem do wniosku, że problemem są zdecydowanie rozproszone dostępy do pamięci (punkt 6.1), co stało się jednym z powodów rozważania kwestii zminimalizowania ilości danych przez kompresję.



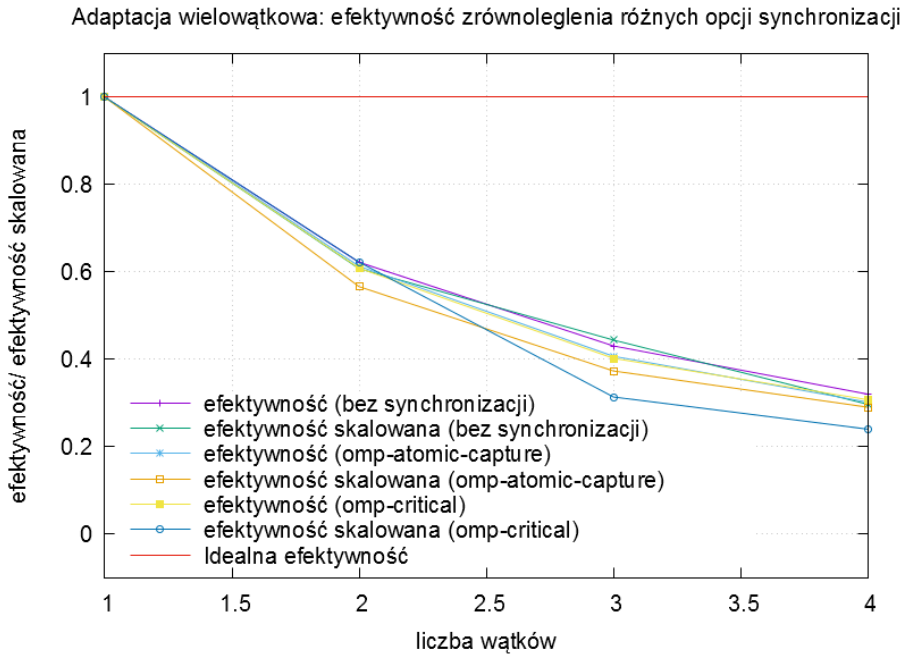
Rysunek 6.2: Graf z profilera callgrind. Na samym dole odpowiadająca operatorowi $\mathcal{G}_{eo}(\mathcal{W}(\mathcal{E}))$ funkcja `mmr_el_node_coor` i liczba jej wywołań: 418 682 548. Następnie powyżej funkcje wywołujace powyższą, wraz z krotnością. Warto zwrócić uwagę na lewy górny róg grafu, gdzie widać fragment grafu odpowiadający za obliczenie elementowej macierzy dla budowania globalnego układu równań. W sumie metoda `utr_comp_stiff_mat` jest wołana 59 161 razy, a następnie woła 64 408 246 razy metody faktycznie wołające pobieranie wartości współrzędnych geometrycznych w wierzchołkach elementu.



Rysunek 6.3: Wykres czasu trwania adaptacji dla 1M elementów w zależności od wątków.



Rysunek 6.4: Wykres przyspieszenia adaptacji dla 1M elementów w zależności od wątków.



Rysunek 6.5: Wykres efektywności przyspieszenia adaptacji dla 1M elementów w zależności od liczby wątków.

6.3. Kompresja współrzędnych geometrycznych siatki.

Z badań i analiz opisanych dotychczas w tej pracy wynika, że istnieje jeden szczególny operator, który ma bardzo duże znaczenie dla ogólnej wydajności schematu zarządzania siatką. Jest to operator pobierania współrzędnych geometrycznych dla wszystkich wierzchołków należących do pewnego elementu skończonego. Korzystając z definicji omówionych w rozdziałach 3 oraz 4 można go ogólnie zapisać jako:

$$\mathcal{G}_{eo}(\mathcal{W}(E)) \rightarrow \{x_1, y_1, z_1, ..., x_n, y_n, z_n\}, \quad E \in \mathcal{E} \wedge x_i, y_i, z_i \in \mathbb{R}, i \in \mathbb{N} \tag{6.2}$$

Chociaż operator sam w sobie jest raczej prostym i intuicyjnym operatorem siatki, to wysoka częstotliwość $f_{\mathcal{G}_{eo}(\mathcal{W}(E))}$ jego występowania w każdym z badanych przypadków (np. rys. 6.1, 6.2) oraz wykazana w pkt. 6.1.1 niska intensywność obliczeniowa AI realizacji tego operatora, powoduje, iż ma on fundamentalne znaczenie dla wydajności. Wnioski wynikające z badań i analiz w pkt. 6.1 oraz analiza literatury omówionej w rozdziale 1.4, m.in. prac [116, 48, 31, 28, 27] pozwalają nam sformułować następujące obserwacje:

1. Najbardziej pożądaną sytuacją jest, gdy dla danej siatki M używając reprezentacji P, można zdefiniować porządki liniowe dla elementów $(\mathcal{E}, \preceq)$ i wierzchołków $(\mathcal{W}, \preceq)$ takie, że :
$$AI(\mathcal{G}_{eo}(\mathcal{W}(E))) \rightarrow \frac{|\mathcal{E}|}{|\mathcal{W}|} \tag{6.3}$$
albo przynajmniej
$$AI(\mathcal{G}_{eo}(\mathcal{W}(E))) \rightarrow 1 \tag{6.4}$$
w praktyce oznacza to, że każda współrzędna punktu pobrana do pamięci *cache* procesora z pamięci głównej została wykorzystana do obliczeń przynajmniej raz (wzór 6.4), a najlepiej średnio tyle razy, do ilu elementów należy, co stanowi górne ograniczenie intensywności obliczeniowej AI w przypadku tego operatora siatki (wzór 6.3). Jednocześnie z omówionych wcześniej badań wynika, że współczynnik ten przyjmował wartość $AI(\mathcal{G}_{eo}(\mathcal{W}(E))) \rightarrow 0.1$ (wzór 6.1).
2. Nie istnieje algorytm przenumerowania elementów i węzłów siatki obliczeniowej dla arbitralnie dobranej siatki M, który pozwala iterować po węzłach elementów w sposób liniowy z zachowaniem cech 6.3, 6.4. Rozważano tutaj także krzywe wypełniające przestrzeń *space-filling-curves* i ich implementację w postaci kodu Mortona [11, 22].
3. W siatkach obliczeniowych można wyróżnić dwa rodzaje wierzchołków: brzegowe i wewnętrzne. Można zwrócić uwagę, że dla rozważanych w tej pracy siatek z węzłami stacjonarnymi, wierzchołki zewnętrzne (brzegowe) zawierają informację o geometrii obszaru, natomiast wewnętrzne nie, ponieważ tylko definiują ele-

menty w obszarze otoczonym brzegiem. Przy czym położenie tych pierwszych jest istotne dla definicji zagadnienia początkowo-brzegowego, ale w praktyce jest otrzymywane z dokładnością stosunkowo niskiego rzędu (10^{-3} względem wymiarów siatki) , co wynika z praktycznych aspektów sposobu odwzorowania rzeczywistości (rysunek techniczny, zdjęcia, niedokładność wytwarzania badanych elementów) oraz istotnych ograniczeń związanych z fizyką modelowanych zjawisk (rozszerzalność cieplna, wpływ ciśnienia etc.). Położenie węzłów wewnętrznych ma wpływ na jakość elementów skończonych, a co za tym idzie dokładność numeryczną (w szczególności wymiar charakterystyczny h elementu skończonego), ale poza tym jest arbitralne i może być zmieniane dopóty, dopóki jakość elementów jest zachowana (co jest wykorzystywane np.: w r-adaptacji) lub w celu zmniejszenia współczynnika h mającego wpływ na błąd rozwiązania (h-adaptacjia i de-adaptacja, r-adaptacja).

4. Zazwyczaj współrzędne punktów w siatkach nieregularnych są przechowywane jako liczby zmiennoprzecinkowe, a decyzja o pojedynczej lub podwójnej precyzji jest uzależniona od sposobu reprezentowania wartości stopni swobody, tak by operacje arytmetyczne były dokonywane na tej samej reprezentacji liczbowej, bez obciążania procesora konwersjami typów. W rzeczywistości przechowywanie położenia punktów z dokładnością rzędu 10^{-6} (dokładność pojedynczej prezycji) albo 10^{-15} (dokładność podwójnej prezycji) względem faktycznego wymiaru siatki dla danej geometrii jest nadmiarowe.
5. Dokładność opisu geometrii leży u założeń stosowalności metody elementów skończonych jako zasada ciągłości fizycznej modelowanego ośrodka. Z geometrycznego punktu widzenia istotne rozróżnianie punktów z już na poziomie jednej milionowej względem rozmiaru badanego ośrodka wykracza poza to założenie i jeżeli zjawisko wymaga takiej dokładności, to wymaga również zastosowania modelowania wieloskalowego [44].
6. W siatce h-adaptacyjnej nowo powstające wierzchołki mają współrzędne, które są całkowicie zależne od położenia współrzędnych wierzchołków macierzystych obiektów siatki, co w praktyce oznacza, że współrzędne każdego powstałego w wyniku adaptacji wierzchołka można wyrazić pewną funkcją, której argumentami są wierzchołki macierzyste.

6.3.1. Opis formalny

Dana jest siatka M, korzystającą z reprezentacji P, dla której zdefiniowane są:

1. Operator identyfikatora obiektu topologicznego $Guid(T)$ (wzór 4.29).
2. Operator dostępu do wierzchołków elementu $\mathcal{W}(E)$ (tabela 4.1).
3. Operator dostępu do współrzędnych wierzchołka $\mathcal{G}_{eo}(V)$ (wzór 4.18).

Można zdefiniować odwzorowanie, które współrzędnym każdego punktu przypisze pewną arbitralnie wybraną liczbę

$$C(\cdot) : \mathbb{R}^3 \to \mathbb{N}^+ \tag{6.5}$$

co jest wykorzystywane m.in. w definicjach krzywych wypełniających przestrzeń. Następnie możemy zdefiniować odwzorowanie odwrotne do $C(\cdot)$:

$$C_M^{-1} = D(\cdot)_M : \mathbb{N}^+ \to \mathbb{R}^3 \tag{6.6}$$

Ponieważ w siatce M zbiór punktów $\boldsymbol{\mathcal{W}}_M \subset \mathbb{R}^3$ jest określony i skończony ($n_{max} = |\{\mathcal{W}_M\}|$), możemy zapisać, że dla tej siatki:

$$\begin{aligned} C_M(\cdot) :& \boldsymbol{\mathcal{W}}_M \to \mathbb{N}_M \\ C^{-1} = D(\cdot) :& \mathbb{N}_M \to \boldsymbol{\mathcal{W}}_M, \\ & gdzie \qquad \mathbb{N}_M \subset \mathbb{N}^+ \wedge |\mathbb{N}_M| = n_{max} \wedge \max \mathbb{N}_M = n_{max} \end{aligned} \tag{6.7}$$

Następnie możemy przyjąć, że tak długo, jak C_M jest relacją odwrotną do D_M, relacja ta jest bijekcją między dwoma równolicznymi zbiorami:

$$\forall w \in \boldsymbol{\mathcal{W}}_M \quad Guid_M(w) = n \quad \Leftrightarrow \quad C_M(\mathcal{G}_{eo}(w)) = n \quad \Leftrightarrow \quad D_M(Guid_M(w)) = \mathcal{G}_{eo}(w) \tag{6.8}$$

W ten sposób otrzymaliśmy relację, która dla konkretnej siatki dokonuje projekcji dyskretnego podzbioru przestrzeni trójwymiarowej na przestrzeń jednowymiarową i odwrotnie. Następnie można zauważyć, że zależności 6.7 będą również spełnione dla dowolnego innego n_{max} spełniającego warunek

$$n_{max} \geqslant |\boldsymbol{\mathcal{W}}_M| \tag{6.9}$$

Zwróćmy też uwagę, że dla konkretnej siatki M, zbiór geometrycznych punktów $\boldsymbol{\mathcal{W}}_M$ mieści się w niewielkim podzbiorze przestrzeni trójwymiarowej $\boldsymbol{G_M}$ będącym iloczynem kartezjańskim przedziałów zawierających współrzędne wszystkich punktów siatki M.

$$\boldsymbol{G_M} \subset \mathbb{R}^3 : \forall w \in \boldsymbol{\mathcal{W}}_M \quad \mathcal{G}_{eo}(w) \in \boldsymbol{G_M} \tag{6.10}$$

Można więc zdefiniować dyskretny podzbiór $\boldsymbol{G_M^D}$ przestrzeni $\boldsymbol{G_M}$, posiadający przynajmniej n_{max} punktów, wśród których znajdują się wszystkie współrzędne wierzchołków zawartych w siatce M:

$$\begin{aligned} \boldsymbol{G_M^D} \subset \boldsymbol{G_M} \subset \mathbb{R}^3 : \quad & |\boldsymbol{G_M^D}| \geqslant |\boldsymbol{\mathcal{W}}_M| \\ & \wedge \forall w \in \boldsymbol{\mathcal{W}}_M \quad \mathcal{G}_{eo}(w) \in \boldsymbol{G_M^D} \end{aligned} \tag{6.11}$$

Zauważmy, że w ramach zbioru $\boldsymbol{G_M^D}$ możemy określić punkt minimalny $\boldsymbol{G_M^D} \ni g_{min} = [min_x, min_y, min_z]$ i maksymalny $\boldsymbol{G_M^D} \ni g_{max} = [max_x, max_y, max_z]$ tego zbioru. Wykorzystując podzbiór niedyskretny $\boldsymbol{G_M} \subset \mathbb{R}^3$, możemy w jego obrębie zdefiniować inny podzbiór dyskretny $\boldsymbol{P_M}$ zawierający $m \geqslant n_{max}$ punktów taki, że:

$$\begin{aligned}
&\boldsymbol{P_M} \subset \boldsymbol{G_M^D} \subset \boldsymbol{G_M} \subset \mathbb{R}^3 : \\
&\qquad\qquad |\boldsymbol{P_M}| = m, \quad m \gg n_{max} \\
&\qquad\qquad \wedge \forall p \in P_M \\
&\qquad\qquad p = [x_p, y_p, z_p] = [min_x + d_x \cdot i_x, min_y + d_y \cdot i_y, min_z + d_z \cdot i_z] \\
&\qquad gdzie \qquad x_p, y_p, z_p, min_x, min_y, min_z, d_x, d_y, d_z \in \mathbb{R} \\
&\qquad\qquad i_x, i_y, i_z \in \mathbb{N}
\end{aligned} \tag{6.12}$$

Zauważmy, że m możemy zapisać w postaci

$$m = m_x \cdot m_y \cdot m_z, \qquad m, m_x, m_y, m_z \in \mathbb{N}^+ \wedge m \geqslant m_x, m_y, m_z \tag{6.13}$$

I wtedy wartości mnożników naturalnych i_x, i_y, i_z określić jako

$$\begin{aligned}
i_x &\in \{0, ..., m_x\} \\
i_y &\in \{0, ..., m_y\} \\
i_z &\in \{0, ..., m_z\}
\end{aligned} \tag{6.14}$$

A dotychczas nieznane wartości d_x, d_y, d_z można zdefiniować jako:

$$\begin{aligned}
d_x, d_y, d_z &\in \mathbb{R}^+ \\
d_x &= (max_x - min_x)/m_x \\
d_y &= (max_y - min_y)/m_y \\
d_z &= (max_z - min_z)/m_z
\end{aligned} \tag{6.15}$$

Warto zwrócić uwagę, że dla wartości granicznych $i_0 = [0, 0, 0]$ oraz $i_m = [m_x, m_y, m_z]$ otrzymujemy odpowiednio elementy minimalne i maksymalne dla zbioru $\boldsymbol{P_M}$, które są nimi także dla zbioru $\boldsymbol{G_M^D}$:

$$\begin{aligned}
[x_0, y_0, z_0] &= [min_x + d_x \cdot 0, min_y + d_y \cdot 0, min_z + d_z \cdot 0] = g_{min} \\
[x_m, y_m, z_m] &= [min_x + d_x \cdot m_x, min_y + d_y \cdot m_y, min_z + d_z \cdot m_z] = g_{max}
\end{aligned} \tag{6.16}$$

Posiadając tak zdefiniowane dwa zbiory dyskretne $\boldsymbol{G_M^D}$ oraz $\boldsymbol{P_M}$ możemy określić relację, która każdemu elementowi zbioru $\boldsymbol{G_M^D}$ przypisze pewien element zbioru

$\boldsymbol{P_M}$. Ponieważ $|\boldsymbol{G_M^D}| >> |\boldsymbol{P_M}|$ istnieje wiele punktów ze zbioru $\boldsymbol{P_M}$, które można przypisać jednemu punktowi ze zbioru $\boldsymbol{G_M^D}$. Rozważmy zatem dwa przypadki:

1. Dla punktu ze zbioru $\boldsymbol{G_M^D}$ istnieje dokładny odpowiednik w zbiorze $\boldsymbol{P_M}$, więc zgodnie z 6.12:

$$\begin{aligned} &\exists![i_x^p, i_y^p, i_z^p] : [g_x, g_y, g_z] = [min_x + d_x \cdot i_x^p, min_y + d_y \cdot i_y^p, min_z + d_z \cdot i_z^p] \\ &\quad gdzie \qquad [g_x, g_y, g_z] \in \boldsymbol{G_M^D} \\ &\quad gdzie \qquad i_x^p, i_y^p, i_z^p \in \mathbb{N} \end{aligned} \tag{6.17}$$

   Zwróćmy więc uwagę, że korzystając z 6.7, 6.8 i 6.12 dla siatki M określającej przestrzeń współrzędnych $\boldsymbol{G_M^D}$ oraz określonym dla niej podzbiorze dyskretnym $\boldsymbol{P_M}$ można zdefiniować odwzorowanie:

$$\begin{aligned} &\forall w \in \boldsymbol{\mathcal{W}}_M \wedge \mathcal{G}_{eo}(w) \in \boldsymbol{P_M} \\ &\qquad Guid_M^P(w) = n_p \\ &\qquad n_p = i_x^p \cdot \frac{m}{m_x} + i_y^p \cdot \frac{m}{m_y} + i_z^p \cdot \frac{m}{m_z} \end{aligned} \tag{6.18}$$

   dla którego n_p jest kombinacją liniową, zależącą dla stałej siatki tylko od wartości i_x^p, i_y^p, i_z^p, ponieważ wyrażenia $\frac{m}{m_x}, \frac{m}{m_y}, \frac{m}{m_z}$ są niezmienne dla każdego punktu w M. Dalej analogicznie jak w 6.8:

$$\begin{aligned} &\forall w \in \boldsymbol{\mathcal{W}}_M \wedge \mathcal{G}_{eo}(w) \in \boldsymbol{P_M} \\ &\qquad C_M^P(\mathcal{G}_{eo}(w)) = n_p = i_x^p \cdot \frac{m}{m_x} + i_y^p \cdot \frac{m}{m_y} + i_z^p \cdot \frac{m}{m_z} \\ &\qquad\qquad \Leftrightarrow \\ &\qquad D_M(Guid_M^D(w)) = \mathcal{G}_{eo}(w) \\ &\qquad\qquad \Leftrightarrow \\ &\qquad D_M^P(i_x^p, i_y^p, i_z^p) = \mathcal{G}_{eo}(w) \end{aligned} \tag{6.19}$$

2. Dla punktu ze zbioru $\boldsymbol{G_M^D}$ nie istnieje dokładny odpowiednik w zbiorze $\boldsymbol{P_M}$, możemy zatem policzyć odległość od punktu $g \in \boldsymbol{G_M^D}$ do najbliższego punktu $p \in \boldsymbol{P_M}$. Zwróćmy uwagę, że z definicji $\boldsymbol{P_M}$ (6.12) wynika, że maksymalna odległość pomiędzy dwoma punktami w $\boldsymbol{P_M}$ wynosi $\sqrt{d_x^2 + d_y^2 + d_z^2}$, zatem skoro wiemy, że g nie odpowiada żadnemu z punktów, to znaczy, że musi być gdzieś między nimi, zatem maksymalna odległość punktu g od punktu p wynosi $\sqrt{d_x^2 + d_y^2 + d_z^2}/2$. Stosując analogiczne rozumowanie jak w 6.17 - 6.19 otrzymujemy:

$$
\begin{aligned}
&\forall w \in \boldsymbol{\mathcal{W}}_M \wedge \mathcal{G}_{eo}(w) \notin \boldsymbol{P_M} \\
&\qquad C_M^P(\mathcal{G}_{eo}(w)) = n_p = i_x^p \cdot \frac{m}{m_x} + i_y^p \cdot \frac{m}{m_y} + i_z^p \cdot \frac{m}{m_z} \\
&\qquad\qquad \Leftrightarrow \\
&\qquad D_M(Guid_M^D(w)) = \mathcal{G}_{eo}(w) + c_{err} \\
&\qquad\qquad \Leftrightarrow \\
&\qquad D_M^P(i_x^p, i_y^p, i_z^p) = \mathcal{G}_{eo}(w) + c_{err} \\
&gdzie \qquad c_{err} \leqslant \frac{\sqrt{d_x^2 + d_y^2 + d_z^2}}{2} \\
&\qquad\qquad oraz \lim_{m \to \inf} c_{err} = 0
\end{aligned}
\tag{6.20}
$$

W efekcie dla siatki M posiadającej zbiór punktów $\boldsymbol{\mathcal{W}}_M$ otrzymaliśmy operator C_M^P przypisujący każdemu punktowi tej siatki liczbę naturalną oraz odwrotny operator D_M^P odtwarzający na podstawie tej liczby współrzędne tego punktu z dokładnością co najmniej c_{err}. Oba te operatory są w sumie zależne od 12 parametrów, z których 9 jest wspólnych dla wszystkich punktów w siatce.

W szczególności można na przykład przyjąć, że – w zależności od ilości punktów w siatce obliczeniowej – punktów dyskretnych w przestrzeni $\boldsymbol{P_M}$ będzie np.: $m_{32} = 4,294967296 \cdot 10^9$ (czyli tyle ile można rozróżnić, stosując liczbę całkowitą w zapisie 32 bitowym) lub też $m_{64} = 18,44674407 \cdot 10^{18}$(odpowiednio dla liczb 64 bitowych). Daje to liczbę (średnio) rzędu 10^3 lub też 10^6 punktów w każdym z kierunków. Co oznacza, że niezależnie od faktycznych współrzędnych w przestrzeni, jesteśmy w stanie odwzorować punkt z błędem wynoszącym co najwyżej $c_{err}^{32} = \frac{10^{-3}}{2}$ lub $c_{err}^{64} = \frac{10^{-6}}{2}$ względem rozpiętości siatki obliczeniowej w danym kierunku.

6.3.2. Algorytm kompresji współrzędnych

W praktycznym zastosowaniu dla siatki M korzystając z C_M^P, D_M^P opisanych formalnie w poprzednim punkcie zrealizowane zostały operatory:

$$
\begin{aligned}
\mathcal{G}_{eo}(t), &\qquad t \in \{\mathcal{W}\} \cup \{\mathcal{K}\} \cup \{\mathcal{S}\} \cup \{\mathcal{E}\} \\
Guid(t), &\qquad t \in \{\mathcal{W}\} \cup \{\mathcal{K}\} \cup \{\mathcal{S}\} \cup \{\mathcal{E}\}
\end{aligned}
\tag{6.21}
$$

Do których działania konieczne jest zaimplementowanie poniższych algorytmów:

— Tworzenie bazy dla skompresowanej reprezentacji siatki M (algorytm 6). Odpowiada określeniu przestrzeni granic dyskretnej przestrzeni rzeczywistych punk-

tów $\boldsymbol{G_M^D}$ oraz obliczeniu wszystkich współczynników koniecznych do przeprowadzenia kompresji współrzędnych.

— Kompresja współrzędnych geometrycznych (algorytm 7). Odpowiada za realizację operatora $Guid(t)$ siatki M za pomocą operatora C_M^P (wzory 6.19, 6.20).

— Dekompresja współrzędnych geometrycznych (algorytm 8). Odwrotnie do powyższego, odpowiada za realizację operatora $\mathcal{G}_{eo}(t)$ siatki M za pomocą operatora D_M^P.

6.3.3. Teoretyczna analiza wydajności

Rozważmy przypadek, w którym chcemy przejść ze standardowego zapisu punktów na omawiany zapis skompresowany. Zgodnie z modelem przedstawionym w pkt. 3.3 należy uwzględnić zarówno wydajność pamięciową, jak i obliczeniową.

Wymagania pamięciowe

W klasycznej reprezentacji współrzędne wierzchołków są przechowywane najczęściej jako 3 zmienne podwójnej precyzji po $8B$ każda, zatem dla n_{vt} potrzeba $n_{vt} \cdot 24B$ pamięci. Jednocześnie przyjmując najprostsze założenie, że siatka jest czworościenna, dla każdego z n_{el} musimy przechowywać $n_{el}vt = 4$ identyfikatorów wierzchołków, każdy zajmujący $4B$ (przy założeniu, że identyfikator to liczba 32 bitowa). Zatem przechodząc z klasycznego jawnego przechowywania współrzędnych punktów, na skompresowaną reprezentację dyskretną zmieniamy zapotrzebowanie na pamięć do przechowywania siatki w sposób przedstawiony w tabeli 6.1 i na rysunku 6.6 odpowiednio dla 32 i 64 bitowych identyfikatorów.

Złożoność obliczeniowa.

Analizując algorytmy przedstawione w pkt 6.3.2, można zwrócić uwagę, że algorytmy 6, 7 i 8 będą wykonywane dla każdego zestawu współrzędnych. Przyjmując więc klasyczną notację złożoności obliczeniowej $O(\cdot)$:

1. Złożoność obliczeniowa dla algorytmu 6 dla n punktów w siatce wymaga jednokrotnie wykonania średnio $3n$ porównań i dodatkowo ok. 15 operacji arytmetycznych co w rezultacie daje złożoność liniową $O(n)$. Wynika z tego, że intensywność obliczeniowa $AI \to 1$.
2. Dla algorytmu 7 zauważmy, że możemy pominąć początkowe przypisania, ponieważ służą one jedynie klarowności zapisu. W głównej pętli, uruchamianej dla każdego z n punktów jest trzykrotnie wykonywana operacja, ale zawiera ona zależność przenoszone w pętli klasy Read-After-Write(RAW) oraz Write-After-Write (WAW) ze względu na $IDS[n]$, więc nie ma niestety tutaj możliwości optymalizacji. Ostatecznie algorytm ten również ma złożoność $O(n)$ dla n punktów.

Algorytm 6: Algorytm tworzenia bazy dla skompresowanej reprezentacji siatki

Dane: N - ilość punktów w siatce,
$W[N][3]$ - tablica ze współrzędnymi punktów
m - ilość punktów w przestrzeni P_M
Rezultat: $M_{encoded}[3][3]$ - tablica z parametrami zakodowanej siatki

$min = max = W[1]$ // *Wyznaczanie punktów minimalnego i maksymalnego przestrzeni* P_M ;
for $n \leqslant N$ **do**
  for $i \leqslant 3$ **do**
    if $W[n][i] \leqslant min[i]$ **then**
      $min[i] = W[n][i]$
    end
    else if $W[n][i] \geqslant max[i]$ **then**
      $max[i] = W[n][i]$
    end
    ++i ;
  end
  ++n ;
end
$rozp = [0, 0, 0]$ // *Wyznaczanie współczynników bazowych rozpiętości przestrzeni* P_M ;
$n_{rozp} = 0$;
$mm = [0, 0, 0]$;
$d = [0, 0, 0]$;
for $i \leqslant 3$ **do**
  $rozp[i] = max[i] - min[i]$;
  $n_{rozp} = n_{rozp} + rozp[i]$;
end
for $i \leqslant 3$ **do**
  $rozp[i] = rozp[i]/minimum(rozp)$ // *Normalizowanie współczynników* ;
  $mm[i] = \frac{m}{n_{rozp}} rozp[i]$ // *Określanie współczynników rozpiętości* ;
  $d[i] = rozp[i]/mm[i]$ // *Określanie wektorów bazowych* ;
  ++i ;
end
$M_{encoded}[1] = min$;
$M_{encoded}[2] = d$;
$M_{encoded}[3] = mm$;

Algorytm 7: Algorytm kompresji współrzędnych siatki.

Dane: N - ilość punktów w siatce,
$W[N][3]$ - tablica ze współrzędnymi punktów
$M_{encoded}[3][3]$ - tablica z parametrami zakodowanej siatki
Rezultat: $IDS[N]$ - tablica z wygenerowanymi identyfikatorami punktów

$min = M_{encoded}[1]$;
$d = M_{encoded}[2]$;
$mm = M_{encoded}[3]$;
for $n \leqslant N$ **do**
    $IDS[n] = 0$;
    for $i \leqslant 3$ **do**
        $IDS[n] = IDS[n] + (W[n][i] - min[i])/d[i] * mm[i]$;
        ++i ;
    end
    ++n ;
end

Uwaga: w praktyce mnożenie przez współczynnik mm lub jego odwrotność wymaga jeszcze kilku dodatkowych zabiegów związanych z reprezentacją liczb na wybranym procesorze, które są tutaj pominięte jako detal techniczny. Szczegółowa realizacja jest dostępna w repozytorium [75].

Algorytm 8: Algorytm dekompresji współrzędnych siatki.

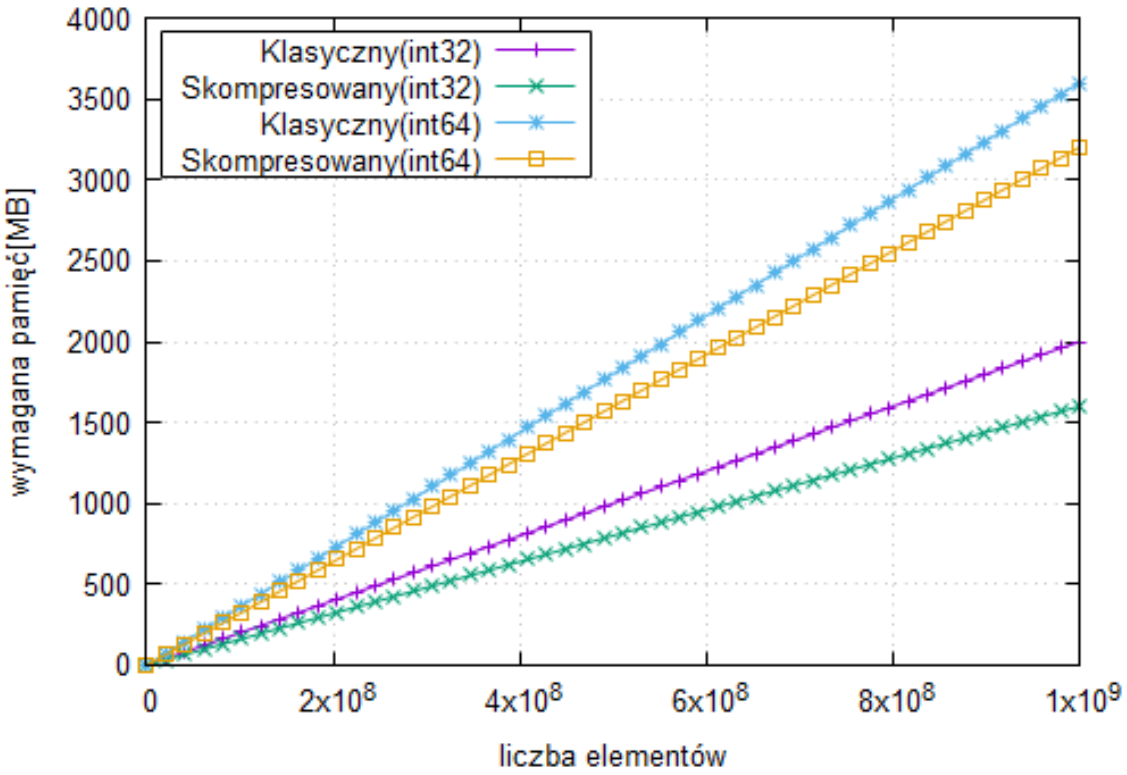
Dane: N - ilość punktów w siatce,
$IDS[N]$ - tablica z identyfikatorami punktów
$M_{encoded}[3][3]$ - tablica z parametrami zakodowanej siatki
Rezultat: $W[N][3]$ - tablica z dekompresowanymi współrzędnymi punktów

$min = M_{encoded}[1]$;
$d = M_{encoded}[2]$;
$mm = M_{encoded}[3]$;
for $n \leqslant N$ **do**
    $W[n] = [0, 0, 0]$;
    for $i \leqslant 3$ **do**
        $W[n][i] = min[i] + d[i] * (IDS[n]/mm[i])$;
        ++i ;
    end
    ++n ;
end

Uwaga: w praktyce dzieleni przez współczynnik mm lub jego odwrotność wymaga jeszcze kilku dodatkowych zabiegów związanych z reprezentacją liczb na wybranym procesorze, które są tutaj pominięte jako detal techniczny. Szczegółowa realizacja jest dostępna w repozytorium [75].

Siatka	Klasyczna	Skompresowana
int32	$(n_{vt} \cdot 24B) + (n_{el} \cdot n_{el}vt \cdot 4B)$	$48B + (n_{el} \cdot n_{el}vt \cdot 4B)$
int64	$(n_{vt} \cdot 24B) + (n_{el} \cdot n_{el}vt \cdot 8B)$	$48B + (n_{el} \cdot n_{el}vt \cdot 8B)$

Tabela 6.1: Porównanie zapotrzebowania na pamięć względem ilości elementów n_{el} dla siatki nieskompresowanej i skompresowanej. Gdzie n_{vt} - ilość wierzchołków w siatce, $n_{el}vt$ - ilość wierzchołków w pojedynczym elemencie siatki.



Rysunek 6.6: Porównanie teoretycznego zapotrzebowania na pamięć standardowej i skompresowanej siatki.

Jednocześnie warto zauważyć, że intensywność obliczeniowa w tym przypadku jest również $AI \to 1$.

3. Algorytm 8 jest analogiczny do algorytmu 7, zatem jego złożoność obliczeniowa wynosi $O(n)$, a intensywność obliczeniowa $AI \to 1$.

Należy zwrócić uwagę, że używając powyższej metody kompresji, zdefiniowaliśmy porządek liniowy spełniający warunek intensywności obliczeniowej $AI \to 1$, tak jak zostało to opisane w 6.3.

6.3.4. Testy wydajnościowe

Na rysunkach 6.7, 6.8 i 6.9 znajdują się wykresy pokazujące, jak zmienia się czas dostępu do danych dotyczących wierzchołków w zależności od ich liczby, wątków oraz

sposobu dostępu. Jako przypadek testowy wybrano prostą operację obliczania sumy współrzędnych wszystkich wierzchołków elementu. Jako sposób dostępu wybrane zostały dwa skrajne przypadki organizacji indeksów wierzchołków:

1. zgodnie z liniową kolejnością, co jest przypadkiem optymalnym dla nieskompresowanej reprezentacji, ponieważ zapewnia najlepsze wykorzystanie danych. W praktycznych siatkach obliczeniowych $3D$ przypadek ten nigdy nie występuje,
2. w losowej kolejności, co reprezentuje pesymistyczny przypadek rzeczywistej siatki obliczeniowej $3D$. W praktycznych siatkach adaptacyjnych, w zależności od przyjętych metod numeracji, przeważnie występuje *pewna niewielka* lokalność indeksowania.

Wykresy przedstawione na rysunkach 6.7 oraz 6.8 pokazują bardzo wyraźnie, że praktycznie w każdym przypadku dysproporcja pomiędzy rezultatami dla kompresji i jej braku rośnie w sposób ponad liniowy na korzyść tych pierwszych. Widać również, iż tylko w przypadku liniowej organizacji indeksów, który jest nierzeczywisty, wyniki uzyskiwane przy zwykłym zapisie są porównywalne z zapisem skompresowanym. Wariant skompresowany wykazuje praktycznie te same wyniki dla obu skrajnych przypadków organizacji indeksów wierzchołków, co wskazuje na niezależność od organizacji indeksowania. Dla każdej pary (zwykły, skompresowany) przetwarzanej przez tę samą liczbę wątków, szybsza jest reprezentacja skompresowana. Badając przyspieszenie obliczeń pokazane na rysunku 6.9 widać, iż w konkretnych przypadkach uzyskane przyspieszenie dzięki skompresowanej reprezentacji jest różne. W szczególności widać wpływ dostępu do cache procesora dla mniejszych ilości elementów, dla których pobierane są wierzchołki.

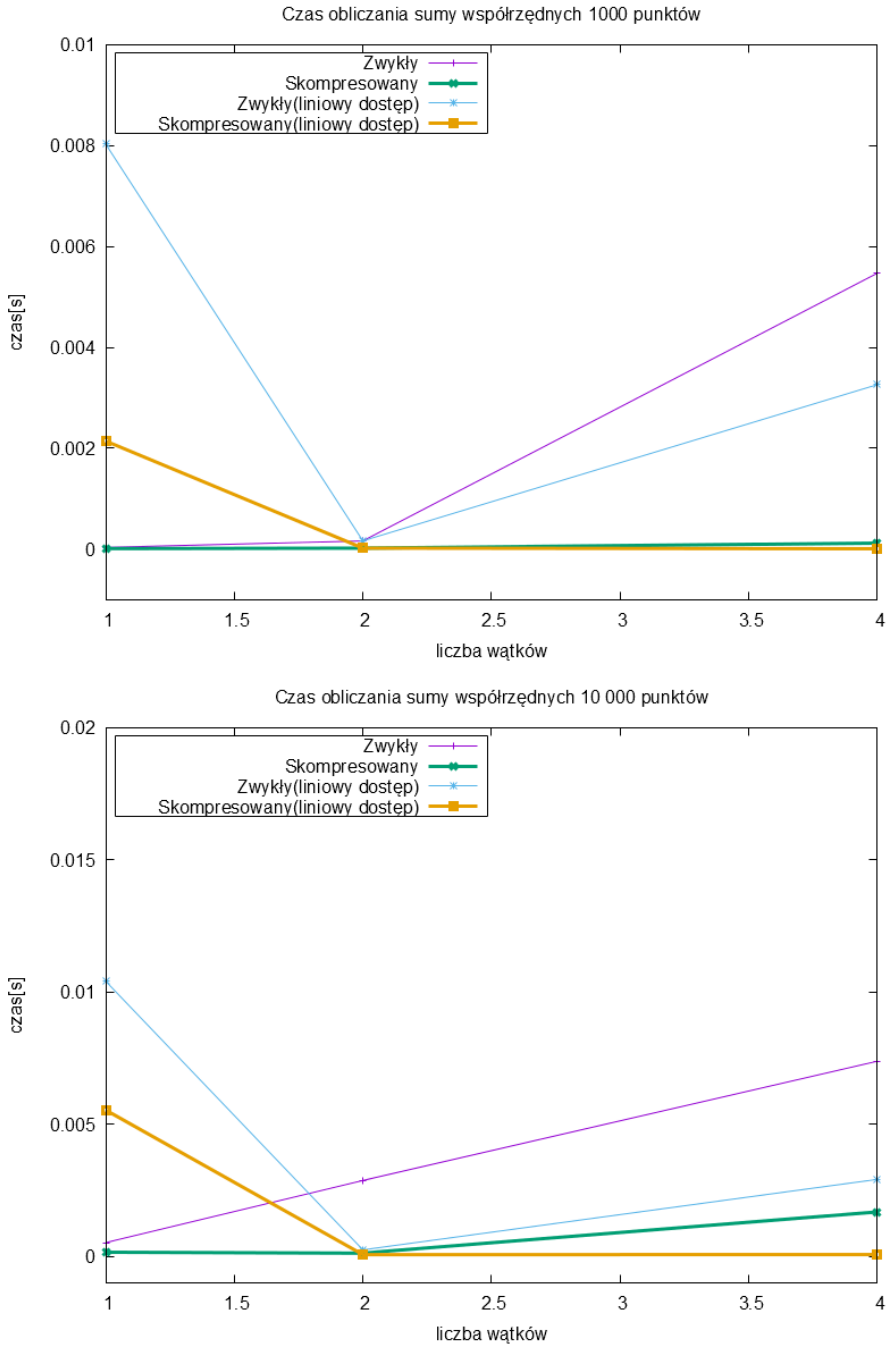
Algorytm ten pozwala bardzo efektywnie zarówno w kontekście użytej pamięci, jak i czasu obliczeń uzyskiwać dane dotyczące współrzędnych wierzchołków, szczególnie w przypadku, gdy dane miałyby leżeć w odległych obszarach pamięci.

Z analizy uzyskanych wyników można wyciągnąć następujące wnioski:

— niezależnie od wybranego przypadku, z uzyskanych wyników wynika, iż siatka skompresowana jest zawsze szybsza w dostarczaniu informacji o współrzędnych wierzchołków w elemencie, niż standardowe przechowywanie ich w pamięci (rysunek 6.7),
— wraz ze wzrostem ilości węzłów czas rozwiązanie zmniejsza się o ponad dziesięcikrotnie względem standardowego podejścia (rysunek 6.8),
— zastosowane rozwiązanie jest skalowalne, co widać na rysunku 6.9.

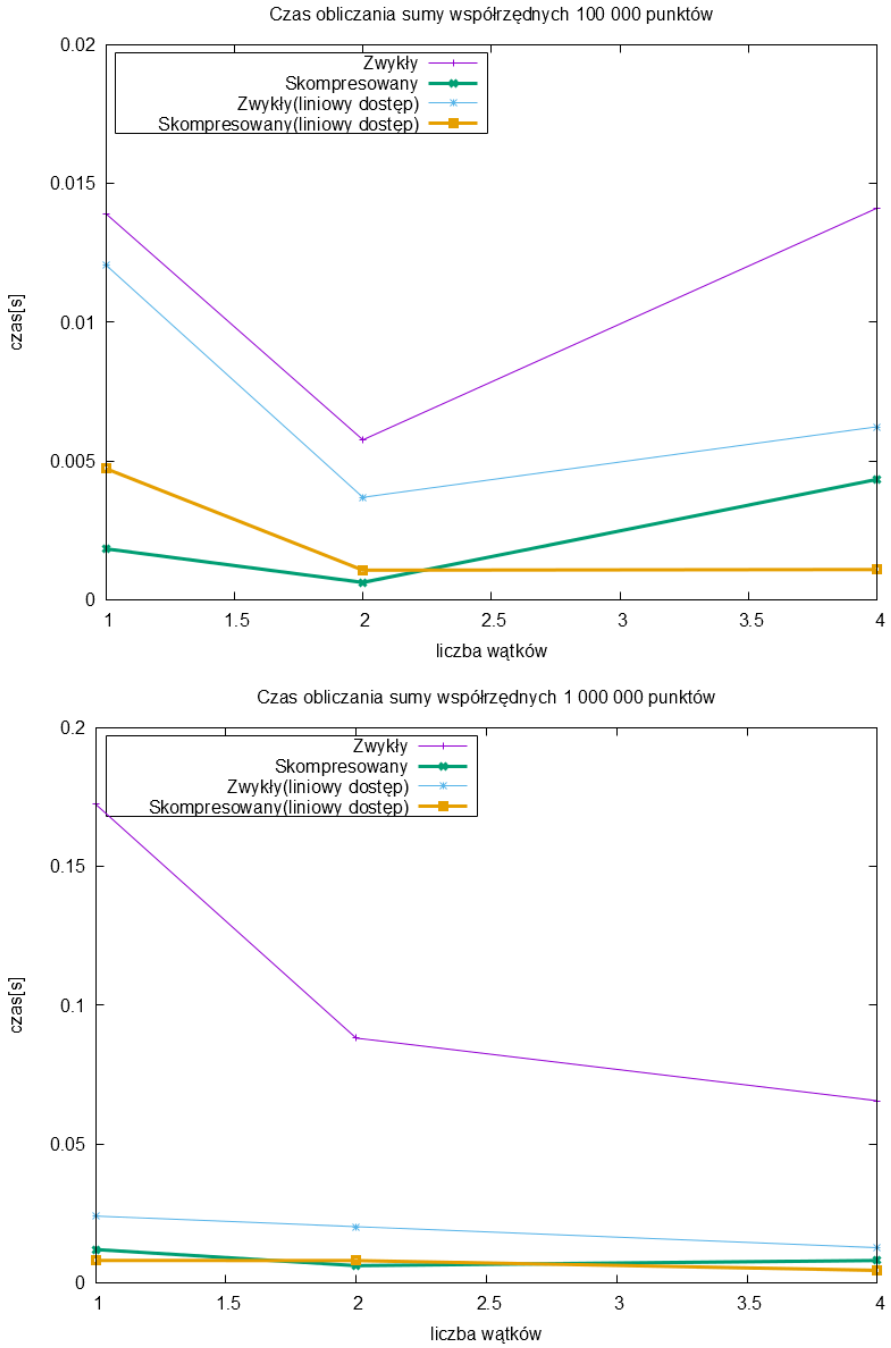
6.3.5. Porównanie z krzywymi wypełniającymi przestrzeń.

Z analizy literatury (rozdział 1.4) wynika, że podobną funkcjonalność do omawianego rozwiązania dostarcza wykorzystanie krzywych wypełniających przestrzeń, w



Rysunek 6.7: Czasy uzyskane w zadaniu testowym dla 1-10 tysięcy elementów.

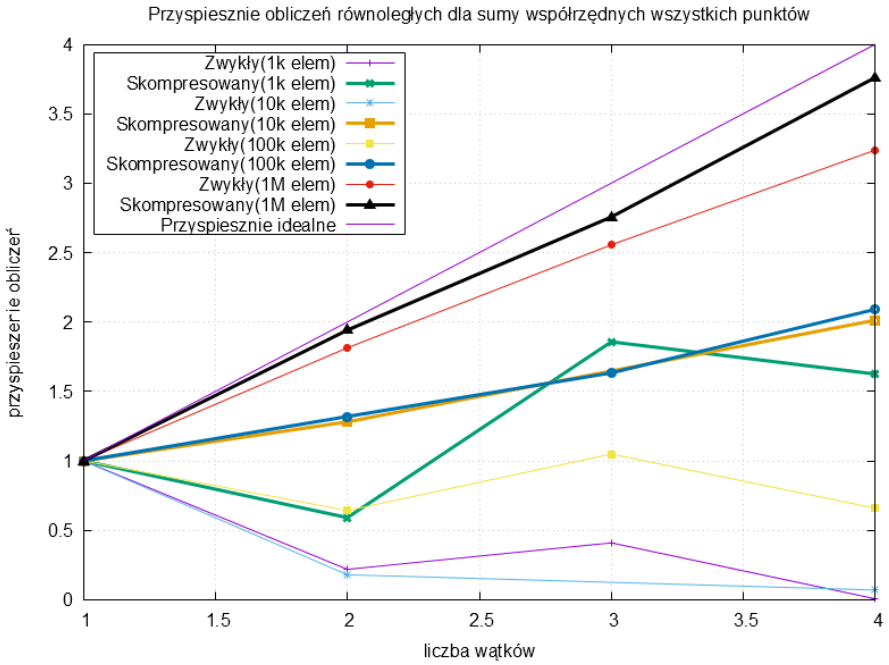
postaci *krzywych Z*, które z kolei są implementowane za pomocą tzw. kodu Mortona. Jest to rozwiązanie znane i szeroko używane, również do przydzielania identyfikatorów siatki m.in. w pakiecie obliczeniowym p4est [31] lub w innych zastosowaniach. Do porównania z omawianym w tej pracy algorytmem wybrana została wysokowydajna biblioteka *libMorton* wykorzystywana w praktycznych zastosowaniach [11]. Umożliwia ona skorzystanie zarówno ze standardowego algorytmu kodowania/de-



Rysunek 6.8: Czasy uzyskane w zadaniu testowym dla 100 000 do 1 000 000 elementów.

kodowania kodu Mortona, jak i również implementacji opartej m.in. na tablicach logicznych tzw. *lookup tables*.

O ile idea i implementacja kodu Mortona była przedmiotem ww. publikacji, należy zaznaczyć, iż formalnie kod Mortona jest odwzorowaniem liniowym $\mathbb{R}^3 \to \mathbb{N}$, ale wymaga on wcześniejszego zapisania współrzędnych rzeczywistych omawianego podzbioru w postaci liczb całkowitych. Najczęściej dokonuje się tego za pomocą



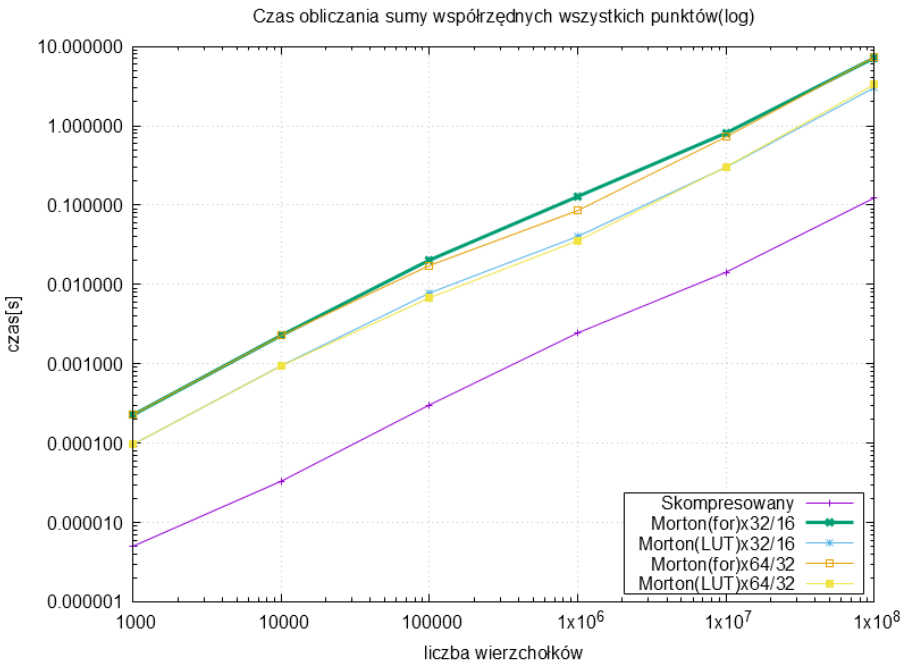
Rysunek 6.9: Przyspieszenie uzyskane w testach.

tzw. normalizacji do zakresu $[0, 1]$ lub $[0, MAX]$, gdzie MAX jest największą liczbą całkowitą możliwą do zapisania w wybranej reprezentacji. Normalizowanie nie jest *stricte* częścią kodowania Mortona, więc nie jest zaimplementowane w bibliotece *libMorton* i *nie* zostało uwzględnione w porównaniu. Należy też podkreślić, że kod Mortona z numerycznego punktu widzenia dokonuje przekształcenia dwóch liczb w liczbę o dwukrotnie większej reprezentacji bitowej. W związku z tym, aby zachować pojedynczą precyzję zmiennych punktu w $3D$, po wcześniejszym znormalizowaniu, należy użyć kodu Mortona $(x32, x32) \rightarrow (x64)$.

Analiza uzyskanych wyników, przedstawionych na rysunku 6.10 jednoznacznie wskazuje, że opracowany algorytm jest wydajniejszy niż kod Mortona w procesie kodowania / dekodowania współrzędnych o rząd wielkości. Widać również, że różnica ta jest stabilna i utrzymuje się, bez względu na wielkość zadania. Również dla kodowania Mortona $(x16, x16) \rightarrow (x32)$ prezentowane rozwiązanie jest wydajniejsze. Należy podkreślić, że dla kodowania Mortona konieczne jest dodatkowe każdorazowe dokonywanie normalizacji, nieuwzględnionej na wykresach.

6.3.6. Porównanie z dostępnymi pakietami zarządzania siatką

W trakcie analizy opracowanego algorytmu zostało przeprowadzone również porównanie szybkości działania operatora $\mathcal{G}_{eo}(\mathcal{W}(Element))$ zrealizowanego za pomocą algorytmów 7 i 8 w porównaniu z jednym z najbardziej znanych pakietów obliczeniowych wysokiej wydajności. Jako reprezentant zostało wybrane The Portable, Exten-



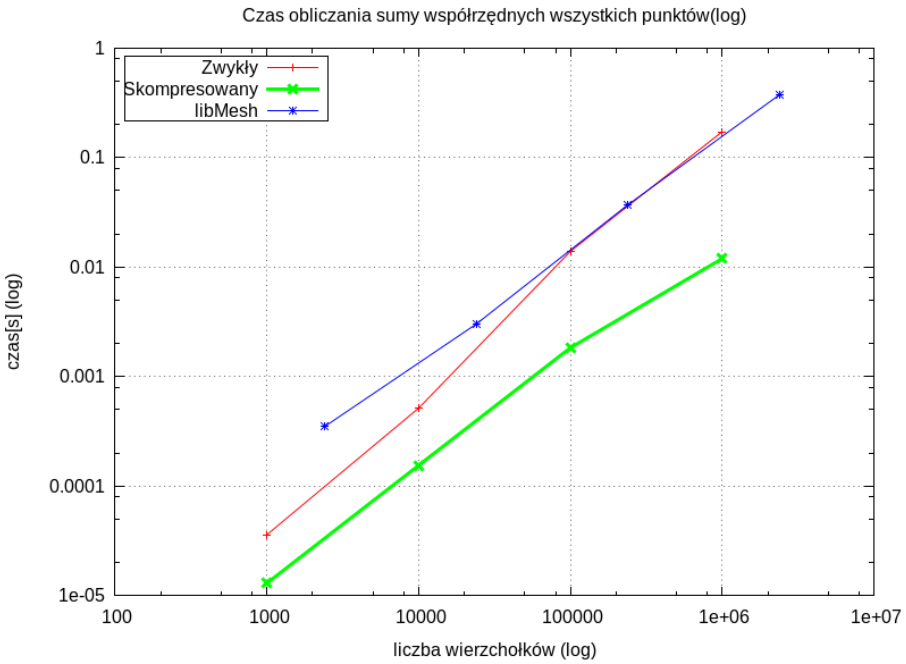
Rysunek 6.10: Porównanie efektywności przedstawianego algorytmu z kodem Mortona - skala logarytmiczna.

sible Toolkit for Scientific Computation (PETSC), posiadające co najmniej 767 udokumentowanych publikacjami przypadków użycia w obliczeniach naukowo-technicznych wg. autorów pakietu [12]. Ponieważ PETSC jest oprogramowaniem modularnym, żeby zachować obiektywność porównania, została użyta biblioteka bezpośrednio odpowiedzialna za zarządzanie siatkami, którą w PETSC jest libMesh. Wykorzystano najnowszą (2016) wersję PETSC i libMesh. Celem zadania testowego było porównanie czasu dostępu do współrzędnych wierzchołków elementów, przy założeniu standardowej siatki czworościennej o geometrii prostopadłościanu. Za podstawę poprawnego zdefiniowania zadania przyjęto załączone przykłady wraz z dostępną dokumentacją. Dla odniesienia sprawdzono również przykład, w którym wierzchołki i ich współrzędne są przechowywane w zwykłej tablicy. Wielokrotnie powtarzane próby są podsumowane na wykresie 6.11.

Z wykresu 6.11 wynika, że dla reprezentatywnego zadania testowego opracowany algorytm jest około o rząd wielkości szybszy od implementacji libMesh wykorzystywanej w PETSC.

6.3.7. Wady i ograniczenia.

Powyższa metoda ma następujące wady i ograniczenia, po części wynikające z niej samej, a po części charakterystyczne dla metod realizujących podobną funkcjonalność.



Rysunek 6.11: Porównanie efektywności przedstawianego algorytmu z PETSC/libMesh- skala logarytmiczna.

— Dla siatek korzystających z niestacjonarnych wierzchołków opisywana metoda wymaga uaktualniania identyfikatorów i zapisu skompresowanego, co ma negatywny wpływ na wydajność stosowania.

— Skompresowana reprezentacja punktów obarczona jest (bardzo niewielkim, ale jednak) błędem przybliżenia.

— Dla dużych siatek adaptacyjnych wymagane jest skorzystanie z reprezentacji opartej na liczbach 64 bitowych ze względu na geometryczne zawężanie obszaru dostępnych punktów do wyboru wraz ze zmniejszaniem rozmiaru liniowego elementów. Przykładowo dla siatki obliczeniowej obiektu o długości $1[m]$ liniowy rozmiar elementu skończonego musiałby być mniejszy niż $0.0006153[m] = 0.6[mm]$. Dla reprezentacji 64 bitowej analogiczny rozmiar musiałby być mniejszy niż $3 \cdot 10^{-7}[m]$, co jest 2 rzędy wielkości mniejszą wartością niż np.: rozszerzalność żelaza w temperaturze pokojowej oraz o rząd mniejszą wartością niż rozmiar niektórych ziaren w stali (materiał nie może być dłużej rozpatrywany jako ciągły ośrodek).

— Dla ekstremalnych przypadków metoda może skutkować przydzieleniem tego samego identyfikatora dwóm różnym punktom.

Rozdział 7

Wieloprocesowość z pamięcią rozproszoną

Ponieważ przetwarzanie w modelu z pamięcią rozproszoną wymaga uwzględniania dodatkowych funkcjonalności takich jak implementacja komunikacji, synchronizacja podobszarów czy przekazywanie informacji o rozwiązaniu pomiędzy obszarami granicznymi [51, 55, 68], badania przedstawione w pkt. 6.1, zostały powtórzone dla tego przypadku. Również i tym razem używany był program ModFEM [79, 80], ale wzbogacony o dodatkowe moduły pozwalające na realizację w środowisku rozproszonym. W szczególności były to moduły:

Komunikacji równoległej właściwa nazwa: `pcl_mpi_safe` od *parallel communication library mpi safe*. Moduł ten powstał na zasadzie rozwinięcia przez autora wersji modułu komunikacji za pomocą standardu MPI [20, 79, 80]. Jest on wzbogacony o obsługę wielu buforów przesyłania na raz, przyjmowanie wiadomości poza kolejnością (out-of-order), kontrolę typów przesyłanych, kontrolę dekompresji danych z bufora, system testowania i szereg innych pomniejszych usprawnień mających na celu zwiększenie niezawodności i rzetelności komunikacji równoległej.

Równoległy adapter siatki właściwa nazwa: `mmpl_adapter` od *mesh module parallel library adapter*. Jest to napisany od nowa moduł mający służyć za uniwersalny adapter dla (nierozproszonych) modułów siatek w projekcie ModFEM (aktualnie są cztery osobne moduły siatki nierozproszonej). Odpowiada za realizację dostarczonego schematu podziału na pod-obszary, synchronizację pod-obszarów, przesyłanie danych, przenoszenie i kopiowanie obiektów siatki między obszarami itd. itp.

Równoległa aproksymacja właściwa nazwa: `apl_std_lin` od *approximation parallel library standard linear*. Jest to istniejący już wcześniej moduł odpowiedzialny za realizację rozproszonej aproksymacji [20, 79, 80]. Został wzbogacony o wykorzystanie informacji o sąsiedztwie dostarczanej przez moduł `mmpl_adapter` w celu zminimalizowania komunikacji między obszarami.

Dekompozycja domeny właściwa nazwa: `ddl_parmetis` od *domain decomposition library parmetis*. Jest to moduł napisany w celu wykorzystania biblioteki

PARMETIS i METIS [69] jako narzędzia to generowania wysokiej jakości schematu podziału na pod-obszary oraz poprawiania go.

Równoległe operacje wejścia-wyjścia właściwe nazwy: `utl_io_intf`, `utl_io_results` od *utilities library input output interface, results*. Są to moduły odpowiedzialne za równoległe operacje wejścia-wyjścia w środowisku z pamięcią rozproszoną, w szczególności służące do poprawnego zbierania rezultatów obliczeń oraz danych analitycznych.

Moduł adaptacji właściwa nazwa: `utl_adapt` od *utilities library adaptation*. Stanowi on wydzielony fragment programu odpowiedzialny za współpracę modułów siatki i aproksymacji w czasie adaptacji oraz realizuje (za monolitycznym interfejsem) adaptację w pamięci wspólnej lub rozproszonej wedle potrzeby.

7.1. Niezależność procesów, synchronizacja i problem uzgadniania w czasie adaptacji

Jak wykazała analiza literatury i praktyka obliczeniowa, jednym z zagadnień istotnych dla wykorzystywania pamięci rozproszonej jest jednoznaczne identyfikowanie obiektów. Jednym z podejść wykorzystywanych do jednoznacznego przydzielania identyfikatorów w sposób stwarzający możliwość uniknięcia synchronizacji na poziomie globalnym lub lokalnym, jest wykorzystanie krzywych wypełniających przestrzeń. Przedstawiona w tej pracy realizacja kompresji współrzędnych siatki ma pewne cechy podobne do dotychczas używanych krzywych wypełniających przestrzeń. Otóż wykorzystując te cechy, został opracowany wydajny schemat identyfikowania obiektów w odniesieniu do ich faktycznego geometrycznego położenia.

Początkowo odnośnie globalnej identyfikacji obiektów przyjęto rozwiązanie, bazujące na koncepcji operatora $\mathcal{O}(\cdot)$ określającego przypisanie obiektu siatki do pewnego procesu p spośród wszystkich procesorów $\boldsymbol{P}$. Następnie wprowadzono globalny identyfikator $Guid_O(\cdot)$ jako operator, który każdemu obiektowi siatki t przypisuje parę liczb oznaczających numer procesu p będącego właścicielem t oraz lokalny identyfikator $Loc_{ID}(t)$ w procesie posiadającym obiekt:

$$\begin{aligned} Guid_O &: \mathbb{N} \to \mathbb{N}^2 \\ Guid_O(t) &= [p, loc_t^p] : \forall k \in \boldsymbol{P} \\ &\qquad \mathcal{O}_k(t) = p \wedge Loc_{ID}^p(t) = loc_t^p \end{aligned} \tag{7.1}$$

Niestety okazało się, że ten rodzaj identyfikatora wymaga by operacje na siatkach były linearyzowalne, oraz by porządek linearyzacji był znany. W przeciwnym wypadku, w czasie dynamicznej adaptacji na brzegach pod-obszarów obliczeniowych zsynchronizowanie identyfikatorów jest bardzo kosztowne. Praktycznie oznacza to,

że sposób organizacji siatki obliczeniowej na każdym węźle jest zależny od utrzymania spójnej globalnej organizacji siatki, czego efektem jest zależny sposób numeracji, orientacji, kolejność wykonywania operacji etc. Na rysunku 7.1 widać mapę wywołań z profilowania wskazującą, iż przeprowadzenie takiej synchronizacji, która ostatecznie musi mieć globalny zakres, powoduje gwałtowny wzrost czasu przeznaczonego na komunikację wraz ze wzrostem ilości procesów.

Rysunek 7.1: Zrzut ekranu z profilera callgrind - mapa wywoływanych funkcji w trakcie symulacji w środowisku rozproszonym z adaptacją. Powierzchnia mapy odpowiada 100% czasu programu. Wskazywany strzałkami obszar to wewnętrzne funkcje implementacji w standardzie MPI odpowiedzialne za przekazywanie komunikatów między procesami w modelu pamięci rozproszonej.

W związku z tym niezależna dynamiczna adaptacja na każdym węźle obliczeniowym wymaga innego sposobu określania globalnego identyfikatora. Opracowane rozwiązanie jest zaprezentowane w następnym punkcie.

7.2. Jednoznaczna identyfikacja w oparciu o dyskretną przestrzeń punktów

Bazą do nowej metody jednoznacznej identyfikacja dedykowanej do przetwarzania w modelu pamięci rozproszonej (choć nie tylko) jest koncepcja wykorzystania dyskretnej chmury regularnie rozłożonych punktów, która została również użyta do zdefiniowania algorytmu kompresji współrzędnych siatki w rozdziale 6.3 (wzór 6.10 i dalej). W tym celu możemy wyznaczyć osobny operator dla każdej kategorii obiektów w siatce, a zbiór tych operatorów będzie obejmował wszystkie obiekty w siatce:

$$Guid_{glob} = \{Guid_e, Guid_s, Guid_k, Guid_w\} \tag{7.2}$$

Zanim zdefiniujemy operatory $Guid_e, Guid_s, Guid_k, Guid_w$ zwróćmy uwagę, że podstawowa koncepcja definicji jest bardzo prosta i polega na wykorzystaniu koncepcji punktu środkowego. Otóż dla każdej kategorii obiektu topologicznego (niekrzywoliniowego) można wyznaczyć punkt środkowy, bazując na skrajnych wierzchołkach.

wierzchołka - jest nim on sam,
krawędzi - jest punkt dokładnie pomiędzy końcami krawędzi (odcinka),
ściany - jest nim punkt środkowy ściany,
elementu - jest nim punkt środkowy objętości.

Przy czym można zauważyć, że przy założeniu, że każdy obiekt składa się z pewnych wierzchołków, a wierzchołek składa się z samego siebie, to wszystkie te przypadki sprowadzają się do policzenia średniego położenia wierzchołka, ze wszystkich wierzchołków określających dany topologiczny obiekt. Określonemu w ten sposób punktowi środkowemu może zostać przypisany numer, wykorzystując ten sam mechanizm, który posłużył do skompresowanego zapisu siatki. Zakładając, że zgodnie z opisem w pkt 6.3 siatka posiada zapis skompresowany uzyskujemy możliwość przedstawienia operacji identyfikacji w następujący sposób:

Identyfikowanie wierzchołków Dla danego wierzchołka $w \in \boldsymbol{\mathcal{W}}$ siatki M jako operator definiujący jego *jednoznaczny globalny identyfikator* $Guid_w$ stosujemy operator $Guid(\cdot)$ jak opisany w pkt. 6.21 i wcześniej.

Identyfikowanie krawędzi Dla krawędzi $k \in \boldsymbol{\mathcal{K}}$ siatki M operator $Guid_k$ definiujemy jako

$$\begin{aligned} Guid_k &: \boldsymbol{\mathcal{K}} \to \mathbb{N}^+ \\ Guid_k(k) &= Guid_w\left(\frac{\sum \mathcal{W}(k)}{|\mathcal{W}(k)|}\right) \end{aligned} \tag{7.3}$$

Identyfikowanie ścian Dla ściany $s \in \boldsymbol{\mathcal{S}}$ siatki M operator $Guid_s$ definiujemy jako

$$\begin{aligned} Guid_s &: \boldsymbol{\mathcal{S}} \to \mathbb{N}^+ \\ Guid_s(s) &= Guid_w\left(\frac{\sum \mathcal{W}(s)}{|\mathcal{W}(s)|}\right) \end{aligned} \tag{7.4}$$

Identyfikowanie elementów Dla elementu $e \in \boldsymbol{\mathcal{E}}$ siatki M operator $Guid_e$ definiujemy jako

$$\begin{aligned} Guid_e &: \boldsymbol{\mathcal{E}} \to \mathbb{N}^+ \\ Guid_e(e) &= Guid_w\left(\frac{\sum \mathcal{W}(e)}{|\mathcal{W}(e)|}\right) \end{aligned} \tag{7.5}$$

W ten sposób podaliśmy niewymagający komunikacji ani synchronizacji operator $Guid_{glob}$ składający się ze zbioru operatorów $\{Guid_e, Guid_s, Guid_k, Guid_w\}$.

Rozważmy jeszcze jeden interesujący aspekt. Otóż w powyższych wzorach wyznaczenie identyfikatora globalnego wymaga pobierania współrzędnych wierzchołków obiektów siatki. Wprowadźmy zatem operator wyznaczania punktu środkowego Mid zbioru punktów w przestrzeni punktów dyskretnych $\boldsymbol{P_M}$ (wzór 6.17), który dla danej siatki M zbiorowi n punktów w $\boldsymbol{P_M}$, przypisuje jeden punkt w $\boldsymbol{P_M}$:

$$
\begin{aligned}
Mid :&{\boldsymbol{P_M}}^n \to \boldsymbol{P_M} \\
Mid(\{w_1, w_2, ..., w_n\}) =& \frac{\sum_{w=1}^{n} \left(i_x^w \cdot \frac{m}{m_x} + i_y^w \cdot \frac{m}{m_y} + i_z^w \cdot \frac{m}{m_z} \right)}{n} \\
=& m \cdot \frac{\sum_{w=1}^{n} \left(\frac{i_x^w}{m_x} + \frac{i_y^w}{m_y} + \frac{i_z^w}{m_z} \right)}{n} \\
=& m \cdot \frac{\sum_{w=1}^{n} \left(\frac{i_x^w}{m_x} \right) + \sum_{w=1}^{n} \left(\frac{i_y^w}{m_y} \right) + \sum_{w=1}^{n} \left(\frac{i_z^w}{m_z} \right)}{n} \\
=& m \cdot \left(\frac{\sum_{w=1}^{n} \left(\frac{i_x^w}{m_x} \right)}{n} + \frac{\sum_{w=1}^{n} \left(\frac{i_y^w}{m_y} \right)}{n} + \frac{\sum_{w=1}^{n} \left(\frac{i_z^w}{m_z} \right)}{n} \right) \\
=& m \cdot \left(\frac{1}{m_x} \frac{\sum_{w=1}^{n} i_x^w}{n} + \frac{1}{m_y} \frac{\sum_{w=1}^{n} i_y^w}{n} + \frac{1}{m_z} \frac{\sum_{w=1}^{n} i_z^w}{n} \right) \\
=& i_x^{Mid} \cdot \frac{m}{m_x} + i_y^{Mid} \cdot \frac{m}{m_y} + i_z^{Mid} \cdot \frac{m}{m_z}
\end{aligned}
\tag{7.6}
$$

Jak widać w ostatniej linijce wzoru 7.6 nowe położenie punktu środkowego w zbiorze punktów dyskretnych $\boldsymbol{P_M}$ jest niezależne od wartości m, m_x, m_y, m_z, które są stałe dla całej siatki. Oznacza to, że aby obliczyć taki punkt środkowy nie ma potrzeby dekodowania jego faktycznego fizycznego położenia, bo można wyznaczyć go na podstawie średniej wartości parametrów i_x^w, i_y^w, i_z^w punktów przekazanych jako argument operatora. Wykorzystując tę właściwość, dla siatki M globalny operator identyfikacji $Guid_{glob}$ można zdefiniować w następujący sposób (odpowiednio do wzorów 7.2, 7.3, 7.4, 7.5):

$$
\begin{aligned}
Guid_w(w) =& Guid(w) \\
Guid_k(k) =& Guid_w(Mid(\mathcal{W}(k))) \\
Guid_s(s) =& Guid_w(Mid(\mathcal{W}(s))) \\
Guid_e(e) =& Guid_w(Mid(\mathcal{W}(e)))
\end{aligned}
\tag{7.7}
$$

Tak określone identyfikatory mają następujące zalety:

— nie wymagają pobierania informacji o faktycznym geometrycznym położeniu,
— nie wymagają synchronizacji,

— nie przydziela się ich, nie zwalnia się ich, (brak zarządzania identyfikatorami)
— są jednoznaczne,
— mogą być obliczane niezależnie od siebie,
— ich obliczanie jest „żenująco równoległe" (ang. embarrassingly parallel)
— nie są związane z podziałem na pod-obszary obliczeniowe,
— w większości przypadków identyfikatory nie duplikują się pomiędzy różnymi kategoriami obiektów.

Oraz następujące wady:

— przy zbyt gęstym rozlokowaniu obiektów w przestrzeni, identyfikatory mogą nie być jednoznaczne (jak w pkt. 6.3.7),
— przy podziale hierarchicznym siatki, jeżeli zachodzi sytuacja, iż obiekt potomny powstaje dokładnie w środku obiektu dzielonego, to mogą oba te obiekty otrzymać ten sam identyfikator; wymaga to rozróżnienia kolekcji identyfikatorów obiektów na różnych poziomach hierarchii, np.: w postaci różnych struktur danych.

7.3. Analiza skalowalności

7.3.1. Środowisko i zadanie testowe

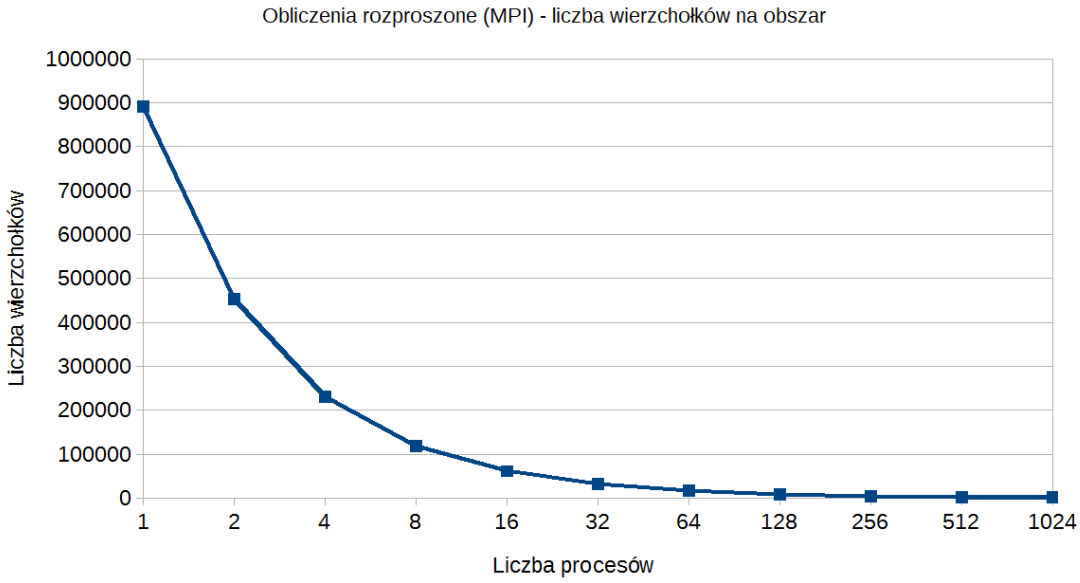
Przeprowadzano pomiary w środowisku klastra Prometheus (Cyfronet AGH), które dostarczyły informacji na temat czasu przeznaczonego na faktyczne realizowanie obliczeń wykorzystujących opracowane rozwiązania. Klaster Prometheus (węzeł dostępowy: prometheus.cyfronet.pl) wyprodukowany przez firmę Hewlett-Packard (Model Apollo 8000 Gen9) posiada teoretyczną moc obliczeniową 2399 TFlops. Składa się z 2160 serwerów HP XL730f Gen9, każdy z dwoma procesorami Intel Xeon E5-2680v3 2,50 GHz (w smie 24 rdzenie) i 128GB RAM na węzeł (sumarycznie 279 TB RAM). Posiada również 72 węzły HP XL750f Gen9, które dodatkowo posiadają po dwa akceleratory Nvidia Tesla K40 XL. Węzły obliczeniowe są połączone siecią Infiniband FDR 56Gb/s. Całość działa pod kontrolą systemu operacyjnego CentOS 7 i systemu kolejkowego SLURM.

Jak to zostało opisane wcześniej, do badań użyto pakietu ModFEM, budowanego bezpośrednio na klastrze. Wykorzystano kompilatory firmy Intel: icc, icpc, mpicc, mpiicpc w wersji 16.0.2, biblioteki metis 5.1 i parmetis 4.0 (obie wersje 32bitowe). Przygotowano szereg zadań dla różnej ilości procesów od 1 procesu do 1024 procesów, uruchamianych za pomocą skryptów do systemu kolejkowego, nakładając ograniczenie 4 zadań na fizyczny węzeł obliczeniowy wynikające z wymaganej pamięci RAM na pojedyncze zadanie. Badano zarówno połączenie MPI z OpenMP (do 6 wątków na proces), ale przedstawione tutaj rezultaty dotyczą wykorzystania samego MPI.

Do badania skalowalności wybrano nieregularną siatkę obliczeniową wykorzystywaną w praktycznych zagadnieniach do modelowania procesu chłodzenia kół zębatych. Siatka uwzględnia element chłodzony, umocowanie oraz elementy dociskające, jak również płyn chłodzący. Ze względu na kanały chłodzące jest to siatka o skomplikowanej geometrii wewnętrznej posiadającą początkowo 5 167 618 elementów 3D czworościennych, 10 384 044 ścian, 6 108 192 krawędzi oraz 891 767 węzłów. Rozmiar, rozmieszczenie i orientacja elementów były silnie zróżnicowane. Przykładowa wizualizacja (z wykorzystaniem programu ParaView) podziału na pod-obszary jest przedstawiona na rysunkach 7.4 i 7.5, 7.6.

7.3.2. Wyniki i wnioski

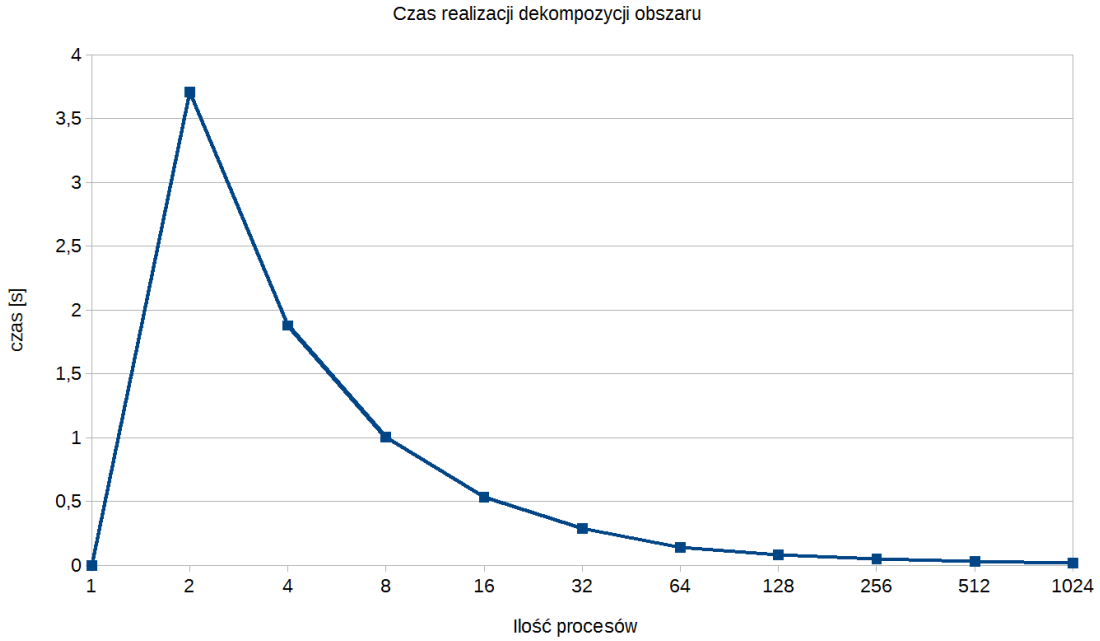
Na rysunku 7.2 widać, jak w miarę zwiększania ilości węzłów (stały rozmiar zadania), spada rozmiar zadania na proces.



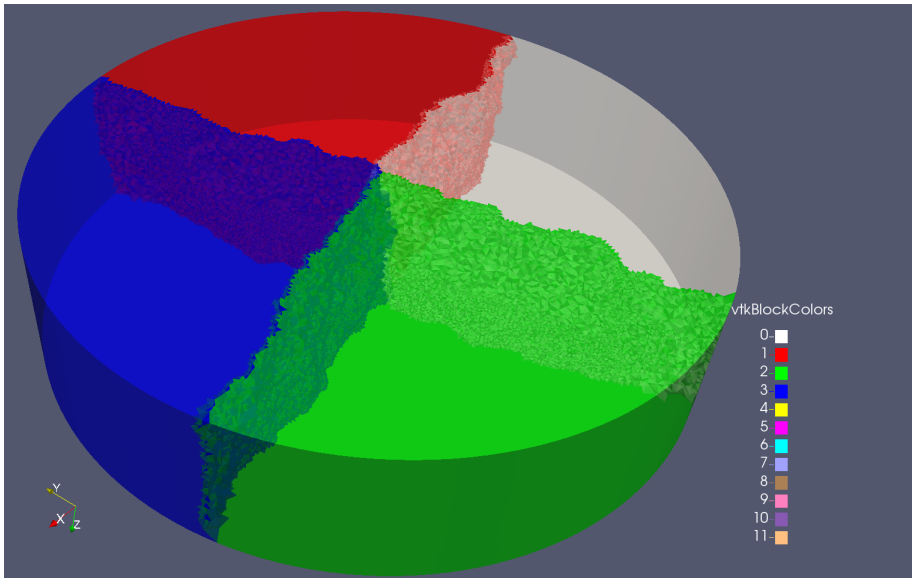
Rysunek 7.2: Zmniejszanie ilości wierzchołków na pod-obszar siatki obliczeniowej w miarę podziału równoległego w modelu pamięci rozproszonej. Dla jednego pod-obszaru było 891767 wierzchołków. Minimalna liczba wyniosła miedzy 1190 a 1516 na pod-obszar.

Jednocześnie na rysunku 7.3 można zaobserwować, ile dla poszczególnych przypadków trwało tworzenie pod-obszarów. Wynika stąd, że w tym kontekście najbardziej wymagające jest zadanie rozwiązywane przez dwa procesy, kiedy rozmiar zadania na proces nadal jest bardzo duży i przez to granica pomiędzy nimi również jest bardzo szeroka. Na tym wykresie, widać, że dalej, wraz z malejącym zadaniem, zaczyna odgrywać znaczenie mniejszy rozmiar zadania, mimo coraz większego są-

siedztwa między procesami. Wizualizacja podziału na pod-obszary jest pokazana na rys. 7.4, 7.5, 7.6.

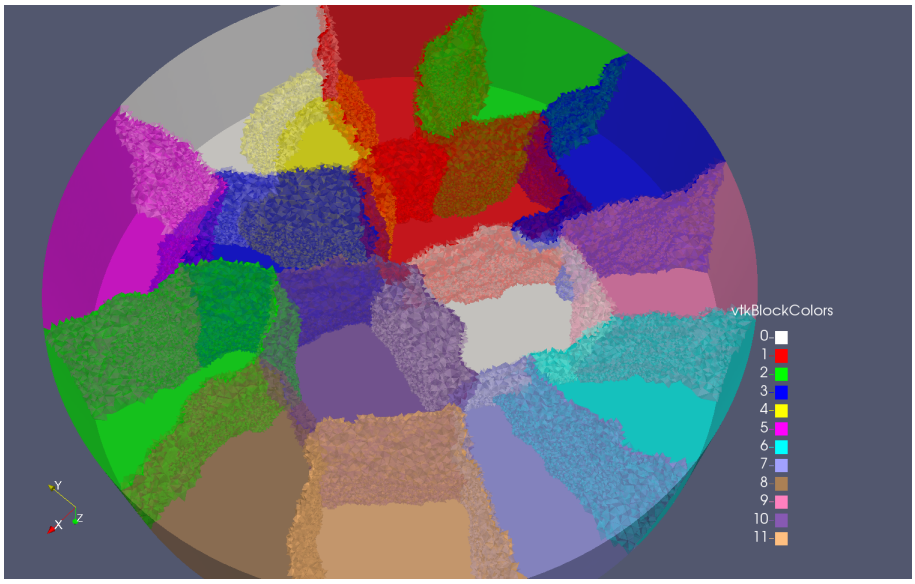


Rysunek 7.3: Czas dekompozycji obszaru na pod-obszary w funkcji liczby procesów. Zakłada się 1 pod obszar na proces. Dla jednego procesu dekompozycja jest pomijana, stąd początkowy czas wynosi 0[s].

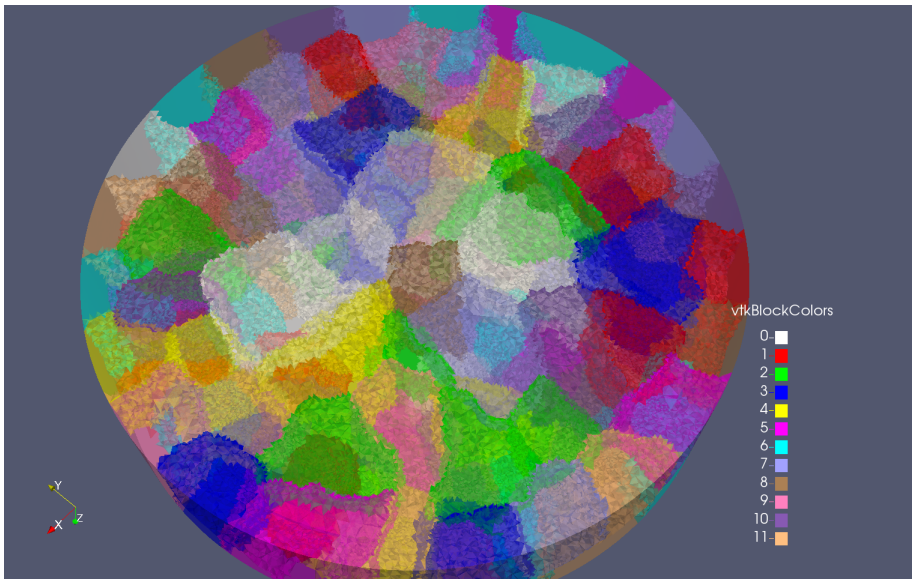


Rysunek 7.4: Podział rozproszonej siatki obliczeniowej MES na 4 pod-obszary.

Interesujący wykres znajduje się na rys. 7.7, który przedstawia czas obliczeń, który jest zasadniczą miarą jakości opracowanego rozwiązania. Skala na osi X nie



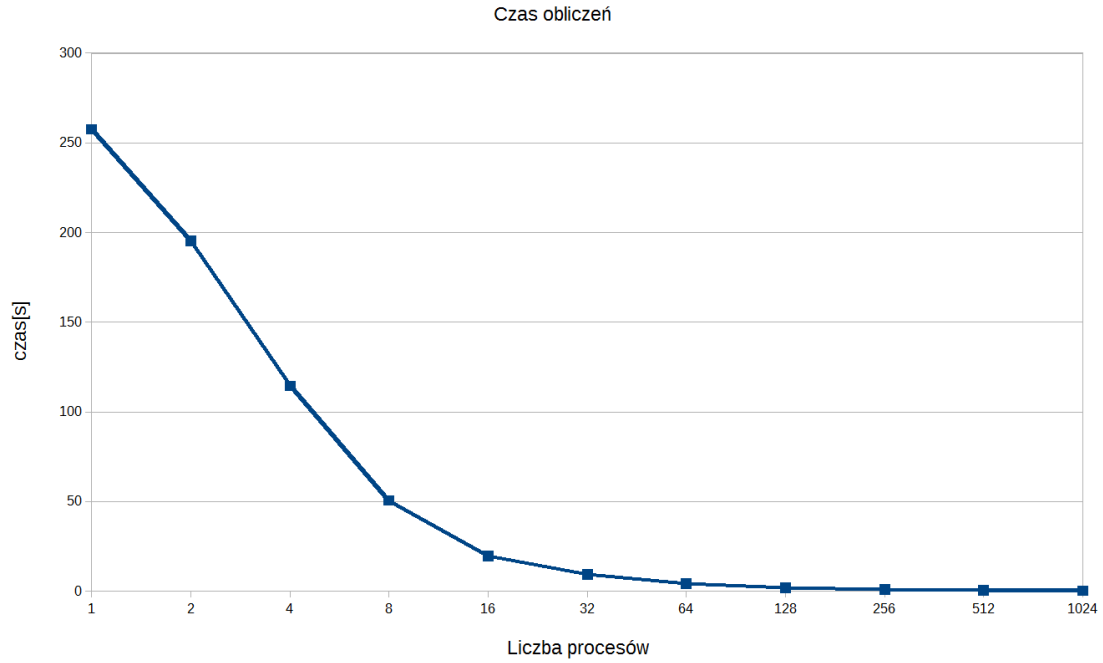
Rysunek 7.5: Podział rozproszonej siatki obliczeniowej MES na 16 pod-obszarów.



Rysunek 7.6: Podział rozproszonej siatki obliczeniowej MES na 128 pod-obszarów.

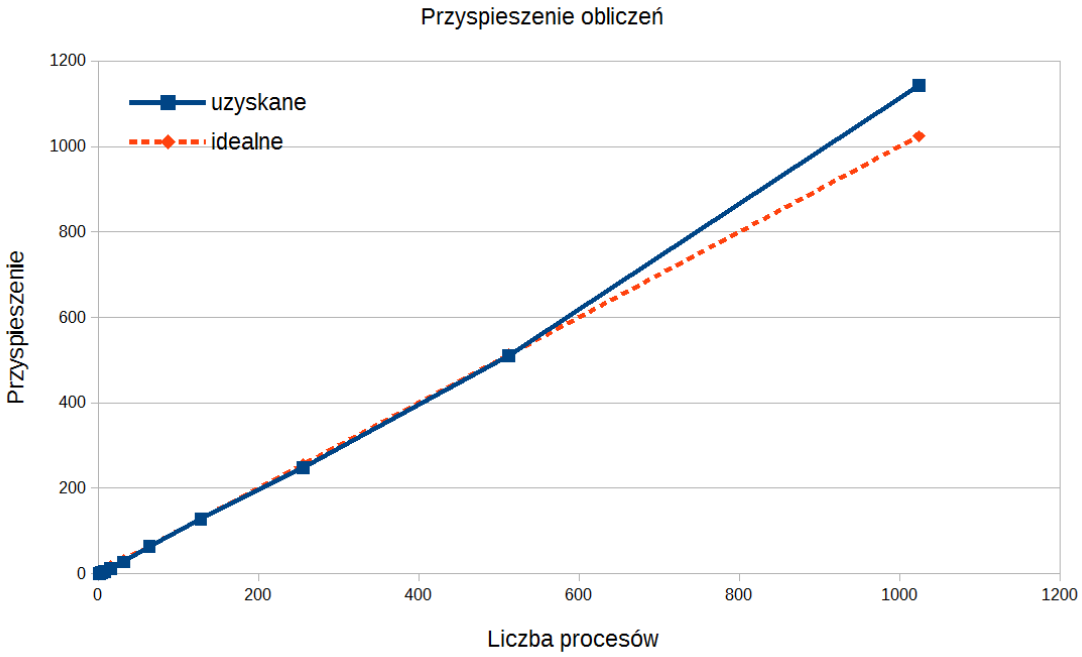
jest liniowa lecz wykładnicza, by lepiej zobrazować wyniki. Warto zwrócić uwagę, że czas obliczeń stale maleje w funkcji ilości procesów.

Z punktu widzenia badania skalowalności silnej, najważniejszy jest wykres przedstawiony na rysunku 7.8. Widać na nim porównanie osiągniętego przyspieszenia do przyspieszenia idealnego. Analiza wykresu uwydatnia, że uzyskane przyspieszenie jest bardzo bliskie idealnemu. Dla 1024 procesów, gdy zagadnienie było już bardzo małe na 1 proces, (jak pokazano na wykresie 7.2) efekty lokalności pozwoliły uzyskać przyspieszenie ponad-idealne. Uzyskanie takiego efektu jest możliwe tylko wtedy, gdy komunikacja międzyprocesowa odbywa się głównie w lokalnym sąsiedz-



Rysunek 7.7: Czas obliczeń w funkcji liczby procesów. Zakłada się 1 pod-obszar na proces. Poprzez określenie czas obliczeń rozumie się rozwiązanie 1 pełnego kroku w ramach całkowania po czasie, uwzględniając uzgadnianie rozwiązania pomiędzy procesami.

twie procesu z minimalną komunikacją typu *wszyscy-do-jednego* i bez komunikacji typu *wszyscy-do-wszystkich*.



Rysunek 7.8: Przyspieszenie obliczeń w funkcji liczby procesów. Zakłada się 1 pod-obszar na proces. Linia ciągła przedstawia wartość średnią ze wszystkich procesów, linia przerywana przyspieszenie idealne.

Rozdział 8

Przykłady zastosowania

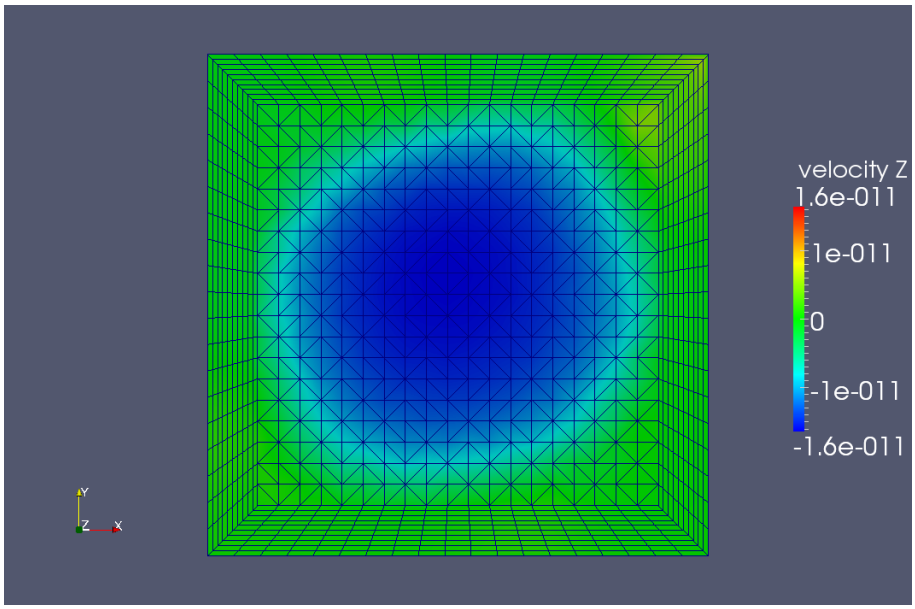
8.1. ModFEM

Omawiane w tej pracy rozwiązania i analizy są związane z projektem ModFEM [94], w którego rozwój autor jest zaangażowany [79, 80] i który stanowi dogodną bazę do pracy nad zagadnieniami siatek, ze względu na wewnętrzną strukturę jawnie separującą moduły siatek od innych modułów realizujących modelowanie metodą elementów skończonych.

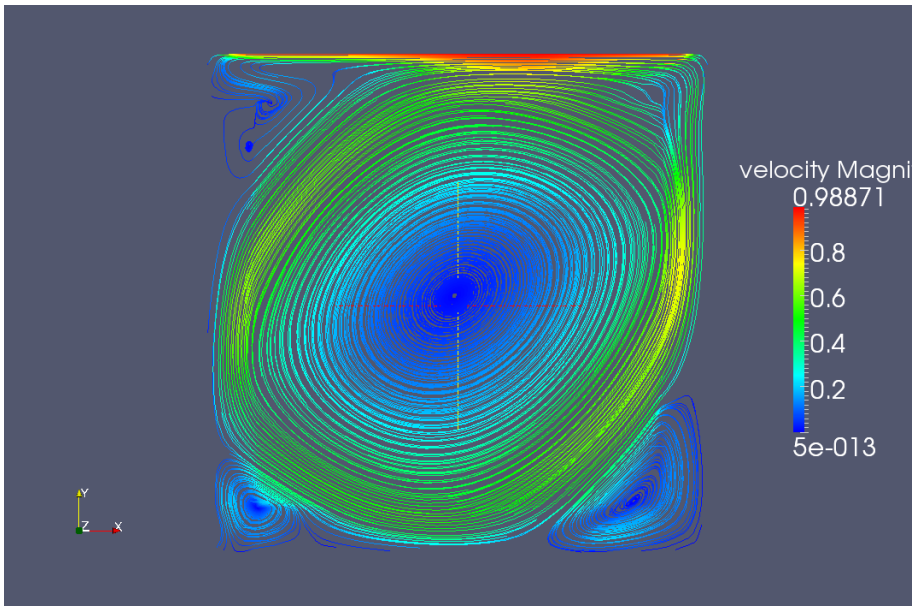
W ramach rozwoju pakietu ModFEM powstał szereg aplikacji służących do modelowania różnych zjawisk, w których były i są praktycznie używane omawiane w tej pracy rozwiązania. Przede wszystkim jednak korzystanie ze współpracy w ramach projektu ModFEM pozwoliło na walidację uzyskiwanych rozwiązań w standardowych testach takich jak przepływy płynu nieściśliwego typu Lid-Drive-Cavity, Backward Facing Step, Von Karman Vertx Street, Heat Drive Cavity itp.

8.1.1. Przepływu typu Lid driven cavity

W przypadku przepływy płynu nieściśliwego typu Lid-Driven-Cavity duże znaczenie ma dokładne modelowanie warstwy przyściennej płynu, w której występuje bardzo duży gradient prędkości, związany z warunkiem brzegowym *no-slip*, który sprawia, że na samej granicy płyn-ciało stałe nie płynie on w ogóle (rys.8.2). Zastosowanie siatki hybrydowej pozwala na wygenerowanie gęstszej siatki przy brzegu, a elementach anizotropowych w kształcie (rys. 8.1). Ponieważ elementy czworościenne niezbyt nadają się do tego typu zadań, więc zastosowano warstwy elementów pryzmatycznych. Pozwoliło to na otrzymanie wyników zgodnych z doświadczeniami laboratoryjnymi dotyczącymi przepływu tego typu, wykorzystując stosunkowo niewielką siatkę obliczeniową.



Rysunek 8.1: Przepływ Lid Driven Cavity - siatka hybrydowa. Widoczne elementy warstwy przybrzeżnej.

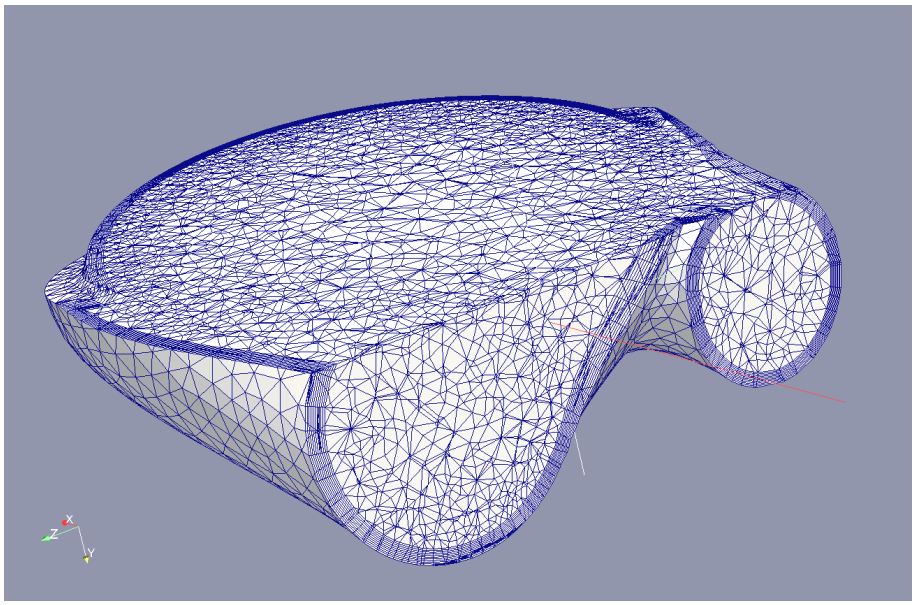


Rysunek 8.2: Przepływ Lid Driven Cavity - linie prądu, rozwiązanie na siatce hybrydowej.

8.2. Modelowanie polskiego sztucznego serca

Na podstawie projektu ModFEM powstał również program do modelowania przepływu krwi w komorze polskiego sztucznego serca oraz kanałach doprowadzających krew, z uwzględnieniem zastawek wlotowych i wylotowych [93]. Prace te były wykonywane w ramach projektu „Projekt Polskie Sztuczne Serce (3/0-PW/P02-PBZ-MNiSW/2007)". Program służył jako narzędzie pomocnicze służące do wspierania procesu projekto-

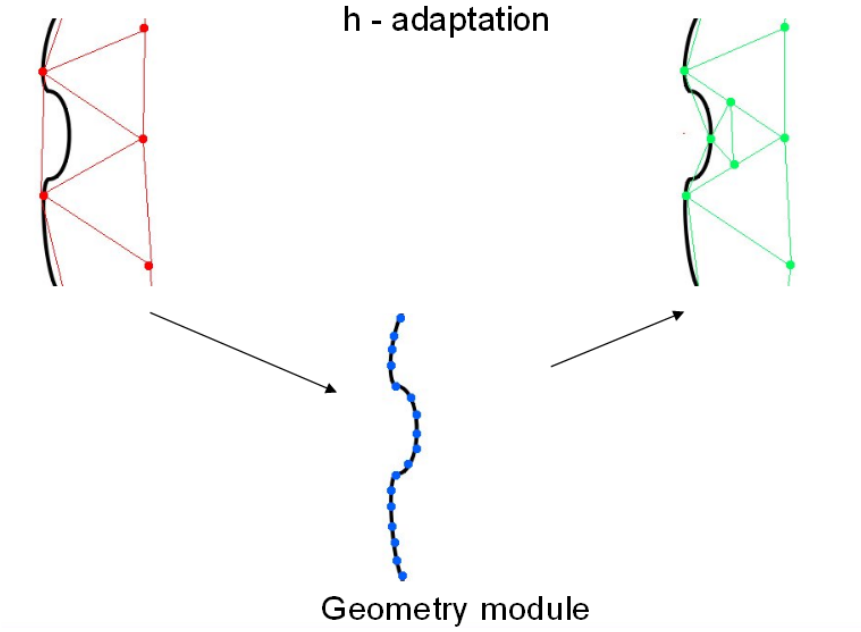
wania kształtu komory informacjami o krytycznych prędkościach przepływu oraz gradientach ciśnienia. Ze względu na skomplikowaną geometrię przepływu oraz istnienie warstwy przyściennej istotną rolę odegrała tu hybrydowa siatka adaptacyjna pokazana na rys. 8.3 i 8.6. Dodatkowo zastosowano rozwiązanie polegające na dokładniejszym modelowaniu brzegu obszaru w czasie h-adaptacji [18]. Schemat tego rozwiązania jest pokazany na rysunku 8.4. Wymagana dokładność obliczeń wymuszała przeprowadzanie modelowania na klastrach m.in. klastrze „Perszeron" Politechniki Krakowskiej. Modelowane również były geometrie o z uwzględnieniem zastawek co można zobaczyć na rys. 8.5. Ostatecznie uzyskane rozwiązania były wykorzystywane do obliczania ciśnień w projektowanej komorze sztucznego serca (rys. 8.7).



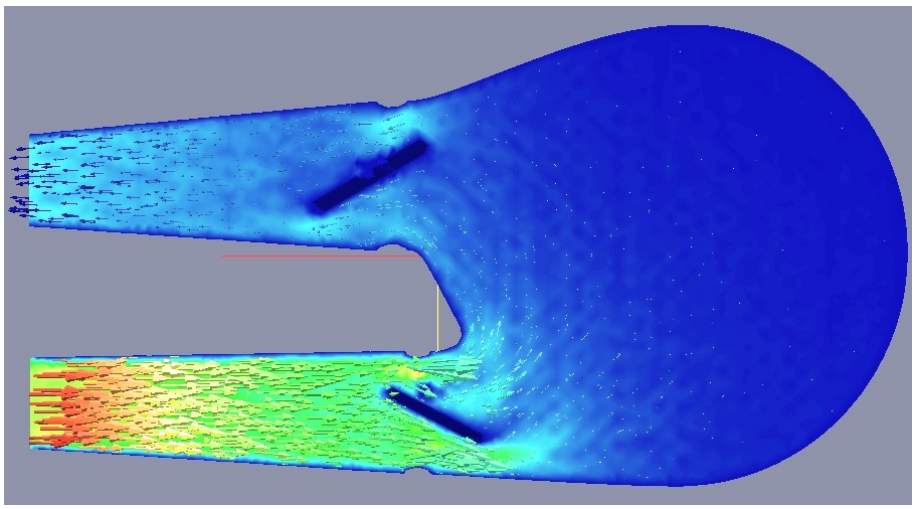
Rysunek 8.3: Polskie sztuczne serce - warstwa przyścienna w komorze sztucznego serca. Warto zwrócić uwagę na dostosowanie warstwy przyściennej do skomplikowanej geometrii.

8.3. Modelowania jeziorka spawalniczego

Z zastosowań bardziej przemysłowych, omawiane rozwiązania były i są użytkowane w celu modelowania zjawiska jeziorka spawalniczego w czasie trwania niestacjonarnego procesu spawania liniowego [111, 110]. Badania i prace te były początkowo realizowane w ramach projektu „Opracowanie i wdrożenie komputerowego systemu wspomagania procesów spawania w konstrukcjach lotniczych" (ZPB/33/63903/IT2/10). Spawanie liniowe jest zjawiskiem niestacjonarnym również w tym sensie, że powstaje przesuwające się jeziorko spawalnicze, jak pokazano na rys. 8.8, 8.10. Efektywne modelowania tego zjawiska wymaga zastosowania dynamicznej adaptacji i de-adaptacji

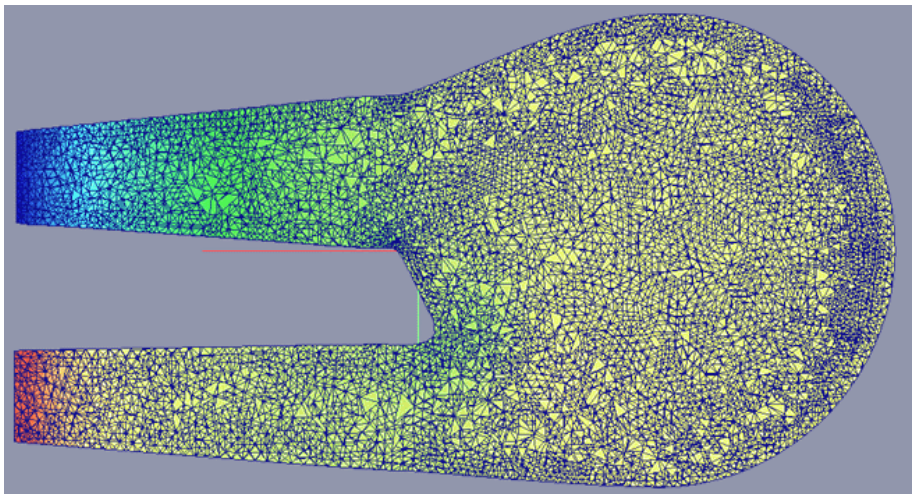


Rysunek 8.4: H-adaptacja z wykorzystaniem dodatkowego modułu przechowującego dokładną (gęstą) siatkę na brzegu obszaru wykorzystywaną do definiowania poprawnych położeń wierzchołków pojawiających się w czasie podziału elementów przy brzegu.

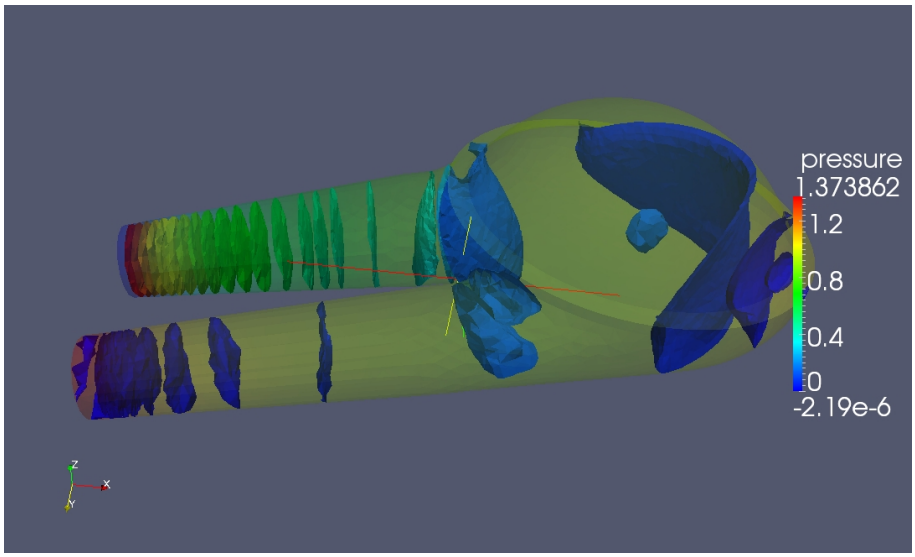


Rysunek 8.5: Polskie sztuczne serce - przepływ krwi przez komorę z zastawkami. Pokazane wektory prędkości płynu wpływającego (dolny kanał). Widać wpływ zastawek na rozkład prędkości w komorze.

siatki obliczeniwej, co zostało zrobione. Przykład zmiennego regionu zagęszczania siatki widać na rys. 8.9. Na rysunku widać kolejne stadia rozwoju jeziorka spawalniczego: a) nagrzewanie powierzchni i początek powstawania jeziorka (prędkość max. płynu w jeziorku 1.48 m/s); b) pogłębienie jeziorka (prędkość max. płynu w jeziorku



Rysunek 8.6: Polskie sztuczne serce - modelowanie przepływu krwi przez komorę sztucznego serca: rozkład ciśnień obliczony przy wykorzystaniu siatki adaptacyjnej.

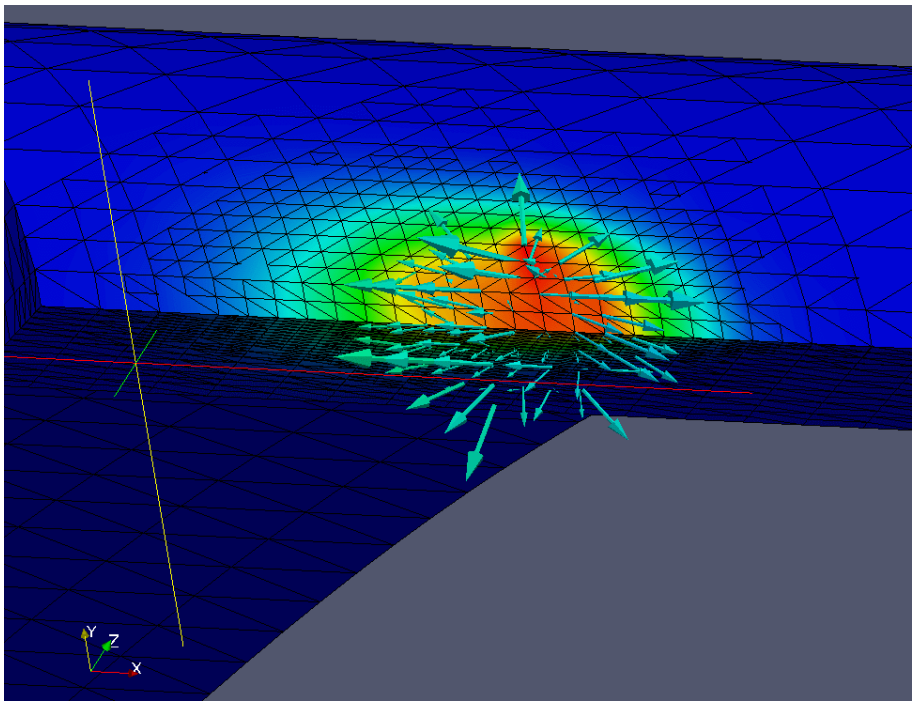


Rysunek 8.7: Polskie sztuczne serce - izopowierzchnie ciśnienia wewnątrz komory sztucznego serca. W tym przypadku zastawki otwarte i widać wyraźnie jak ciśnienie wpływającego płynu narasta na przeciwległej ścianie komory względem wlotu.

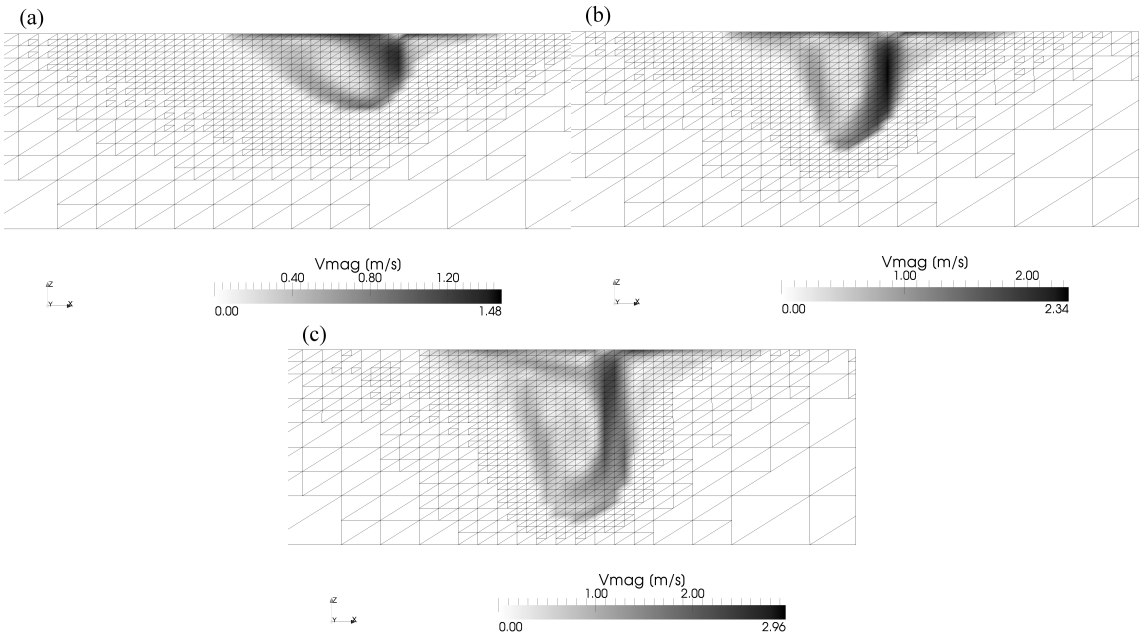
2.43 m/s); c) jeziorko w pełni ukształtowane (prędkość max. płynu w jeziorku 2.96 m/s). Dodatkowo na brzegach jeziorka spawalniczego następuje przemiana fazowa (topienie metalu), która również wymaga dokładnej siatki obliczeniowej.

8.4. Usługa ModFEM_met w ramach projektu PlGrid+.

Jednocześnie na bazie ModFEM jako kontynuacja poprzednich badań powstała usługa ModFEM_met w ramach projektu „Dziedzinowo zorientowane usługi i zasoby infrastruktury PL-Grid dla wspomagania Polskiej Nauki w Europejskiej Prze-

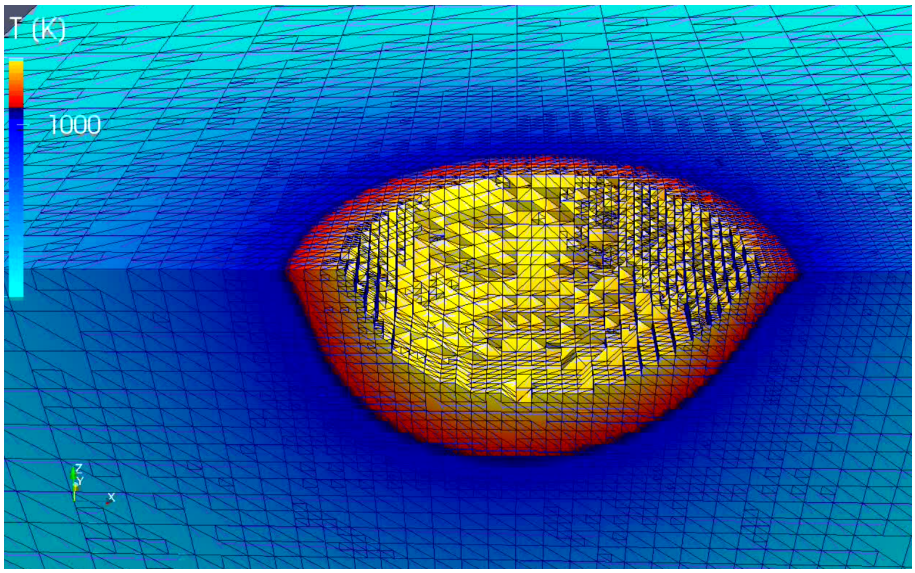


Rysunek 8.8: Jeziorko spawalnicze - problem niestacjonarny. Pokazany przekrój przez spawany element, kolory ukazują rozkład temperatury, a wektory prędkości ciekłego metalu w jeziorku spawalniczym.



Rysunek 8.9: Adaptacyjna siatka w czasie modelowania spawania.

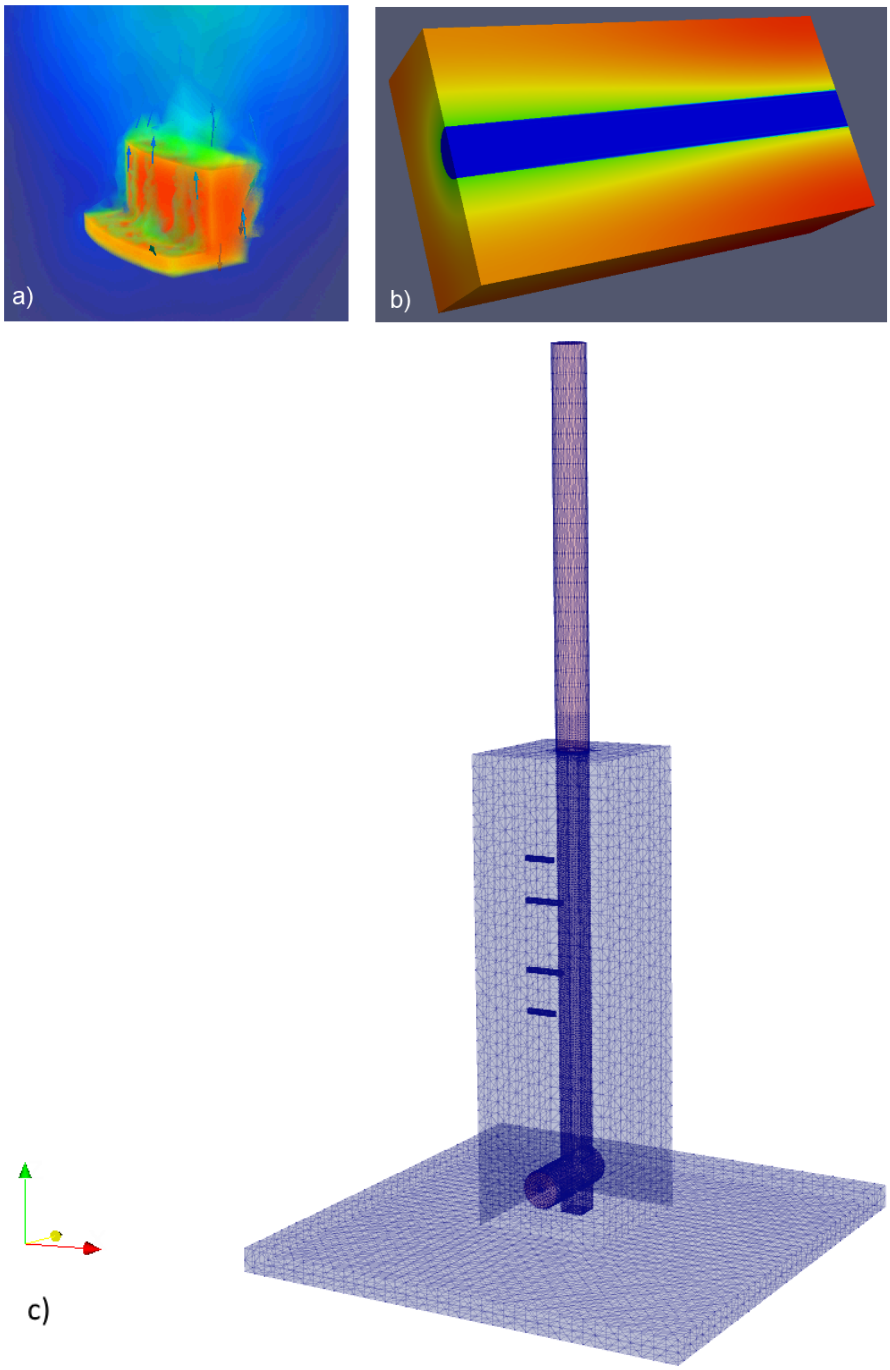
strzeni Badawczej – PLGrid Plus" (POIG.02.03.00-00-096/10) prowadzonego przez Cyfronet, AGH [21]. Usługa ModFEM_met została wzbogacona względem ModFEM o zewnętrzny interfejs użytkownika, który został połączony z aplikacją do zdalnego uruchamiania zadań na klastrach obliczeniowych *QCG-Icon*.



Rysunek 8.10: Jeziorko spawalnicze. Z widoku usunięto płynną stal, widać obszar topienia metalu tzw. *mushy zone*.

8.5. Hartowanie olejem

W ramach projektu „Zaawansowane techniki wytwarzania przekładni lotniczych" został użyty aktualnie zaimplementowany moduł siatki obliczeniowej do modelowania procesu przemysłowego hartowania olejem elementów silnika lotniczego. Bardzo skomplikowana geometria składająca się z wielu części i różnych materiałów oraz różnorodne warunki brzegowe między nimi stawiają wysokie wymagania dla modułu siatki obliczeniowej. Ze względu na niejawność danych, ilustracje na rys. 8.11 pokazują tylko zadania kontrolne. W przypadku (a) hartowany obiekt jest zanurzony w chłodziwie, które bez wymuszonej konwekcji go chłodzi. Ruch chłodziwa jest efektem zmiany gęstości pod wpływem odebrania ciepła od obiektu chłodzonego. W przypadku (b) chłodziwo jest wpompowywane w specjalny kanał przepływający przez chłodzony obiekt. Oba zjawiska mają charakter niestacjonarny, a szereg parametrów również charakter nieliniowy.



Rysunek 8.11: Hartowanie olejem (a) swobodne i (b)wymuszone; c) przykład siatki obliczeniowej na geometrii doświadczenia referencyjnego.

Rozdział 9

Podsumowanie

W niniejszej pracy przedstawiony został model formalny, czym jest zarządzanie siatką obliczeniową bazujący na wcześniejszych modelach dostępnych w literaturze. Zarządzanie zostało zdefiniowane jako zbiór konkretnych operatorów, które pozwalają modułom programowym na realizowanie różnych algorytmów z ich wykorzystaniem. Zostały zdefiniowane dokładne zbiory operatorów dla różnego rodzaju siatek. Zaimplementowano moduł zarządzania rozproszoną hybrydową siatką adaptacyjną MES dedykowany na równoległe architektury obliczeniowe, uwzględniając wektoryzację (rozdział 5), wielowątkowość z pamięcią wspólną (rozdział 6) oraz przetwarzanie rozproszone (rozdział 7). Zrealizowany moduł został wielokrotnie zwalidowany na przykładach akademickich i praktyki przemysłowej (rozdział 8). W czasie prac rozwiązano szereg problemów badawczych związanych z poprawności i wysoką wydajnością modelowania. Przeprowadzono także badania mające na celu zidentyfikowanie, które z operatorów mają najistotniejszy wpływ na przeprowadzanie symulacji adaptacyjną metodą elementów skończonych.

W efekcie przeprowadzonych analiz zostały opracowane nowe metody pozwalające na jednoczesną efektywną realizację dwóch istotnych operatorów siatki. Pierwszy realizuje podstawową funkcjonalność siatki, jaką jest dostarczanie informacji o geometrii elementów skończonych (rozdział 6.3). Drugi realizuje funkcjonalność bezpośrednio związaną z architekturami rozproszonymi, czyli globalne jednoznaczne identyfikowanie obiektu, bez konieczności synchronizacji czy komunikacji (rozdział 7). W wyniku przeprowadzonych analiz teoretycznych i doświadczeń numerycznych zostały wykazane ich bardzo dobre właściwości pod względem wydajności, a także pod kątem ich skalowalności i optymalizacji wykorzystania pamięci. Pomimo to, dla siatek korzystających z wierzchołków niestacjonarnych opisywane metody mają takie same wady jak ich odpowiedniki, skutkujące koniecznością częstego uaktualniania, co może mieć negatywny wpływ na wydajność stosowania.

Ponieważ określone zostało ściśle, czym jest zarządzanie siatką i zidentyfikowano, które jego elementy mają istotny wpływ na realizowanie modelowania numerycznego oraz opracowano nowe metody posiadające istotne zalety względem dotychczas

stosowanych, a także zrealizowano je w praktyce, można stwierdzić, iż cele pracy zostały osiągnięte w pełni.

Jednocześnie należy zauważyć, iż zdefiniowanie zarządzania siatką i operatorów siatki oraz przeprowadzone badania zagadnienie adaptacji w modelu z pamięcią wspólną i rozproszoną, otwierają nowe perspektywy badań. Dzięki nowatorskiej realizacji istotnych zagadnień, które mają potencjał, aby wielokrotnie zwiększyć wydajność obliczeniową operatorów siatki, otwierają się nowe perspektywy, między innymi związane z zastosowaniem akceleratorów i kart graficznych do przeprowadzania pewnych długotrwałych operacji na siatce obliczeniowej. Także możliwość niezależnego jednoznacznego identyfikowania obiektów, wraz ze zmniejszeniem dysproporcji wykorzystania pamięci do ilości obliczeń stwarza nowe perspektywy dla dalszego rozwoju zarządzania siatką w modelach pamięci wspólnej i rozproszonej, szczególnie pod kątem jednoczesnego realizowania operatorów adaptacji i de-adaptacji siatki obliczeniowej.

Porównanie z dotychczas szeroko stosowanymi rozwiązaniami, w tym z pakietem obliczeniowym PETSc, wskazują, że prezentowane rozwiązania mogą, w ramach swojej funkcjonalności, znacznie zwiększyć efektywność symulacji oraz ułatwić realizację zadań stawianych przed schematami zarządzania siatką obliczeniową. Ponieważ posiadają one podobne cechy co niektóre ze stosowanych powszechnie rozwiązań, a jednocześnie wykazano lepszą wydajność, mogą stanowić interesującą alternatywę szczególnie w realizacji szybkiego dostępu do współrzędnych elementów skończonych, jednoznacznego określania identyfikatorów globalnych oraz skompresowanego zapisu siatki obliczeniowej.

Bibliografia

[1] Acm gordon bell prize 2000. http://www.sc2000.org/bell/pastawrd.htm, 2000.

[2] Acm gordon bell prize 2006-2011. http://www.sighpc.org/recognition/bell-prize, 2011.

[3] *ANSYS FLUENT Theory Guide*. Nov. 2011.

[4] *PHAML User's Guide, Version 1. 0*. CreateSpace Independent Publishing Platform, 2013.

[5] Acm gordon bell prize 2013+. http://sc13.supercomputing.org/content/acm-gordon-bell-prize, 2014.

[6] M. Ainsworth and J. Coyle. Hierarchic finite element bases on unstructured tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 58(14):2103–2130, 2003.

[7] S. G. Akl and M. Nagy. *Parallel Computing. The Future of Parallel Computation.* Springer London, 2009.

[8] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[9] L. Anders, M. Kent-Andre, and W. Garth. Automated solution of differential equations by the finite element method. *Lecture Notes in Computational Science and Engineering vol. 84*, 2012.

[10] I. Babuska and M. Suri. The p and h-p versions of the finite element method, basic principles and properties. *SIAM Review Vol. 36, No. 4*, pages 578–632, 1994.

[11] J. Baert, A. Lagae, and P. Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 27–32, New York, NY, USA, 2013. ACM.

[12] S. Balay, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2014.

[13] D. S. Balsara and C. D. Norton. Highly parallel structured adaptive mesh refinement using parallel language-based approaches. *Parallel Computing*, 27(1–2):37 – 70, 2001. New Trends in High Performance Computing.

[14] K. Banaś. A model for parallel adaptive finite element software. *Domain Decomposition Methods in Science and Engineering, Vol. 40 of Lecture Notes in Computational Science and Engineering*, pages 159–166, 2004.

[15] K. Banaś. A modular design for parallel adaptive finite element computational kernels. *Computational Science – ICCS 2004, 4th International Conference, Krakow, Poland, June 2004, Proceedings, Part II, Vol. 3037 of Lecture Notes in Computer Science*, pages 155–162, 2004.

[16] K. Banaś, P. Cybułka, P. Macioł, K. Michalik, and P. Płaszewski. Towards using adaptive hybrid meshes in fem simulations of flow in artificial heart chambers. *Computer Methods in Materials Science : quarterly / Akademia Górniczo-Hutnicza ; ISSN 1641-8581*, page 190–195, 2011.

[17] K. Banaś, P. Cybułka, K. Michalik, and A. Siwek. Modelowanie procesu hartowania metodą elementów skończonych — finite element modeling of quenching processes. *KomPlasTech 2016 : informatyka w technologii metali : Wisła, 17–20 stycznia 2016 : program XXIII konferencji KomPlasTech : streszczenia prac.*, page 64–65, 2014.

[18] K. Banaś and K. Michalik. Design and development of an adaptive mesh manipulation module for detailed fem simulation of flows. *Procedia Computer Science ; ISSN 1877-0509. – 2010 vol. 1*, page 2037–2045, 2010.

[19] K. Banaś and K. Michalik. Design and development of an adaptive mesh manipulation module for detailed fem simulation of flows. *ICCS 2010 : proceedings of the 10 International Conference on Computational Science : Amsterdam, May 31 – June 2, 2010*, page 1–9, 2010.

[20] K. Banaś. *Zastosowanie adaptacyjnej metody elementów skończonych do obliczeń wielkiej skali*. Monografia - Politechnika Krakowska im. Tadeusza Kościuszki: Podstawowe Nauki Techniczne. Politechnika Krakowska, 2004.

[21] K. Banaś, K. Chłoń, P. Cybułka, K. Michalik, P. Płaszewski, and A. Siwek. *Adaptive Finite Element Modelling of Welding Processes*, pages 391–406. Springer International Publishing, Cham, 2014.

[22] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28, Jan. 2012.

[23] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[24] R. E. Bank and M. Holst. A new paradigm for parallel adaptive meshing algorithms. *SIAM J. Sci. Comput*, 22:1411–1443, 2003.

[25] R. Barik, J. Zhao, and V. Sarkar. Automatic vector instruction selection for dynamic compilation. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 573–574, New York, NY, USA, 2010. ACM.

[26] M. W. Beall, J. E. Flaherty, M. S. Shephard, and J. D. Teresco. Adaptive and parallel finite element computations on heterogeneous systems. In *PPSC*, 1999.

[27] M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64 – 84, 1989.

[28] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984.

[29] R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13(6):437 – 452, 1994.

[30] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and colorin. *Scientific Programming*, 20(2), 2012.

[31] C. Burstedde, L. Wilcox, and O. Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[32] J. Chen and V. E. Taylor. Parapart: Parallel mesh partitioning tool for distributed systems. In *In Proc. IRREGULAR'99*, pages 12–111, 1999.

[33] P. G. Ciarlet. *Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[34] H. Cougny, M. Shephard, and C. Özturan. Parallel three-dimensional mesh generation on distributed memory mimd computers. *Engineering with Computers*, 12(2):94–106, 1996.

[35] T. Coupez, H. Digonnet, and R. Ducloux. Parallel meshing and remeshing. *Applied Mathematical Modelling*, 25(2):153 – 175, 2000. Dynamic load balancing of mesh-based applications on parallel.

[36] T. A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions On Mathematical Software*, 30(2):196–199, June 2004.

[37] H. L. De Cougny and M. S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46(7):1101–1125, 1999.

[38] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A general object oriented framework for discretizing nonlinear evolution equations. *Proceedings of The 1st Kazakh-German Advanced Research Workshop on Computational Science and High Performance Computing*, 2005.

[39] L. Demkowicz. *Computing with hp Finite Elements. I.One- and Two-Dimensional Elliptic and Maxwell Problems*. Chapman & Hall, 2006.

[40] L. Demkowicz, J. Kurtz, D. Pardo, M. Paszenski, W. Rachowicz, and A. Zdunek. *Computing with hp-ADAPTIVE FINITE ELEMENTS: Volume II Frontiers: Three Dimensional Elliptic and Maxwell Problems with Applications*. Chapman & Hall/CRC Applied Mathematics & Nonlinear Science. Taylor & Francis, 2007.

[41] E. D'Hollander, J. Dongarra, I. Foster, L. Grandinetti, and G. Joubert, editors. *Transition of HPC towards exascale computing*, volume 24 of *Advances in Parallel Computing*. IOS Press, 2013.

[42] L. Diachin, A. Bauer, B. Fix, J. Kraftcheck, K. Jansen, X. Luo, M. Miller, C. Ollivier-Gooch, M. S. Shephard, T. Tautges, and H. Trease. Interoperable mesh and geometry tools for advanced petascale simulations. *Journal of Physics: Conference Series*, 78(1):012015, 2007.

[43] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive fem. *Parallel Comput.*, 26(12):1555–1581, Nov. 2000.

[44] Y. Efendiev and T. Y. Hou. *Multiscale finite element methods : theory and applications*. Surveys and tutorials in the applied mathematical sciences. Springer, New York, NY, 2009.

[45] J. E. Flaherty, M. S. Shephard, H. L. De Cougny, C. Ozturan, C. L. Bottasso, and M. W. Beall. Parallel automated adaptive procedures for unstructured meshes, 1995.

[46] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, Sept. 1972.

[47] T. M. Forum. Mpi: A message passing interface, 1993.

[48] R. V. Garimella. Mesh data structure selection for mesh generation and FEA applications. *International Journal of Numerical Methods in Engineering*, 55(4):451–478, Oct. 2002.

[49] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing*, 33(10-11):685–699, 2007.

[50] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[51] B. T. Gunney, A. M. Wissink, and D. A. Hysom. Parallel clustering algorithms for structured {AMR}. *Journal of Parallel and Distributed Computing*, 66(11):1419 – 1430, 2006.

[52] J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[53] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[54] J. Hogg. Microsoft corporation: Auto-vectorizer in visual studio 2012 – rules, 2012.

[55] R. D. Hornung, A. M. Wissink, and S. R. Kohn. Managing complex data and geometry in parallel structured amr applications. *Eng. with Comput.*, 22(3):181–195, Dec. 2006.

[56] R. Huber, , H. A. Mayer, and R. Schwaiger. netgen - a parallel system generating problem-adapted topologies of artificial neural networks by means of genetic algorithms, 1995.

[57] T. Hughes. *The finite element method: linear static and dynamic finite element analysis*. Dover Civil and Mechanical Engineering Series. Dover Publications, 2000.

[58] Intel. A guide to vectorization with intel c++ compilers, 2010.

[59] Y. Ito, A. M. Shih, A. K. Erukala, B. K. Soni, A. Chernikov, N. P. Chrisochoides, and K. Nakahashi. Parallel unstructured mesh generation by an advancing front method. *Math. Comput. Simul.*, 75(5-6):200–209, Sept. 2007.

[60] M. Jarkowski, M. Woodgate, G. Barakos, and J. Rokicki. Towards consistent hybrid overset mesh methods for rotorcraft cfd. *International Journal for Numerical Methods in Fluids*, 74(8):543–576, 2014.

[61] J. Jeong, T. Goktekin, and X. Wu. An interactive parallel multigrid fem simulator. *Medical Simulation, Vol. 3078 of Lecture Notes in Computational Science*, pages 124–133, 2004.

[62] T. Jurczyk and B. Głut. Tree structures for adaptive control space in 3d meshing. *Computer Science*, 17(4):541, 2017.

[63] V. Kandasamy. Parallel fem simulation using gpus. 2012.

[64] C. Kavouklis and Y. Kallinderis. Parallel adaptation of general three-dimensional hybrid meshes. *J. Comput. Phys.*, 229(9):3454–3473, May 2010.

[65] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. `http://dx.doi.org/10.1007/s00366-006-0049-3`.

[66] J. Kitowski, M. Turala, K. Wiatr, and L. Dutka. In M. Bubak, T. Szepieniec, and K. Wiatr, editors, *PL-Grid*, Lecture Notes in Computer Science, pages 1–14. Springer.

[67] S. R. Kohn, S. B. Baden, and S. Kohn. Parallel software abstractions for structured adaptive mesh methods.

[68] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, L. Kale, and P. Ricker. Scalable algorithms for distributed-memory adaptive mesh refinement. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 100–107, Oct 2012.

[69] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. *27th IEEE International Parallel & Distributed Processing Symposium*, 2013.

[70] L. Lämmer and M. Burghardt. Parallel generation of triangular and quadrilateral meshes. *Advances in Engineering Software*, 31(12):929 – 936, 2000.

[71] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330 – 354, 2000.

[72] L. Madej, P. Spytkowski, K. Michalik, F. Krużel, P. Macioł, K. Banaś, and M. Pietrzyk. Study on development of an adaptive finite element – cellular automata model for austenite-ferrite phase transformation. *ECCM 2010 : IV European Conference on Computational Mechanics : solids, structures and coupled problems in engineering : France : May 16–21, 2010 / ECCOMAS European Community on Computational Methods in Applied Sciences.*, page 1–2, 2010.

[73] M. Mantyla. *Introduction to Solid Modeling*. W. H. Freeman & Co., New York, NY, USA, 1988.

[74] P. J. Matuszyk, M. Sieniek, and M. Paszyński. Fully automatic 2d hp-adaptive finite element method for non-stationary heat transfer. *Procedia Computer Science*, 51:2883 – 2887, 2015. International Conference On Computational Science, {ICCS} 2015Computational Science at the Gates of Nature.

[75] K. Michalik. Biblioteka do kompresji siatki obliczeniowej (c++), 2016.

[76] K. Michalik, K. Banaś, and P. Macioł. Design of a parallel modular framework for multi-scale finite element simulations of fluid-structure interaction : [poster]. *KKMP 2010 : XIX Krajowa Konferencja Mechaniki Płynów = 19th Polish National Fluid Dynamics Conference : 05–09. September 2010, Poznań, Poland : submitted papers. ISBN 978-83-89949-90-5.*, page 68, 2010.

[77] K. Michalik, K. Banaś, P. Płaszewski, and P. Cybułka. Modfem: modular framework for generic parallel fem simulations. *Cracow'12 Grid Workshop : October 22–24, 2012, Krakow, Poland : proceedings. ISBN: 978-83-61433-06-4*, page 83–84, 2012.

[78] K. Michalik, K. Banaś, P. Płaszewski, and P. Cybułka. A parallel modular framework for multi-scale and multi-physics finite element simulations of fluid flow. *KKMP 2012 : XX fluid mechanics conference : Gliwice, 17–20 September 2012 : book of abstracts / eds. Tadeusz Chmielniak, Włodzimierz Wróblewski ; Silesian University of Technology. Institute of Power Engineering & Turbomachinery, Committee of Mechanics of PAS. Prace Naukowe, Monografie, Konferencje / Politechnika Śląska. Instytut Maszyn i Urządzeń Energetycznych ; ISSN 1506-9702 ; z. 29*, page 181–182, 2012.

[79] K. Michalik, K. Banaś, P. Płaszewski, and P. Cybułka. Modfem : a computational framework for parallel adaptive finite element simulations. *Computer Methods in Materials Science ; ISSN 1641-8581*, Vol. 13, No. 1:3–8, 2013.

[80] K. Michalik, K. Banaś, P. Płaszewski, and P. Cybułka. Modular fem framework "modfem" for generic scientific parallel simulations. *Computer Science ; ISSN 1508-2806*, Vol. 14 (3):513–528, 2013.

[81] K. Michalik, F. Krużel, P. Macioł, and K. Banaś. Parallel mesh management for finite element simulations of complex problems. *ECCM 2010: IV European Conference on Computational Mechanics : solids, structures and coupled problems in engineering : France : May 16–21, 2010 / ECCOMAS European Community on Computational Methods in Applied Sciences.*, page 1–2, 2010.

[82] K. Michalik, P. Macioł, and K. Banaś. Mesh adaptations for accurate fem modelling of flows. *CMS'09 : Computer Methods and Systems : 7 conference : 26–27 November 2009, Kraków, Poland*, page 431–434, 2009.

[83] A. A. Mirin, R. H. Cohen, B. B. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the ibm-sp system, 1999.

[84] W. F. Mitchell and M. A. Mcclain. A survey of hp-adaptive strategies for elliptic partial differential equations.

[85] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[86] I. NVIDIA. Performance specification for nvidia tesla p100 accelerators. `http://www.nvidia.com/object/tesla-p100.html`. Dostęp: 2016-11-10.

[87] L. Oliker, R. Biswas, and H. N. Gabow. Parallel tetrahedral mesh adaptation with dynamic load balancing. *Parallel Computing*, 26:1583–1608, 1999.

[88] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[89] M. Parashar, James, and C. Browne. Systems engineering for high performance computing software: The hdda/dagh infrastructure for implementation of parallel structured adaptive mesh refinement. In *In Structured Adaptive Mesh Refinement Grid Methods, IMA Volumes in Mathematics and its Applications*, pages 1–18. Springer-Verlag, 1997.

[90] Y. M. Park and O. J. Kwon. A parallel unstructured dynamic mesh adaptation algorithm for 3-d unsteady flows. *International Journal for Numerical Methods in Fluids*, 48(6):671–690, 2005.

[91] A. Patra, J. Long, and A. Laszloffy. Efficient parallel adaptive finite element methods using self-scheduling data and computations. In P. Banerjee, V. Prasanna, and B. Sinha, editors, *High Performance Computing – HiPC'99*, volume 1745 of *Lecture Notes in Computer Science*, pages 359–363. Springer Berlin Heidelberg, 1999.

[92] B. Patzák and D. Rypl. Object-oriented, parallel finite element framework with dynamic load balancing. *Advances in Engineering Software vol. 47*, pages 35–50, 2012.

[93] M. Pietrzyk, K. Banaś, A. Milenin, M. Kopernik, D. Szeliga, P. Macioł, P. Cybułka, and K. Michalik. Model numeryczny sztucznej komory wspomagania pracy serca — [numerical model of ventricular assist device]. *Technologie inżynierii materiałowej i technologie metrologiczne dla potrzeb polskich protez serca : program Polskie Sztuczne Serce : praca zbiorowa. Monografia oprac. i wyd. w ramach realizacji programu wieloletniego „Polskie Sztuczne Serce" na lata 2007–2012. — ISBN: 978-83-63310-04-2*, page 373–439, 2012.

[94] A. Pracownia Obliczeń Równoległych i Rozproszonych, WIMiIP. Dokumentacja online projektu modfem. `http://www.modfem.agh.edu.pl/modfem_doc/`. Dostęp: 2016-11-10.

[95] A. Pracownia Obliczeń Równoległych i Rozproszonych, WIMiIP. Repozytorium projektu modfem. `https://git.plgrid.pl/projects/MODFEM/repos/modfem2015/browse/src/`. Dostęp: 2016-11-10.

[96] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[97] R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers.* Apress, Berkely, CA, USA, 1st edition, 2013.

[98] L. Rainald, C. Jose, and M. Marshal. Parallel unstructured grid generation. *Computer Methods in Applied Mechanics and Engineering*, 95(3):343 – 357, 1992.

[99] T. Rauber and R. Gudula. *High Performance Computational Science and Engineering.* Springer US, 2005.

[100] J. Reinders. *Intel Threading Building Blocks.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[101] J.-F. Remacle, O. Klaas, J. E. Flaherty, and M. S. Shephard. Parallel algorithm oriented mesh database. *Int. J. Numer. Meth. Engng*, 58:349–374, 2001.

[102] M. M. Resch. *Sustained Simulation Performance 2012. Proceedings of the joint Workshop on High Performance Computing on Vector Systems, Stuttgart (HLRS), and Workshop on Sustained Simulation Performance, Tohoku University.* Springer Berlin Heidelberg, 2012.

[103] J. Rokicki, J. Żółtak, D. Drikakis, and J. Majewski. Parallel performance of overlapping mesh technique for compressible flows. *Future Generation Computer Systems*, 18(1):3 – 15, 2001. I. High Performance Numerical Methods and Applications. II. Performance Data Mining: Automated Diagnosis, Adaption, and Optimization.

[104] R. Said, N. Weatherill, K. Morgan, and N. Verhoeven. Distributed parallel delaunay mesh generation. *Computer Methods in Applied Mechanics and Engineering*, 177(1–2):109 – 125, 1999.

[105] E. Seegyoung Seol and M. S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Eng. with Comput.*, 22(3):197–213, Dec. 2006.

[106] P. M. Selwood and M. Berzins. Parallel unstructured tetrahedral mesh adaptation: algorithms, implementation and scalability. *Concurrency - Practice and Experience*, 11(14):863–884, 1999.

[107] M. S. Shephard and S. Seegyoung. *Flexible Distributed Mesh Data Structure for Parallel Adaptive Analysis.* Wiley, 2009.

[108] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 3:1–3:11, New York, NY, USA, 2011. ACM.

[109] A. Siwek, K. Banaś, K. Chłoń, P. Cybułka, K. Michalik, and J. Bielański. Finite element modelling of laser welding for materials with different properties. *XXI FMC : XXI Fluid Mechanics Conference : Krakow, 15–18 June 2014; ISBN: 978-83-7464-704-5*, page 98, 2014.

[110] A. Siwek, K. Banaś, J. Rońda, K. Chłoń, P. Cybułka, K. Michalik, and P. Płaszewski. Modelowanie procesu spawania z wykorzystaniem programu adaptacyjnej metody elementów skończonych modfem — computer modelling of welding process with adaptive finite element code modfem. *Hutnik Wiadomości Hutnicze : czasopismo naukowo-techniczne poświęcone zagadnieniom hutnictwa ; ISSN 1230-3534*, page 248–253, 2013.

[111] A. Siwek, J. Rońda, K. Banaś, P. Cybułka, K. Michalik, and P. Płaszewski. Modeling of inconel 625 tig welding process — modelowanie spawania metodą TIG stopu inconel 625. *Computer Methods in Materials Science : quarterly / Akademia Górniczo-Hutnicza ; ISSN 1641-8581*, page 181–187, 2013.

[112] P. Solin. *Partial differential equations and the finite element method*. Pure and applied mathematics : a Wiley-interscience series of texts, monographs and tracts. Hoboken, N.J. Wiley-Interscience, 2006.

[113] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[114] H. Sutter. The free lunch is over, 2005.

[115] H. Sutter. Welcome to the Jungle, 2011.

[116] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst. MOAB: a mesh-oriented database. SAND2004-1592, Sandia National Laboratories, Apr. 2004. Report.

[117] J. Teresco, M. Beall, J. Flaherty, and M. Shephard. A hierarchical partition model for adaptive finite element computation. *Computer Methods in Applied Mechanics and Engineering*, 184(2–4):269 – 285, 2000.

[118] V. Thomée. *Galerkin Finite Element Methods for Parabolic Problems (Springer Series in Computational Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[119] B. Topping and B. Cheng. Parallel and distributed adaptive quadrilateral mesh generation. *Computers & Structures*, 73(1–5):519 – 536, 1999.

[120] J. S. Vetter. *Contemporary High Performance Computing: From Petascale Toward Exascale*. Chapman & Hall/CRC, 2013.

[121] R. W. *Metoda elementów skończonych i brzegowych. Podstawy kontroli błędu i adaptacji.* Monografia - Politechnika Krakowska im. Tadeusza Kościuszki: Podstawowe Nauki Techniczne. Politechnika Krakowska, 2012.

[122] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. Technical report, Parallel Comput, 2000.

[123] N. Whitehead and A. Fit-florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus.

[124] L. Zhao and S. Zhao. Virtual design of bike based on solidworks. In *EMEIT*, pages 129–132. IEEE, 2011.

[125] O. Zienkiewicz, R. Taylor, and J. Zhu. *The Finite Element Method: Its Basis and Fundamentals: Its Basis and Fundamentals.* Elsevier Science, 2005.

[126] O. C. Zienkiewicz and J. Z. Zhu. The superconvergent patch recovery and a posteriori error estimates. part 1: The recovery technique. *International Journal for Numerical Methods in Engineering*, 33(7):1331–1364, 1992.

[127] C. Özturan, H. L. deCougny, M. S. Shephard, and J. E. Flaherty. Parallel adaptive mesh refinement and redistribution on distributed memory computers. Technical report, Comput. Methods Appl. Mech. Engrg, 1993.

Spis rysunków

Spis tabel