



**POLITECHNIKA
GDAŃSKA**

WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI I INFORMATYKI



Imię i nazwisko studenta: Michał Niegrzybowski
Nr albumu: 143303
Studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność/profil: -

Imię i nazwisko studenta:
ŁUKASZ ROMANOWSKI
Nr albumu: 143330
Studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność/profil: -

Imię i nazwisko studenta: Błażej Galiński
Nr albumu: 143219
Studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność/profil: -

PROJEKT DYPLOMOWY INŻYNIERSKI

Tytuł projektu w języku polskim: Porównanie technologii dla obliczeń równoległych na akceleratorach Intel Xeon Phi

Tytuł projektu w języku angielskim: Comparison of technologies for parallel computations on Intel Xeon Phi

Potwierdzenie przyjęcia projektu	
Opiekun projektu	Kierownik Katedry/Zakładu
<i>podpis</i>	<i>podpis</i>
dr hab. inż. Paweł Czarnul	

Data oddania projektu do dziekanatu:

STRESZCZENIE

Poniższa praca opisuje zagadnienia z dziedziny Informatyki związane z przetwarzaniem równoległym. Jest to rodzaj obliczeń, w którym wiele instrukcji jest wykonywanych równocześnie. Obecnie stosowanych jest wiele technologii pozwalających na zastosowanie takiego modelu, jednak celem poniższej pracy jest porównanie Message Passing Interface, Open Multi-Processing oraz Open Computing Language na akceleratorach Intel Xeon Phi 5120. Zestawienie zostało zrealizowane dla trzech najważniejszych obszarów problemowych związanych z tego typu obliczeniami. Jest to duża liczba działań, ogrom danych oraz mnogość komunikacji. Do zobrazowania problemów, w ramach opisanego projektu inżynierskiego, napisano łącznie dwanaście aplikacji, po cztery do reprezentacji każdego ze wspomnianych obszarów. Wśród implementacji znajduje się wersja szeregową, która była podstawą do stworzenia wersji MPI, OpenMP oraz OpenCL, tak, aby ich porównanie było rzetelne i niezakłamane. Warto zauważyć, że w realizowanym projekcie zastosowano model offload, w którym właściwe algorytmy obliczenia są przeprowadzane jedynie na koprocesorze.

Programy reprezentujące problemy opierają się na zobrazowaniu zbioru Mandelbrota (duża liczba prostych obliczeń), tworzeniu obrazu złożonego z prostokątów z wykorzystaniem algorytmu genetycznego (obliczenia na obszernych danych), stworzeniu aplikacji typu klient-serwer (spora liczba komunikacji). Każda z implementacji, jak również wyniki porównania zostały opisane w poniższej pracy. Dokument zawiera również opis realizacji projektu oraz doświadczenia z nim związane.

Na końcu pracy przedstawiono wyniki porównania oraz wyróżniono najlepszą technologię z uwagi na testowane obszary problemów. Zaproponowano optymalizacje, które mogłyby dopełnić pracę pod względem merytorycznym.

Słowa kluczowe:

Przetwarzanie równoległe, C++, MPI, OpenCL, OpenMP, Intel Xeon Phi

Dziedzina nauki i techniki, zgodnie z wymogami OECD:

Nauki inżynieryjne i techniczne, Inżynieria informatyczna

ABSTRACT

Following document describes the issues, in the field of Computer Science, related to parallel computing. This is a type of calculation, wherein many instructions are executed simultaneously. Currently there are multiple existing technologies, which can be used in implementation of such a model, but the purpose of this paper is to compare the Message Passing Interface, Open Multi-Processing and Open Computing Language on the accelerator Intel Xeon Phi 5120. The comparison was made for the three main problem areas associated with this type of computations. Those are a large number of calculations, the enormity of the data and a multitude of the communication. In order to illustrate this problem, in the context of this paper, the total number of twelve applications, four for each of the mentioned areas, had been made. Among implementation there is a serial version, which was the basis used for creation of a MPI, OpenMP and OpenCL versions, so that the comparison was fair and correct. It is worth noting that in the project offload model was used, so the appropriate algorithm computations were performed only on the coprocessor site.

Applications, which illustrate the problems, are performing vizualization of Mandelbrot set (a large number of simple computations), creation of an image consisting of rectangles which are generated using genetic algorithm (calculations on extensive data), and the simulation of a client-server architecture (high intensity of communication). Each of some implementations, as well as the results of the comparison, are described in the following document. Which also contains a description of the bachelors and the experience associated with it.

At the end of this paper, the results of the comparison were presented, and the best technologies, regarding selected problem areas, were pointed. Optimizations, which could complement the essence of this paper, were also proposed.

Keywords:

Parallel Computing, C++, MPI, OpenCL, OpenMP, Intel Xeon Phi

Field of science and technology, according to the OECD:

Engineering and technology, Information engineering

Wykaz ważniejszych oznaczeń i skrótów.....	7
1. Wprowadzenie (B. Galiński, Ł. Romanowski, M. Niegrzybowski)	9
1.1. Kontekst pracy	9
1.2. Cel pracy.....	10
1.3. Ograniczenia.....	10
1.4. Zakres pracy	10
1.5. Podział pracy implementacyjnej	11
2. Zagadnienia przetwarzania równoległego (B. Galiński)	12
2.1. Ogólne problemy i ich rozwiązanie	12
2.2. Zastosowanie	14
3. Xeon Phi (M. Niegrzybowski)	16
3.1. Architektura	16
3.2. Dostęp do maszyny	20
4. Opis Porównywanych technologii	22
4.1. MPI (B. Galiński)	22
4.2. OpenMP (Ł. Romanowski)	24
4.3. OpenCL (M. Niegrzybowski)	26
5. Opis implementowanych algorytmów (M. Niegrzybowski)	30
5.1. Duża liczba prostych obliczeń (algorytm generujący obraz zbioru Mandelbrota)	30
5.2. Duża liczba komunikacji (aplikacja typu klient-serwer)	31
5.3. Obliczenia na dużej liczbie danych (algorytm genetyczny)	34
6. Opis implementacji algorytmu Mandelbrota	36
6.1. Szeregowy (B. Galiński)	36
6.2. MPI (B. Galiński, M. Niegrzybowski)	38
6.3. OpenMP (M. Niegrzybowski, Ł. Romanowski)	42
6.4. OpenCL (Ł. Romanowski)	44
7. Opis implementacji aplikacji typu klient-serwer	49
7.1. Szeregowy (B. Galiński)	49
7.2. MPI (M. Niegrzybowski)	50
7.3. OpenMP (B. Galiński)	53
7.4. OpenCL (Ł. Romanowski)	55
8. Opis implementacji algorytmu genetycznego	59
8.1. Szeregowy (M. Niegrzybowski)	59
8.2. MPI (Ł. Romanowski)	62
8.3. OpenMP (Ł. Romanowski)	65
8.4. OpenCL (M. Niegrzybowski)	68
9. Wyniki porównania technologii (B. Galiński, Ł. Romanowski, M. Niegrzybowski)	74
9.1. Algorytm generujący obraz zbioru Mandelbrota	74
9.1.1. MPI	74
9.1.2. OpenMP	76

9.1.3.	OpenCL.....	77
9.1.4.	Porównanie wyników każdej technologii	78
9.2.	Aplikacja klient-serwer	79
9.2.1.	MPI.....	79
9.2.2.	OpenMP	80
9.2.3.	OpenCL.....	82
9.2.4.	Porównanie wyników	83
9.3.	Algorytm genetyczny.....	83
9.3.1.	MPI.....	83
9.3.2.	OpenMP	85
9.3.3.	OpenCL.....	86
9.3.4.	Porównanie wyników każdej technologii	86
10.	Opis prac implementacyjnych	88
10.1.	Wykorzystane narzędzia i technologie (B. Galiński, Ł. Romanowski)	88
10.2.	Ograniczenia (B. Galiński, M. Niegrybowski)	90
10.3.	Testy jednostkowe (M. Niegrybowski)	91
11.	Opis realizacji projektu (B. Galiński)	93
11.1.	Plan komunikacji w zespole i jego realizacja	93
11.2.	Plan przedsięwzięcia.....	95
12.	Zebrane doświadczenia	96
12.1.	Praca w zespole (B. Galiński, Ł. Romanowski, M. Niegrybowski)	96
12.2.	Metodyka i system pracy (B. Galiński, Ł. Romanowski, M. Niegrybowski)	97
12.3.	Narzędzia (B. Galiński, Ł. Romanowski)	98
12.4.	Testowane technologie	99
12.4.1.	MPI (B. Galiński)	100
12.4.2.	OpenMP (Ł. Romanowski)	101
12.4.3.	OpenCL (M. Niegrybowski)	101
13.	Podsumowanie (B. Galiński, Ł. Romanowski, M. Niegrybowski)	103
	Wykaz literatury	105
	Wykaz rysunków.....	106
	Wykaz tabel	107
	Dodatek A: Czasy wykonania programu realizującego algorytm genetyczny (algorytm)	108
	Dodatek B: Czasy wykonania programu realizującego algorytm genetyczny (środowisko)	109
	Dodatek C: Czasy wykonania programu realizującego algorytm Mandelbrot.....	110
	Dodatek D: Czasy wykonania programu typu klient-serwer.....	111
	Dodatek E: Wykres przedstawiający działanie programu Mandelbrot w technologii OpenMP przy wartości zmiennej MIC_KMP_AFFINITY równej compact	112
	Dodatek F: Wykres przedstawiający działanie programu Mandelbrot w technologii OpenMP przy wartości zmiennej MIC_KMP_AFFINITY równej scatter	113

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

MPI	– Message Passing Interface – środowisko zapewniające komunikację pomiędzy procesami w środowisku rozproszonym
OpenCL	– Open Computing Language – niskopoziomowe API przeznaczone do programowania aplikacji heterogenicznych
OpenMP	– Open Multi-Processing – interfejs umożliwiający programowanie aplikacji wielowątkowych, o pamięci współdzielonej
Offload	– Biblioteka firmy Intel, realizowana w postaci zbioru dyrektyw, umożliwiająca wykonanie kodu programu po stronie koprocatora
TBB	– Intel Threading Building Blocks – biblioteka szablonów firmy Intel umożliwiająca realizację zrównoleglonej aplikacji
MIC	– Many Integrated Core – architektura firmy Intel, polegająca na wielordzeniowości
pipeline	– potok
master	– główny wątek/proces w aplikacji o modelu master-slave
slave	– wątek/proces podrzędny podlegający pod wątek/proces główny (master) w aplikacji modelu master-slave
API	– Application Programming Interface – jest to zbiór procedur udostępnionych przez bibliotekę
ccNUMA	– cache coherent Non-Uniform Memory Access – rodzaj architektury polegający na niejednorodnym dostępie do danych z zapewnieniem spójności pamięci podręcznej
cache	– pamięć podręczna
scheduler	– mechanizm kolejujący wykonywanie procesów
prefetcher	– mechanizm zapewniający pobieranie danych przed wykonaniem instrukcji wykorzystujących je
stand-up	– spotkanie na którym omawiane są postępy w pracy, plan pracy oraz ewentualne problemy. Wykorzystywany przy metodykach zwinnych
GOLS	– protokół wykorzystywany w Tag Directory, opisujący stan linii pamięci podręcznej w aspekcie wszystkich rdzeni
MESI	– Modified, Exclusive, Shared, Invalid – protokół opisujący stan linii pamięci podręcznej w danym rdzenia.
TD	– Tag Directory – Komponent koprocatora Intel Xeon Phi, zapewniający spójność linii pamięci podręcznej.
DMA	– Direct Memory Acces – umożliwia transfer danych pomiędzy urządzeniami z pominięciem procesora.
PNG	– Portable Network Graphic – rastrowy format plików
NaN	– Not a Number – wyrażenie określające wartość która nie jest liczbą
kernel	– jądro obliczeniowe wykorzystywane w technologii OpenCL
SP	– Single Precision - operacje pojedynczej precyzji

DP	– Double Precision - operacje podwójnej precyzji
VPU ISA	– Vector Microarchitecture Instruction Set Architecture
SIMD	– Single Instruction Multiple Data - oznacza pojedynczą operacją na wielu danych, tzw. operacje wektorowe.
hardware	– sprzęt komputerowy
software	– oprogramowanie
fork	– podział, tworzenie kopii procesu lub wątku
join	– złączenie, łączenie kopii procesu lub wątków

1. WPROWADZENIE

Problem związany z wydajnością procesorów jest obecnie powszechnie znany. Człowiek od czasów powstania pierwszego procesora starał się, aby go ulepszyć, usprawnić, tak, aby można było rozwiązywać dzięki niemu coraz bardziej złożone problemy. Jedną z innowacji było wprowadzenie wielowątkowości na pojedynczej jednostce wykonawczej procesora – rdzeniu. Jest to możliwość jednoczesnego wykonywania obliczeń na rdzeniu przez wiele wątków, gdyż praca jednego nie wykorzystuje w pełni możliwości jednostki wykonawczej procesora. Aby przyspieszyć obliczenia na procesorze, zaczęto również zwiększać jego taktowanie, tak, aby był w stanie wykonać większą liczbę instrukcji w tym samym czasie. Niestety, przyspieszanie taktowania zegara ma swoje ograniczenia techniczne. Coraz szybsza praca jednej jednostki łączy się z większą ilością wydzielanego przez nią ciepła, czego nie wytrzymują wrażliwe układy elektroniczne i jednostka ulega przegrzaniu. Chcąc, mimo wszystko, nadal poprawiać wydajność rozwiązywania problemów, zaczęto zwiększać liczbę rdzeni tak, aby komunikując się ze sobą, mogły rozwijać bardziej skomplikowane problemy.

Wielowątkowość, a także komunikację pomiędzy procesami, niekoniecznie tego samego urządzenia, definiuje problematyka przetwarzania równoległego. Jest to forma wykonywania obliczeń, w której wybrana pula instrukcji wykonywana jest równocześnie. Algorytmy przetwarzania są dużo bardziej skomplikowane. Wprowadzają konieczność odpowiedniego sposobu rozłożenia obliczeń, problemy związane z synchronizacją i dostępem do tej samej porcji danych oraz komunikacji, aby jak najwydajniej wykorzystać dostępne urządzenia. Z tego powodu dużo łatwiej popełnić błąd, jest jednak wiele technologii, które w różny sposób ułatwiają pracę w środowisku równoległym. Są to między innymi MPI, OpenCL, OpenMP, Intel Click Plus, SHMEM, Unified Parallel C czy przeznaczona wyłącznie dla układów graficznych firmy Nvidia, technologia CUDA. W pracy skupiono się na porównaniu pierwszych trzech technologii na nowoczesnym koprocesorze Intel Xeon Phi. Są to dodatkowe karty podłączane do komputera za pomocą złącz PCI Express, które znacznie zwiększają jego wydajność przetwarzania równoległego. Koprocesory te składają się z bardzo wielu rdzeni o niskim taktowaniu, dzięki temu dla wybranych problemów są znacznie wydajniejsze od zwykłych procesorów. W tym miejscu należy jednak zauważyć, że dla dużej liczby algorytmów, a także w przypadku, gdy programy równoległe napisane są w nieoptymalny sposób, czasy wykonania programów na Intel Xeon Phi, mogą być dłuższe w porównaniu z czasami pracy na zwykłych procesorach.

1.1. Kontekst pracy

Poniższa praca obejmuje porównanie technologii MPI, OpenCL oraz OpenMP na koprocesorach Intel Xeon Phi. Dobór ich spowodowany jest faktem, iż obecnie są to najpopularniejsze technologie przetwarzania równoległego na rynku. Sens takiego porównania wynika z zastosowanych rozwiązań każdej z nich w kwestii komunikacji oraz przesyłania danych. Główna różnica polega na wykorzystywanych do obliczeń części programów. Należy zauważyć, że główną zaletą MPI jest praca w środowisku rozproszonym, z tego względu programy pisane w tej technologii tworzą odrębne procesy, które umożliwiają takie rozwiązanie. Inaczej jest w

przypadku OpenMP oraz OpenCL. Te programy z założenia powinny być szybsze, gdyż operują na pamięci współdzielonej, a obliczenia wykonywane są przez mniej zasobochłonne wątki, lecz bez dodatkowych technologii mogą korzystać jedynie z zasobów jednego urządzenia. Każde z tych rozwiązań jest bardziej lub mniej odpowiednie przy zastosowaniu dla wybranych problemów algorytmicznych, co tworzy kontekst niniejszej pracy.

1.2. Cel pracy

Celem niniejszego projektu jest ukazanie podobieństw i różnic pomiędzy wybranymi technologiami przetwarzania równoległego dla najważniejszych problemów algorytmicznych z nim związanych. Porównanie oparto na czasach wykonania zarówno całych programów, jak i ich części składowych, takich jak komunikacja lub obliczenia na koprocesorze Intel Xeon Phi.

Dzięki tej pracy chciano pokazać użytkownikom badanych technologii, na którą z nich powinni się zdecydować w zależności od ich potrzeb. Skupiono się na aspektach zarówno wydajnościowych, łatwości pisania programów oraz na zrozumieniu stosowanych modeli w opisywanych technologiach.

1.3. Ograniczenia

Chcąc ukazać najważniejsze aspekty i problemy przetwarzania równoległego, z tego względu w napisanych programach pominięto kwestie interfejsu użytkownika testowanych programów. Są one uruchamiane z poziomu konsoli na maszynach laboratoryjnych.

Ograniczeniem podczas pisania niniejszej pracy było również użycie ogólnodostępnej maszyny Katedry Architektury Systemów Komputerowych apl12, w której skład wchodzi koprocesor Intel Xeon Phi. Ograniczenia te łączyły się z dostępem do danej maszyny, która w wybranych okresach z przyczyn niezależnych była okresowo wyłączana. Problemem też stanowił dostęp zdalny z komputerów domowych, bowiem wymagał on sprawnie działającej sieci internetowej.

Kolejnym ograniczeniem przedstawionej pracy inżynierskiej są zainstalowane technologie na maszynie apl12 posiadającej koprocesor Intel Xeon Phi. Dostępne wersje szczególnie ograniczyły rozwiązania zastosowane w OpenCL oraz OpenMP, jednak szczegółowe utrudnienia implementacji zostały opisane w dalszej części pracy.

1.4. Zakres pracy

Aby osiągnąć cel, napisano trzy programy, z których każdy symuluje inne utrudnienie. Jeden z nich wykonuje dużą ilość obliczeń, drugi przeprowadza działania na mnogiej liczbie danych, natomiast trzeci przedstawia problem potrzeby częstej komunikacji. Programy dla każdej z technologii uruchomiono z różnymi parametrami, definiując liczbę procesów lub wątków, a także wielkość danych, na jakich mają być przeprowadzane obliczenia oraz zmierzono czasy ich wykonania. Następnie korzystając ze zdobytej wiedzy oraz specjalistycznych aplikacji do profilowania uruchamianych programów określono, która z technologii MPI, OpenMP lub OpenCL, jest odpowiednia dla zdefiniowanych wcześniej problemów lub ich części.

Warto zauważyć, że programy napisano w modelu wykonawczym, w którym wszystkie obliczenia wykonywane są jedynie na koprocesorach Intel Xeon Phi. Główny proces programu uruchomionego na komputerze laboratoryjnym odpowiada jedynie za dostarczenie i odebranie danych z akceleratora. Dokładne rozwiązanie stosowanego modelu wykonawczego dla każdej z testowanych technologii opisane jest w dalszej części pracy.

1.5. Podział pracy implementacyjnej

We wstępie warto zaznaczyć wkład autorów w implementację poszczególnych części projektu. Udział każdego uczestnika w zależności od realizowanych wersji został przedstawiony na poniższej liście:

- implementacja algorytmu zbioru Mandelbrota:
 - wersja szeregową – Błażej Galiński,
 - MPI – Błażej Galiński, Michał Niegrybowski,
 - OpenMP – Michał Niegrybowski, Łukasz Romanowski,
 - OpenCL – Łukasz Romanowski,
- implementacja aplikacji typu klient-serwer:
 - wersja szeregową – Błażej Galiński,
 - MPI – Michał Niegrybowski,
 - OpenMP – Błażej Galiński,
 - OpenCL – Łukasz Romanowski,
- implementacja algorytmu genetycznego:
 - wersja szeregową – Michał Niegrybowski,
 - MPI – Łukasz Romanowski,
 - OpenMP – Łukasz Romanowski,
 - OpenCL – Michał Niegrybowski,
- aplikacja do przedstawienia wyników – Łukasz Romanowski,
- testy jednostkowe wersji szeregowych – Michał Niegrybowski.

2. ZAGADNIENIA PRZETWARZANIA RÓWNOLEGŁEGO

Algorytmy przetwarzania równoległego są dużo bardziej skomplikowane od rozwiązań stosowanych w programach, które wykonywane są szeregowo. Problem polega nie tylko na zrozumieniu skomplikowanego modelu obliczeń, ale również na zapewnieniu prawidłowej komunikacji, synchronizacji tworzonych wątków lub procesów oraz na dostępie do odpowiednich danych. Jednak mimo wielu problemów, model przetwarzania równoległego jest bardzo często używany podczas rozwiązań problemów naukowych lub komercyjnych. Przedstawione we wstępie zagadnienia są tematem tej części pracy.

2.1. *Ogólne problemy i ich rozwiązanie*

W tym podrozdziale opisano, na co powinna zwrócić uwagę osoba pisząca program z użyciem technologii równoległych. Pokazano również, jak zapobiegać lub rozwiązywać zaistniałe problemy, gdyż te sposoby różnią się w zależności od zastosowanej biblioteki.

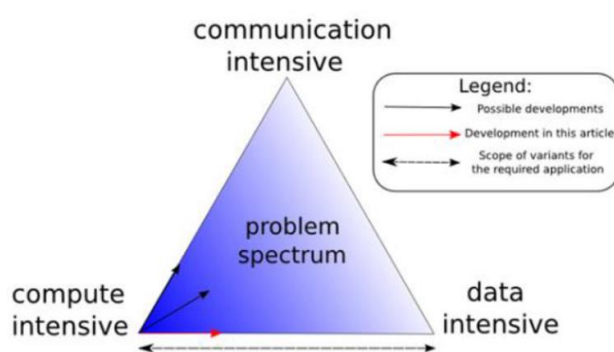
Jednym z głównych problemów towarzyszących programiście podczas pisania programów z użyciem technologii równoległych jest dostęp do prawidłowych danych. Musi on dbać o to, aby przekazywana do obliczeń liczba lub struktura miała oczekiwaną wartość, niezmienną przez niekontrolowany wątek lub proces. Problem, w którym kilka części programu jednocześnie, w sposób różny i niezrozumiały operuje na zmiennej lub grupie zmiennych, nazywamy wyścigiem. Aby zapobiegać takim błędom, najczęściej stosowane są sekcje krytyczne oraz bariery. Pierwsze stwierdzenie odnosi się do fragmentu kodu, który w jednym momencie może wykonać tylko jeden wątek. Wykorzystanie tego typu rozwiązania zapewnia programiście kontrolowane operacje na danych. Zaraz po wyjściu z fragmentu kodu nazywanego sekcją krytyczną, powinna wystąpić bariera. Jest to rozwiązanie wykorzystywane nie tylko w pracy z wątkami, ale również z procesami. Wymusza ona synchronizację pracujących części programu. Dzięki temu programista ma pewność, że w dalszej części kodu, żaden proces, ani wątek nie będzie korzystać z danych, które wciąż są modyfikowane w części kodu występującego przed barierą. W tym miejscu należy zauważyć, że odpowiednia synchronizacja oraz odpowiedni dostęp do danych są jednymi z najważniejszych zadań związanych z pracą w środowisku równoległym. Z tego względu wiele technologii po użyciu podstawowych metod lub dyrektyw związanych, np. z komunikacją, wymusza po jej zakończeniu wystąpienie bariery. Jest ona najczęściej integralną częścią wywoływanych instrukcji. Twórcy w ten sposób zapobiegają błędom związanym z nieuważnym dostępem do modyfikowanego miejsca w pamięci.

Innym utrudnieniem w pisaniu programów równoległych jest rozwiązanie komunikacji. Jednak w tym względzie wiele technologii wprowadza znaczne ułatwienia, pozostawiając programiście jedynie podstawowe decyzje. Wśród nich jest ustalenie, jaka liczba wątków lub procesów ma prowadzić obliczenia czy skąd lub dokąd mają być przesyłane dane. W testowanych w pracy technologiach OpenMP oraz OpenCL, domyślnie programista nie ma wpływu na kontrolę obciążania poszczególnych wątków. Zajmuje się tym scheduler, który decyduje o pracy, jaką wykonać mają poszczególne części programu. Trzeba zauważyć, że zła komunikacja nie musi powodować jedynie błędów w działaniu programu, lecz nieoptymalne

rozłożenie pracy pomiędzy wątki lub procesy może nawet znacznie wydłużyć czas jego wykonania w porównaniu z wersją szeregową. Znacznie większy wpływ na obciążenie procesów programista posiada w trzeciej z testowanych technologii, czyli MPI. Steruje tam każdym odrębnym procesem, decydując o tym, ile, skąd i jakie dane mają być odbierane, a także gdzie przekazywane. Być może problem komunikacji w tej technologii może wydawać się dużo trudniejszy do rozwiązania, lecz daje dużo większą swobodę doświadczonemu programiście. W tym miejscu należy również wspomnieć, że jednym z najpopularniejszych modeli komunikacji w przetwarzaniu równoległym pomiędzy częściami programu wykonującymi obliczenia jest struktura nazywana master-slave. Jak nazwa wskazuje, wyróżnia się jeden proces lub wątek główny, który rozdziela dane pomiędzy inne, zwane slave'ami. To master decyduje o obciążeniu węzłów i to on jest odpowiedzialny za dostarczenie danych do obliczeń oraz zebranie części składowych wyniku i przedstawienie ostatecznego rezultatu programu.

Dodatkowe utrudnienie w rozwiązaniu komunikacji może wprowadzić przełożenie jej z modelu synchronicznego na asynchroniczny. W wielu przypadkach, taki rozwój programu znacznie zwiększa jego wydajność, lecz jest jeszcze trudniejszy w zrozumieniu. Dzieje się tak za sprawą nakładania obliczeń i komunikacji. Węzły pełniące rolę jednostek obliczeniowych po otrzymaniu pakietu danych i rozpoczęciu na nich pracy, równocześnie otrzymują kolejną porcję do przetworzenia. Taki model zalecany jest dla programistów, którzy w pełni opanowali komunikację synchroniczną, gdyż model asynchroniczny poza wprowadzeniem kolejnych zależności i większej liczby działań w tym samym momencie, niczym się nie różni.

Na koniec rozdziału warto również wspomnieć o obszarach problemów algorytmicznych, które stwarzają podstawowe utrudnienia podczas pisania programów równoległych. Można wyróżnić trzy najważniejsze sektory, mianowicie dużą ilość obliczeń, komunikacji oraz danych. Poniższa ilustracja [Rościszewski P., 2014, s. 2] ukazuje trójkąt, w którym możemy umieścić znane algorytmy przetwarzania równoległego.



Rys. 2.1. Obszary problemów algorytmicznych w przetwarzaniu równoległym

Każda testowana podczas pisania pracy technologia przetwarzania równoległego wyróżnione problemy rozwiązuje w mniej lub bardziej podobny sposób. Dlatego celem tej pracy jest sprawdzenie, która z nich jest najbardziej odpowiednia dla wybranego sektora.

Powyższy rozdział pokazuje, że z programowaniem równoległym łączy się wiele problemów, które programista jest zmuszony rozwiązać, aby jego program działał zgodnie

z oczekiwaniami. Jednak mimo tych utrudnień, taki model obliczeń staje się coraz bardziej popularny i stosowany, co postarano się pokazać w kolejnym podrozdziale.

2.2. Zastosowanie

Łatwo zauważyć, że przetwarzanie równoległe jest obecnie coraz częściej wykorzystywane. Wynika to z faktu, iż problemy, które człowiek chce rozwiązywać, są coraz bardziej złożone. Duża ilość obliczeń i danych powoduje, że programy szeregowo działają zbyt wolno, aby czekać na ich rezultat. Czasem problemy są na tyle duże, że zachodzi potrzeba wykorzystania wielu rozproszonych geograficznie urządzeń. To wszystko możliwe jest dzięki wykorzystaniu technologii przetwarzania równoległego.

Należy podkreślić, że operacje równoległe mają swoje zasadnicze zastosowanie w nauce. Dzięki pracy na wielu połączonych ze sobą klastrach obliczeniowych są w stanie w zadowalającym czasie zobrazować reakcje chemiczne, modele zniszczeń, czy też symulować rozchodzące się pożary. Przez fakt, iż w takich przypadkach wiele danych zależy od siebie, płynne zobrazowanie rezultatu obliczeń przez program sekwencyjny nie dałoby oczekiwanego efektu. Spowodowane byłoby to potrzebą przełączania obliczeń pomiędzy częściami obrazu, które mogłyby trwać zbyt długo. Ten problem jest zminimalizowany w przetwarzaniu równoległym, dzięki zastosowaniu takiego modelu symulacje wydają się być bardziej rzeczywiste.

Poza zastosowaniem w nauce, przetwarzanie równoległe wykorzystuje się w różnych komercyjnych serwisach internetowych. Polega ona na równoczesnej obsłudze kilku klientów poprzez np. stworzenie wielu wątków i przydzielenie każdego z nich do konkretnych użytkowników lub ich grup. Dzięki temu goście serwisu są szybciej obsługiwani, a co za tym idzie wzrasta ich zadowolenie i rośnie popularność serwisu. Właśnie te cele powodują, że coraz częściej firmy decydują się na wprowadzenie takiego modelu.

Obliczenia prowadzone równoległe są również bardzo często wykorzystywane w grafice komputerowej, np. podczas przetwarzania obrazów. To za sprawą budowy kart graficznych, których model przypomina ten zastosowany w koprocessorze Intel Xeon Phi. Duża liczba rdzeni o niskim taktowaniu umożliwia przeprowadzanie ogromnej ilości prostych obliczeń w bardzo krótkim czasie. Z tego względu obliczenia równoległe są wykorzystywane podczas transformacji obrazów, gdyż każdy piksel obrazu należy odpowiednio zmienić. Obliczenia sekwencyjne trwałyby w takim przypadku bardzo długo, dlatego obecnie nie są już stosowane w większości tego typu operacjach.

Przetwarzanie równoległe w pracy z dźwiękiem jest równie popularne, jak w grafice komputerowej. Opiera się ona na tej samej zasadzie, potrzeby zastosowania dużej ilości małych obliczeń na całej próbce dźwięku, aby ją oczyścić z tzw. szumów. Dzięki temu pojawia się możliwość, np. odsłuchania głosu z nagrania, który przed przeprowadzonymi zmianami był niezrozumiały dla słuchacza.

W tym rozdziale nie można też pominąć zastosowania nieco innego podejścia do przetwarzania równoległego, mianowicie o volunteer computing. Obliczenia w tym modelu prowadzone są na wielu rozproszonych geograficznie stacjach roboczych, a chęć dołączenia jako

taki węzeł jest zupełnie dobrowolna, stąd pierwszy człon nazwy. Wszystko polega na wykorzystaniu części zasobów zgłoszonej stacji roboczej na obliczenia, dzięki połączeniu internetowemu, poszczególne rezultaty są wysyłane do głównego węzła, który analizując zebrane dane, tworzy wynik końcowy. Należy zauważyć, że osoba udostępniająca swoje zasoby do pracy, najczęściej nie odnosi z tego żadnych korzyści, poza satysfakcją z pomocy w rozwiązaniu jakiegoś znanego problemu. Charakter obliczeń jest zazwyczaj podobny, czyli jest to duża ilość małych obliczeń na porcji danych, przykładowymi problemami rozwiązywanymi z zastosowaniem volunteer computing jest szukanie największych liczb bliźniaczych czy rozwiązywanie szyfrów. Należy zauważyć, że w zastosowaniu takiego modelu do obliczeń nie mamy pewności poprawności wyników oraz nie mamy zapewnionej stałej liczby węzłów, które można wykorzystać.

Jak łatwo zauważyć przetwarzanie równoległe w obecnych czasach znajduje szerokie zastosowanie zarówno w nauce, jak i w pracy z komercyjnymi projektami. Daje to prawo sądzić, iż z czasem popularność takiego modelu prowadzenia obliczeń będzie nadal rosnąć.

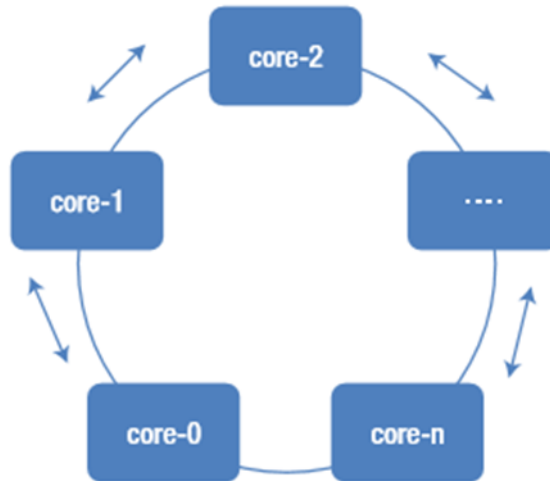
3. XEON PHI

W 2010 roku okazało się, że prosta architektura typu MIC z rdzeniami o niskiej częstotliwości oraz z odpowiednim wsparciem oprogramowania jest możliwa osiągać lepszą wydajność zarówno obliczeniową, jak i związaną z zużyciem prądu, od aktualnie używanych typów architektury. To rozwiązanie wymagało nowej mikroarchitektury, która opierała się na użyciu procesorów typu x86. Została ona opisana w tym rozdziale na podstawie książki Intel Xeon Phi Coprocessor Architecture and Tools [1].

3.1. Architektura

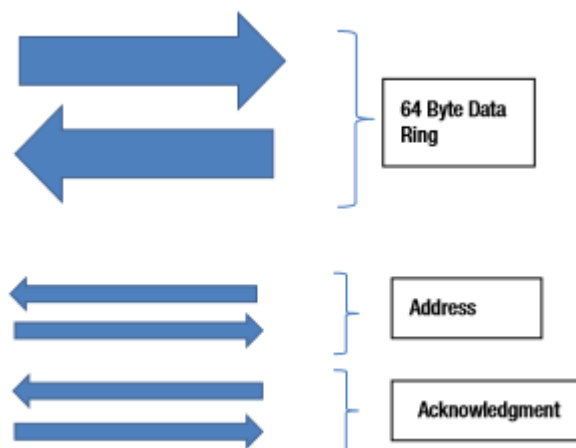
W aspekcie architektury Xeon Phi warto omówić kilka najważniejszych jej cech, z czego pierwszą są potoki. Rdzenie koprocesora Xeon Phi posiadają po dwa niezależne potoki, tzw. U oraz V, które mogą wykonywać jednocześnie różne instrukcje. Należy jednak zaznaczyć, iż potok V jest w stanie wykonać jedynie pewną część wszystkich instrukcji, a w dodatku jest ograniczony przez zasadę parowania instrukcji. Z kolei jednostka wektorowa posiada pięć pipeline'ów, a każdy z nich przetwarza inne instrukcje. DP, czyli double precision pipeline, odpowiada za wykonanie operacji arytmetycznych na 64 bitowych liczbach zmiennoprzecinkowych, porównania liczb zmiennoprzecinkowych oraz konwersji z float64 na float32. SP czyli Single precision pipeline. Odpowiada za wykonanie większości instrukcji. Mask Pipeline zajmuje się wykonaniem instrukcji maskujących z jednocyklowym opóźnieniem. Store pipeline odpowiada za operacje zapisu wektorowego oraz ostatni Scatter/Gather pipeline za czytanie/zapisywanie z rozszanych lokacji w pamięci. Trzeba pamiętać, że w przypadku pisania programów na Intel Xeon Phi, warto jest posługiwać się tymi samymi typami danych, ponieważ w przypadku użycia różnych, mogą one trafić do odmiennych potoków, a komunikacja pomiędzy nimi jest bardzo kosztowna pod względem czasu działania programu.

Kolejnym komponentem wartym uwagi, jest realizacja sieci wewnętrznych połączeń na karcie. Została ona wykonana w topologii dwukierunkowej pętli, tzw. ringu. Polega ona na tym, że każdy element jest połączony z dwoma sąsiednimi dwukierunkowymi połączeniami. Niska złożoność tej topologii jest dobra dla małej liczby rdzeni. W przypadku komunikacji, średnia liczba skoków wynosi jedną czwartą liczby rdzeni.



Rys. 3.1. Wizualizacja dwukierunkowej topologii

Karta Xeon Phi wykorzystuje ten komponent do wymiany danych oraz instrukcji pomiędzy elementami koprocatora. Są one połączone do szyny za pomocą tak zwanych przystanków (ring stops). Owe przystanki zarządzają całym ruchem do i z ringu. Aczkolwiek, mówienie o ringu może być mylące z uwagi na to, że w architekturze Xeona są ich trzy niezależne pary (każdy z nich w przeciwnym kierunku). Szyna danych pierwszej pętli ma szerokość 64 bajtów i odpowiada za dane. Następny ring odpowiada za adresy i komendy odczytu i zapisu, natomiast ostatni za kontrolę przepływu informacji oraz za zgodność wiadomości. Podglądowy rys. 3.2. przedstawia ringi w Xeon Phi.



Rys. 3.2. Przedstawienie Ringów na koprocesorze Xeon Phi

TD, czyli Tag Directory pozwala na zachowanie spójności danych pomiędzy rdzeniami na ringu. Zajmuje się filtrowaniem i przekazywaniem żądań do konkretnych agentów. Odpowiada również za wysyłanie „pytających” żądań do losowych rdzeni w imieniu rdzeni żądających i zwraca linie z pamięci L2. Fizycznie Tag Directory jest przyklejony do każdego z rdzeni. TD implementuje protokół GOLS.

Koprocesor Xeon Phi używa zmodyfikowanego protokołu MESI dla zapewnienia spójności pamięci podręcznej pomiędzy wątkami, które używają Tag Directory. W poprzednim zdaniu padło wiele skrótów, więc warto je wyjaśnić. MESI odwołuje się do stanów pamięci w L2. Wyjaśniając każdą literę skrótu, M od ang. modified oznacza, że linia pamięci podręcznej jest modyfikowana relatywnie w stosunku do pamięci. Tylko jeden rdzeń może mieć dostęp do danej linii w cache'u w jednej chwili. E od ang. exclusive linia pamięci podręcznej jest spójna z pamięcią. Dostęp do niej jest analogiczny jak w stanie M. S od ang. shared oznacza że linia cache'u jest udostępniana i spójna pomiędzy rdzeniami, ale może być niespójna z pamięcią. Wiele rdzeni może mieć pamięć w stanie S. I od ang. invalid oznacza, że linia nie jest ani w pamięci L1, ani L2 na żadnym z rdzeni.

```

graph TD
    Invalid((Invalid))
    Modified((Modified))
    Exclusive((Exclusive))
    Shared((Shared))

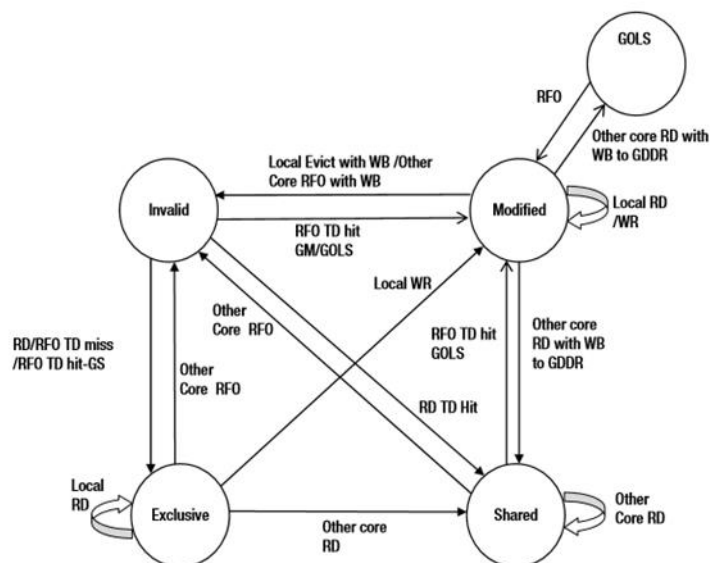
    Invalid -- "Local Evict with WB / Other Core RFO with WB" --> Modified
    Invalid -- "RFO" --> Modified
    Invalid -- "Local RD No Hit" --> Exclusive
    Invalid -- "Other Core RFO" --> Exclusive
    Invalid -- "Other Core RFO" --> Shared
    Invalid -- "RD Hit" --> Shared
    Invalid -- "Other core RD" --> Shared

    Modified -- "Local RD /WR" --> Modified
    Modified -- "Local WR" --> Invalid
    Modified -- "Local WR" --> Exclusive
    Modified -- "Local WR" --> Shared
    Modified -- "Other core RD with WB to GDDR" --> Exclusive
    Modified -- "Other core RD with WB to GDDR" --> Shared

    Exclusive -- "Local RD" --> Exclusive
    Exclusive -- "Local WR" --> Invalid
    Exclusive -- "Local WR" --> Modified
    Exclusive -- "Local WR" --> Shared
    Exclusive -- "Other core RD" --> Invalid
    Exclusive -- "Other core RD" --> Modified
    Exclusive -- "Other core RD" --> Shared

    Shared -- "Local RD" --> Shared
    Shared -- "Local WR" --> Invalid
    Shared -- "Local WR" --> Modified
    Shared -- "Local WR" --> Exclusive
    Shared -- "Other core RD" --> Invalid
    Shared -- "Other core RD" --> Modified
    Shared -- "Other core RD" --> Exclusive
  
```

W celu usunięcia potencjalnych problemów z wydajnością z uwagi na brak właściciela (stan O) w protokole MOESI (rozwińcie protokołu MESI), koprocesor Xeon Phi implementuje TD, aby zarządzać globalnym stanem zmodyfikowanych linii, tak by mogły być one udostępniane pomiędzy rdzeniami bez niepotrzebnego powtórnego zapisu do GDDR. Dzięki temu rys.3.3 zmienia się w rysunek rys. 3.4.



Rys. 3.4. Stany linii pamięci w pamięci podręcznej poziomu L2 z udziałem Tag Directory

Mając na uwadze, że w koprocesorze znajduje się 60 rdzeni, a każdy jest taktowany szybkością około 1.1 GHz, można łatwo policzyć teoretyczną wydajność dla SP, co pokazują podane zależności (3.1, 3.2).

$$\frac{GFLOP}{sec} = 16 * (SP \text{ SIMD Linia}) * 2 (FMA) * 1.1 (GHz) * 60 (\text{procesorów}) = 2112 \text{ dla SP} \quad (3.1)$$

$$\frac{GFLOP}{sec} = 8 * (DP \text{ SIMD Linia}) * 2 (FMA) * 1.1 (GHz) * 60 (\text{procesorów}) = 1056 \text{ dla DP} \quad (3.2)$$

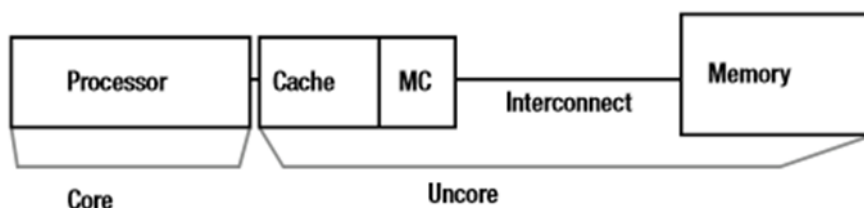
Maksymalną przepustowość pamięci można też policzyć w łatwy sposób z uwagi na to, że osiem kontrolerów pamięci z kanałami GDDR5 działa na 5.5 GT/s. Dzięki tej wiedzy otrzymano następującą zależność (3.3) na zagregowaną przepustowość pamięci

$$p = 8 (\text{kontrolery pamięci}) * 2 (\text{kanały}) * 5.5 \frac{GT}{s} * 4 \frac{\text{bajty}}{\text{transfer}} = 352 \frac{GB}{s} \quad (3.3)$$

gdzie:

p – wartość zagregowanej przepustowości pamięci.

Warto również napisać o Core oraz Uncore, czyli podziale rdzenia w Intel Xeon Phi. Core zawierają silniki przeznaczone do obliczeń, one natomiast zawierają jednostki wektorowe w wielu nowoczesnych procesorach. Uncore natomiast, zawierają pamięć podręczną, pamięć ram oraz komponenty peryferyjne. Kiedyś Core były ważniejsze od Uncore, aczkolwiek w czasach teraźniejszych coraz większe znaczenie ma pierwszy z wymienionych.

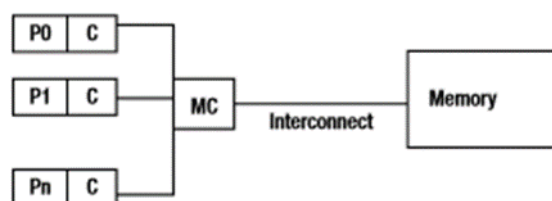


Rys. 3.5. Core i Uncore

VPU czyli Xeon Phi Vector Microarchitecture jest rozszerzeniem rdzenia P54C i komunikuje się z rdzeniami w celu wykonania VPU ISA. VPU otrzymuje polecenia od ALU, natomiast dane, od pamięci podręcznej poziomu L1 poprzez dedykowaną 512-bitową szynę. Ma on swoją własną jednostkę logiczną i komunikuje się z rdzeniami w chwili, gdy jest to potrzebne.

Dzięki temu, że VPU jest w pełni potokowy wykonuje większość instrukcji z opóźnieniem czterech cykli zegarowych i wydajnością pojedynczego, jest to spowodowane między innymi tym, że VPU może wykonywać jeden odczyt i zapis jednocześnie. Dzięki zastosowaniu podwójnego źródła instrukcji można było zwiększyć o około 20 procent klasyczne operacje binarne SIMD. Każdy VPU posiada 128, 512 bitowych rejestrów wektorowych, które odgrywają rolę wejść i są rozłokowane po wszystkich wątkach, gwarantując 32 wejścia per wątek.

Warto również napisać o architekturze typu Multicore oraz Manycore. Ewolucja przetwarzania równoległego wymagała posiadania danych na każdym z rdzeni, który na nich operował, więc dane zawsze musiały być kopiowane do i z każdego, który go potrzebował. Tak powstały procesory typu Multicore. Mimo, że takie procesory były niżej taktowane od takich, które były jednordzeniowe to zapewniały większą wydajność z uwagi na lepszą optymalizację równoległości, aczkolwiek pojawił się problem z kopiowaniem pomiędzy rdzeniami. Było to związane z limitami związanymi z zasilaniem. Rozwiązaniem tego problemu są architektury typu Manycore opierają się one na tym, że do dyspozycji mamy „duże” i „małe” rdzenie. Warto tutaj także rozróżnić dwa typy architektury Manycore, czyli homogeniczna architektura Manycore, gdzie wszystkie rdzenie są podobne, oraz heterogeniczna architektura Manycore w której rdzenie nie muszą być identyczne. Taka ewolucja architektury zapewniła poprawę wydajności aplikacji bez zwiększania częstotliwości zegara procesora. Są też jednak pewne minusy takiego rozwiązania, ciężar zrównoleglania aplikacji przeszedł z osób odpowiedzialnych za hardware na zwykłych programistów, którzy są odpowiedzialni za oprogramowanie, aczkolwiek należy zaznaczyć, że koncepcja Manycore nie jest szeroko używana.



Rys. 3.6. Ewolucja architektury w stronę Manycore. Rozwinięcie skrótów C – pamięć podręczna, MC – kontroler pamięci, P_x – rdzeń procesora

3.2. Dostęp do maszyny

W kontekście pisanej pracy warto opisać, gdzie znajdował się koprocesor Intel Xeon Phi 5120P, na którym realizowano projekt i w jaki sposób uzyskiwano dostęp do tego miejsca.

Procesor Xeon, jak również współpracujący z nim koprocesor zostały zainstalowane na jednej z katedralnych maszyn, a konkretnie na apl12. Dostęp do niego można było uzyskać poprzez skorzystanie z kont studentXX@apl12, łącząc się z wewnątrz wydziałowej sieci internetowej. Możliwe również było użycie kont katedralnych zakładanych studentom obierającym

KASK jako profil dyplomowania. Z tego drugiego sposobu najczęściej korzystano podczas realizacji projektu. Po połączeniu się z siecią katedralną była możliwość skorzystania z dysku będącego wspólną przestrzenią dla wielu urządzeń, w tym dla wykorzystywanego apl12, tam umieszczano wykorzystywane skrypty i zaimplementowane programy. Natomiast po zalogowaniu na maszynę apl12 uruchamiano je, aby zrealizować cel pracy inżynierskiej.

Na koniec tego krótkiego podrozdziału warto wspomnieć, że dostęp do maszyny apl12, a także do samego koprocatora Xeon Phi został opisany w instrukcji laboratoryjnej do przedmiotu Systemy Obliczeniowe Wysokiej Wydajności autorstwa dr hab. Pawła Czarnula, która wyjaśnia nie tylko, jak się połączyć z maszyną, ale również jak uruchomić program na akceleratorze w trybie natywnym.

4. OPIS PORÓWNYWANYCH TECHNOLOGII

W tym rozdziale pracy opisano najważniejsze cechy testowanych technologii, czyli MPI, OpenMP i OpenCL. Zwrócono uwagę na charakterystyczne zachowania każdej z nich w zależności od stosowanych funkcji.

4.1. MPI

Jedną z porównywanych w tej pracy inżynierskiej technologii jest MPI. Jest to skrót od angielskiej nazwy protokołu komunikacyjnego, Message Passing Interface. Protokół ten jest standardem przesyłania wiadomości od punktu do punktu lub, korzystając z procedur komunikacji grupowej, pomiędzy procesami dwóch lub większej liczby stacji roboczych. Ta cecha wyróżnia MPI spośród innych testowanych technologii. Poniższa charakterystyka została napisana w oparciu o dokumentację MPI [13] oraz informacje ze strony MPICH [19].

Warto zauważyć, że istnieje wiele implementacji tego standardu, np. na koprocessorach Intel Xeon Phi wykorzystywana jest Intel MPI Library, biblioteka, która umożliwia kompilację kodu z użytymi funkcjami Message Passing Interface i uruchomienie programu na wspomnianym akceleratorze. Klasyczne implementacje MPI, takie jak MPICH są przeznaczone dla języków C/C++, jednak w chwili obecnej istnieje sporo rozwiązań korzystających z tego standardu dla innych języków, takich jak np. Python lub Java. Są to między innymi pyMPI, mpi4py, mpi Java API. Świadczy to o popularności opisywanego protokołu komunikacyjnego, dzięki czemu jest on nadal rozwijany w różnych kierunkach.

Najważniejszą cechą opisywanej technologii jest fakt, iż tworzy ona procesy. Aby zainicjalizować środowisko MPI w programie, każdy powołany do życia proces musi wywołać funkcję z grupy MPI_Init. Po tej instrukcji programista ma możliwość korzystania w obrębie wykonywanego programu z powołanych do życia procesów dopóki nie zakończy pracy ze środowiskiem MPI metodą MPI_Finalize wywołaną przez każdy działający proces. Dzięki operowaniu na takich jednostkach roboczych standard MPI posiada cechę, która odróżnia tę technologię od innych, testowanych w tej pracy. Mianowicie procesy, z racji, iż z definicji nie współdzielą pamięci, mogą pracować w środowisku rozproszonym. To właśnie dzięki standardowi Message Passing Interface, programista ma możliwość napisania programu, który wykorzysta moc obliczeniową połączonych ze sobą jednostek komputerowych, tzw. węzłów w obrębie lub pomiędzy klastrami. Przesyłanie danych pomiędzy poszczególnymi procesami, tzw. punkt-punkt, jest możliwe dzięki wykorzystaniu rodzin funkcji MPI_Send oraz MPI_Recv, dzięki nim programista definiuje z jakiego lub do jakiego procesu dane mają zostać przekazane, ile ich jest i jakiego są typu. Należy również nadać identyfikator, będący liczbą całkowitą, wysłanemu komunikatowi, tak aby odbiorca wiedział w jaki sposób odebrać dane. Dzięki takiemu rozwiązaniu użytkownik technologii ma możliwość decyzji, które obliczenia mają być prowadzone przez poszczególne procesy, czyli inaczej mówiąc, decyduje o obciążeniu poszczególnych węzłów modelu wykonawczego. W tym miejscu warto napisać, że podstawowe metody służące do komunikacji mają charakter najczęściej blokujący, choć semantyka MPI_Send nie jest do końca sprecyzowana, co zostało opisane w następnym akapicie.

Korzystając z polecenia `MPI_Send`, programista nie ma gwarancji, że dane zostaną odebrane przez proces będący odbiorcą. Standard MPI nie precyzuje zachowania tej funkcji, cytując dokumentację MPICH [19] „This routine may block until the message is received by the destination process.”. Oznacza to, że wykorzystanie tej funkcji może, lecz nie musi być blokujące. Będzie ona taka do czasu, gdy z punktu widzenia wysyłanego komunikatu będzie to bezpieczne zachowanie. Wiadomo jedynie, że komunikat został wysłany, oraz, że po zakończeniu `MPI_Send`, obszar pamięci zajmowany przez komunikat będzie mógł być bezpiecznie użyty do dalszych działań. Jednak nie ma informacji o tym, czy wysłana wiadomość została poprawnie odebrana i przetwarzana przez proces będący jej odbiorcą. Mimo to, MPI dostarcza szereg innych funkcji, których zachowanie jest lepiej sprecyzowane. Dzieli się one na blokujące i nieblokujące. Funkcja blokująca oznacza, że dopóki wysyłanie danych lub ich odbieranie nie zakończy się, proces nie rozpocznie dalszej pracy z kodem. Przykładami takich metod są:

- `MPI_Ssend` – jest to blokujący sposób przesyłania danych, dopóki odbiorca nie zacznie przyjmować danych funkcją `MPI_Srecv` nadawca będzie czekać z wykonaniem dalszej części kodu,
- `MPI_Bsend` – buforowany, blokujący sposób wysyłania danych. Funkcja skończy się, gdy wysyłany komunikat zostanie skopiowany do lokalnego bufora. Zakończenie metody nie musi oznaczać, że dane faktycznie zaczęły być wysyłane,
- `MPI_Rsend` – również blokujący sposób wysyłania danych. Użycie tej funkcji wymusza wywołanie przez odbiorcę funkcji `MPI_Rrecv`, która sygnalizuje gotowość na odebranie komunikatu.

Jeżeli programiście zależy na większej wydajności pisanego programu, warto zastosować funkcje nieblokujące, które zapewniają nakładanie się obliczeń prowadzonych przez proces wraz z odbieraniem lub wysyłaniem przez niego danych. Na podstawie komunikatu pełniącego funkcję uchwytu związanego z daną komunikacją, programista może dowiedzieć się czy instrukcja zakończyła się. Dzięki niemu jest możliwość zaczekania na wykonanie akcji przed rozpoczęciem pracy na odbieranych danych. Funkcje nieblokujące działają analogicznie do wymienionych powyżej, z tym, że umożliwiają asynchroniczną komunikację. Są to między innymi `MPI_Isend`, `MPI_Irecv`, `MPI_Issend`, `MPI_Ibsend`, `MPI_Irsend`.

Pisząc o koncepcji punkt-punkt, warto również wspomnieć krótko o funkcjach zbiorowych. Nie są one czymś niezwykłym, jednak ułatwiają pracę programiście, gdyż opierają się one na komunikacji pomiędzy wszystkimi procesami w grupie. Dzięki temu korzystając np. z funkcji `MPI_Reduce` można zebrać wyniki częściowe ze wszystkich węzłów, wykonać na nich operację i przechować wynik końcowy w głównym węźle.

Każda z technologii przetwarzania charakteryzuje się jakimś modelem obliczeń. W programach z użyciem MPI najczęściej stosuje się tradycyjną hierarchię master-slave. Oznacza ona, że w programie wyróżniony jest jeden z procesów pełniący rolę koordynatora, którego głównym zadaniem jest rozdzielanie danych na poszczególne procesy, na których odebrane komunikaty są przetwarzane. Wątek zerowy, gdyż ten zawsze powinien pełnić tę funkcję, zbiera

również wyniki oraz scala je w wersję finalną. Jest to bardzo prosty w zrozumieniu model, który sprawdza się w rozwiązaniu większości problemów.

Technologia MPI, ma swoje wady związane z niejasną semantyką niektórych funkcji lub też w wielu przypadkach może sprawiać problemy jej zastosowanie, szczególnie modelu asynchronicznego. Jednak daje dużą swobodę programiście, który może decydować o obciążeniu węzłów, a przede wszystkim może wykorzystać zasoby kilku komputerów. To właśnie te cechy technologii Message Passing Interface wpływają na jej wysoką popularność.

4.2. OpenMP

OpenMP, czyli Open Multi-Processing, podobnie jak OpenCL i MPI, jest to wieloplatformowe API przeznaczone do zrównoleglania aplikacji. W odróżnieniu od dwóch wyżej wymienionych frameworków wyróżnia go fakt, iż opiera się na dyrektywach preprocesora, pragmach.

Architektura OpenMP opiera się na przenośności i skalowalności modelu, który pozwala programiście na łatwe i elastyczne wytwarzanie oprogramowania niezależnie od typu urządzenia, na które jest wytwarzane. Implementacja OpenMP, opiera się na modelu fork-join, w którym wątek inicjalny jest powielany, tworząc przetwarzającą grupę wątków. Jednak warto już tutaj zauważyć, że programista nie ma bezpośredniego wpływu na obciążenie poszczególnych części programu zajmujących się obliczeniami, którymi w przypadku opisywanej technologii są wątki. W obrębie grupy są one wykonywane równolegle, jednak należy zauważyć, że po zakończeniu obszaru zrównoleglonego jedną z konstrukcji OpenMP, prace kontynuuje jedynie wątek inicjalny. Istnieje możliwość rozróżnienia ich pomiędzy sobą, gdyż każdy ma przypisany identyfikator.

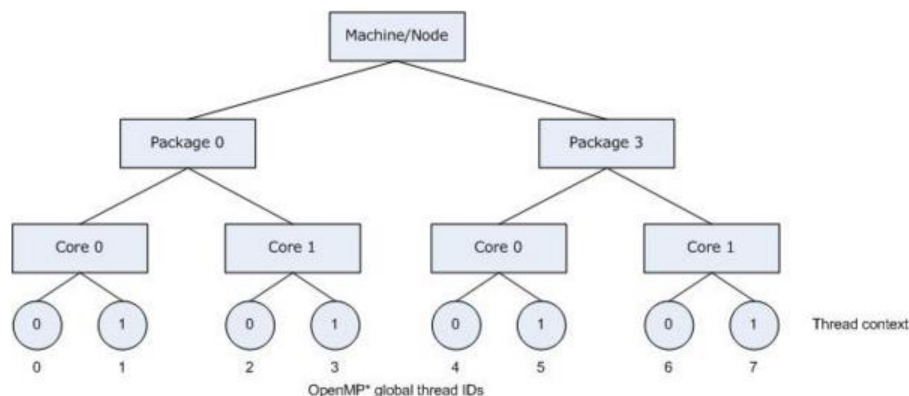
Domyślnie każdy wątek wykonuje sekcję równoległą niezależnie, to znaczy, że możemy osiągnąć zarówno zrównoleglenie zadań, jak i danych. Wartym wspomnienia jest także konstrukcja work-sharing, dzięki której jeden wątek dostanie tylko część danych, która jest mu potrzebna, a nie wszystkie. Wątki są alokowane na procesorze w zależności od jego aktualnego obciążenia w danym momencie.

Aspektem, który warto omówić jest Thread Affinity Interface. Jest to biblioteka umożliwiająca przypisanie wątków OpenMP do fizycznych jednostek. Interfejs jest kontrolowany dzięki zmiennej środowiskowej o nazwie KMP_AFFINITY. W zależności od maszyny, systemu czy też aplikacji, interfejs ten może mieć drastyczny efekt na szybkość działania aplikacji. Dzięki owej zmiennej środowiskowej możemy ustalić w jaki sposób będą rozdzielane wątki na jednostki fizyczne. Dzięki współpracującej z nią zmiennej KMP_PLACE_THREADS programista ma również możliwość ustalenia, ile rdzeni zostanie użytych i ile wątków będzie na każdym z nich stworzonych. Pozwala to na lepsze dopasowanie dostępnych zasobów do tworzonej aplikacji z uwagi na to, że domyślnie Thread Affinity ma ustawienia dopasowane pod ogólne programy.

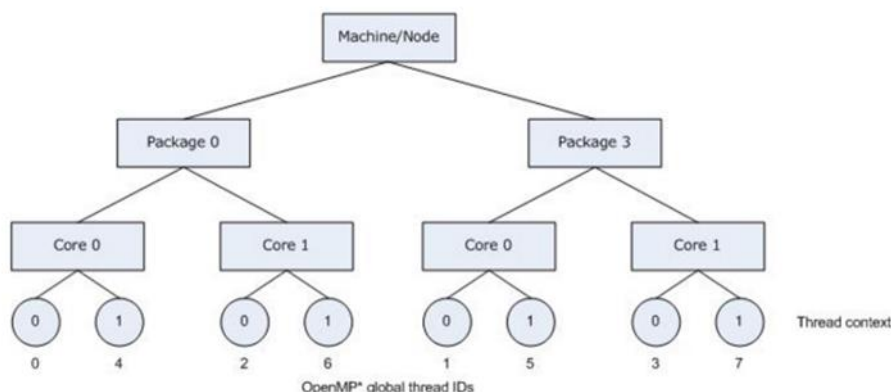
Nie opisano tutaj wszystkich możliwości, które udostępnia nam zmienna środowiskowa KMP_AFFINITY, ale warto omówić trzy najważniejsze opcje podziału wątków pomiędzy procesy. Pierwszą z nich jest none, czyli ustawienie domyślne, które nie przypisuje żadnego wątku

OpenMP do konkretnej puli wątków, jednak jeżeli system wspiera Thread Affinity, kompilator nadal będzie próbował uzyskać informację na temat topologii maszyny.

Następnymi typami które warto omówić w kontekście KMP_AFFINITY jest compact i scatter. Compact zakłada przypisanie wątku OpenMP do najbliższego wolnego z puli dostępnych wątków na procesorze, najbliższego w kontekście do poprzedniego utworzonego wątku OpenMP, rys. 4.1. doskonale obrazuje opisywany tryb. Scatter natomiast, jest niejako odwrotnością compact, polega na przypisaniu kolejnych wątków OpenMP równomiernie do każdego rdzenia. Rys. 4.2. przedstawia ten tryb.

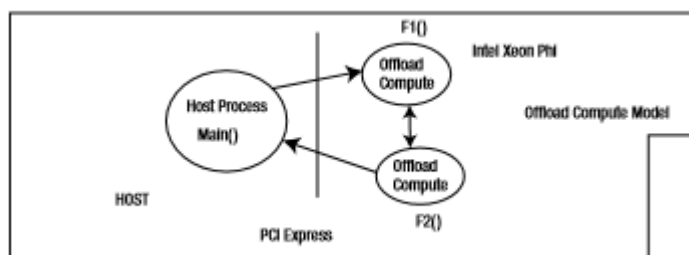


Rys. 4.1. Podział wątków dzięki Thread Affinity przy użyciu typu compact



Rys. 4.2. Podział wątków dzięki Thread Affinity przy użyciu typu scatter

Omawiając technologię OpenMP, wartym wspomnienia jest również tryb wykonania offload. Jest on znany również jako heterogeniczny model programowania. W kontekście uruchamiania programów z użyciem Intel Xeon Phi, ten tryb pozwala na przesłanie fragmentu, bądź też całości danych wraz z instrukcjami obliczeń z procesu głównego programu uruchomionego na procesorze Intel Xeon do procesów lub wątków uruchomionych na koprocesorze Xeon Phi. Kiedy aplikacja jest uruchamiana można zdecydować, co ma zostać wykonane przez utworzone na urządzeniu wątki lub procesy. Obliczenia oraz transfer danych przy wykorzystaniu offload można również prowadzić w trybie asynchronicznym, dzięki użyciu parametru signal. Jest to informacja na temat zakończonej pracy sekcji poprzedzonej dyrektywą preprocesora z rodziny offload. Rys. 4.3. obrazuje opisywany powyżej tryb.



Rys 4.3. Praca z wykorzystaniem trybu offload

W OpenMP kod, który ma zostać wykonany równolegle musi zostać poprzedzony odpowiednią dyrektywą rozpoczynającą się od:

```
#pragma omp parallel
```

pozwała ona na stworzenie grupy wątków, których zadaniem będzie praca z kodem w oznaczonym bloku. To od programisty zależy jak użyje powstałych wątków.

W tej części warto napisać o zrównolegleniu pętli w technologii OpenMP. Jest to możliwe poprzez użycie dyrektywy:

```
#pragma omp parallel for
```

tworzy ona wspomnianą wcześniej grupę wątków, ale dodatkowo zrównolegla następującą po niej pętlę, przyporządkowując poszczególne iteracje różnym wątkom. Kolejnym ważnym aspektem są różne parametry powyższej instrukcji, m.in. są to *static*, *dynamic*, *guided*. Określają one w jaki sposób iteracje mają być podzielone pomiędzy wątki, równomiernie przed rozpoczęciem operacji, czy też dynamicznie już w ich trakcie.

Opisywana technologia posiada bardzo wiele dyrektyw, jednak opisanie ich wszystkich nie jest celem niniejszej pracy, warto jednak jeszcze wspomnieć, że OpenMP może być używane w programach napisanych w następujących językach, C, C++ oraz Fortran.

OpenMP jest prawdopodobnie najprostszą w użyciu z opisywanych technologii, daje wiele możliwości programiście, którego praca jest ułatwiona, np. poprzez brak bezpośredniego wpływu na rozdysponowanie danych pomiędzy wątki i obciążenie każdego z nich. Może to być zarówno wielką zaletą jak i wadą, czego sprawdzenie jest jednym z celów realizowanego projektu inżynierskiego. Mimo to, opisane cechy technologii OpenMP sprawiają, że jest to bez wątpienia jedna z najlepiej rozwijanych technologii na obecnym rynku.

4.3. OpenCL

OpenCL czyli Open Computing Language jest to framework, który wspiera tworzenie aplikacji działających w rozproszonych środowiskach. Nie określa konkretnego środowiska uruchomieniowego. W chwili obecnej urządzenia, na których może być uruchamiany kod w OpenCL, to CPU, GPU i akceleratory, czyli na przykład Intel Xeon Phi. Sięgając trochę w głąb historii i tego jakie były zamiary przy tworzeniu technologii, takiej jak OpenCL, można się dowiedzieć, że chodziło o elastyczność w stosunku do różnych urządzeń obliczeniowych, a więc żeby kod napisany w OpenCL bez żadnych zmian mógł być uruchomiony zarówno na karcie

graficznej, procesorze, akceleratorze czy też na wszystkich równocześnie, w porównaniu do technologii CUDA, która może być uruchamiana tylko na urządzeniach z rodziny Nvidia.

Reasumując, firma Khronos, która jest odpowiedzialna za OpenCL, aby osiągnąć podaną wcześniej elastyczność oparła ten standard na czterech modelach opisujących podstawowe obszary. Są to model platformy, pamięci, wykonawczy i programowania. Warty jest ich omówienie, a także tego co konkretnie one oznaczają dla technologii OpenCL i jak na nią wpływają.

Model platformy dla OpenCL oznacza, że będzie on technologią heterogeniczną, a więc taką, która nie jest zależna od platformy sprzętowo-programowej. W ogólności model platformy w tej technologii polega na tym, że urządzenie, które sprawuje rolę hosta, nie bierze udziału w obliczeniach tylko rozdziela pracę na jednostki obliczeniowe. Wynikałoby z tego, że jednostka hosta nie musi być nadmiernie wydajna, aczkolwiek trzeba mieć na uwadze, że host musi zadbać o przesyłanie danych oraz procedur do i z jednostek obliczeniowych, jak i też odpowiada za rozpoczęcie obliczeń na nich.

Podział pomiędzy hosta, a jednostki obliczeniowe ma też swoje odzwierciedlenie w kodzie z uwagi na to, że tworzą się dwie oddzielne aplikacje. Jedna zawierająca kod hosta, który wyszukuje urządzenia, na których mogą być wykonywane obliczenia i rozdysponowuje je do nich, oraz aplikacje, które są wykonywane na jednostkach obliczeniowych, czyli tak zwane kernele.

Model pamięci w technologii OpenCL charakteryzuje się tym, że są wyróżnione dokładnie cztery jej rodzaje. Mianowicie są to pamięć globalna, lokalna, prywatna oraz stała. Każda z nich różni się pewnymi aspektami, takimi jak, kto ma do niej dostęp, jaką może mieć maksymalną wielkość oraz jaki jest jej czas dostępu. Po pogrupowaniu rodzajów pamięci ze względu na powyższe cechy otrzymujemy tabelę 1.1.

Tabela 1.1. Pogrupowanie pamięci w OpenCL ze względu na ich cechy pogrupowaniu

Aspekt	Poziom	Pamięć stała	Pamięć lokalna	Pamięć prywatna	Pamięć globalna
Dostęp do pamięci	host	Dynamiczna alokacja, Zapis, Odczyt	Alokacja dynamiczna	brak	Alokacja dynamiczna, Zapis, odczyt
	kernel	Alokacja statyczna, odczyt	Alokacja statyczna, Zapis, odczyt	Alokacja statyczna, Zapis, odczyt	Zapis, odczyt
Maksymalna wielkość (dane dla modelu Xeon Phi 5120)	-	256KB	64KB dla każdej grupy	64KB	8GB
Szybkość (gdzie 1 oznacza najszybszą, a 4 najwolniejszą)	-	3	2	1	4

Idąc za poradami Intel'a, podczas projektowania aplikacji w OpenCL na urządzenia Xeon Phi w aspektach modelu pamięci powinno się zwrócić szczególną uwagę na to, aby unikać używania pamięci lokalnej. Programista musi starać się, aby dane były wyrównane do 32 bitów

z uwagi na to, że pozwala to na odczyt w postaci ciągłych bloków, co wydatnie przyspiesza działanie aplikacji.

Model wykonawczy, w odróżnieniu od zwykłych programów pisanych w językach wysokiego poziomu, cechuje się tym, że jest podzielony na dwie części, aplikacje. Jeden jest to poziom API biblioteki OpenCL, natomiast drugim jest jednostka obliczeniowa. Pierwszy wymieniony pracuje na poziomie systemu operacyjnego. Za pomocą tej aplikacji przygotowujemy dodatkowy program, który będzie wysyłany na jednostki obliczeniowe, nazywany kernelem. Za pomocą hosta wysyłane jest do jednostek obliczeniowych żądanie wykonania programu (jądra obliczeniowego), które jest napisane w innym języku niż program gospodarza, do tej pory był to OpenCL C, czyli język w większości zgodny ze standardem C99, aczkolwiek od wersji OpenCL 2.1, jest możliwość pisania kodu jądra w języku zgodnym ze standardem C++14.

Zagłębiając się w sposób, w jaki jednostka obliczeniowa wykonuje program, wartym wyjaśnienia jest pojęcie siatki obliczeniowej (z ang. Computational grid, w kodzie OpenCL przedstawiona jako NDRange). Za jej pomocą określa się sposób w jaki jednostka ma podzielić dane na poszczególne rdzenie obliczeniowe. W przypadku OpenCL może być ona jedno-, dwu-, lub trójwymiarowa. Sieć sama w sobie składa się z jednostek roboczych (z ang. work item), które z kolei tworzą grupy robocze (z ang. work group). Jest oczywiście możliwość odróżnienia pojedynczych jednostek roboczych od siebie za pomocą identyfikatorów globalnych w obrębie siatki (GID), za pomocą lokalnych identyfikatorów w obrębie grupy roboczej (LID), natomiast całą grupę roboczą dzięki identyfikatorowi całej grupy roboczej (WGID).

Kolejnym ważnym aspektem w modelu wykonawczym jest kontekst obliczeń oraz kolejka poleceń. Pierwsze z nich ustalane jest na poziomie programu hosta i polega na ustaleniu tego jakie urządzenia obliczeniowe będą brały udział w obliczeniach oraz w jakim środowisku będziemy pracować. Drugie natomiast jest silnie związane z pierwszym, ponieważ kolejka poleceń jest również tworzona podczas tworzenia kontekstu, gdyż ma on za zadanie za stworzenie zbioru poleceń. Kolejka poleceń, jak sugeruje nazwa, odpowiada za zakolejkowanie, a następnie wysłanie żądania wykonania programów (kerneli) na urządzeniach, jak i również za załadowanie i ściągnięcie danych z i do jednostek obliczeniowych. Oczywiście kolejka jest tworzona w programie hosta, więc właściwie zarządza wszystkimi transferami pomiędzy urządzeniami, na których wykonywane są kernele.

Ostatnim rodzajem jest model programowania, który został jeszcze podzielony na szablon dla danych i dla zadań. Schemat dla danych polega na oparciu się o siatkę obliczeniową. Chodzi o to, że wszystkie operacje w ramach grup roboczych są wykonywane równolegle, aczkolwiek host czeka na wykonanie zadań na jednostkach obliczeniowych, czyli jest on blokowany. Zadanie programisty polega na dobraniu odpowiedniej liczby grup, jednostek, itp. Ważnym aspektem w tym modelu jest synchronizacja. Ponieważ często problemy, które są rozwiązywane nie są idealne i dane dla każdego wątku nie są dostępne tylko dla niego, trzeba zadbać o synchronizację. Może być ona zastosowana w oparciu o barierę dla każdego pojedynczego "work-item", jak i również o kolejkę poleceń. Można również korzystać ze zdarzeń które są wysyłane po zakończeniu obliczeń na jądrze obliczeniowym.

Model zadaniowy nie korzysta z siatki obliczeniowej. Polega on w głównej mierze na wykorzystaniu w ramach grupy roboczej tylko jednej jednostki roboczej, aczkolwiek nadal można wykonywać zadania równolegle.

Z uwagi na temat pracy inżynierskiej wartym wspomnienia są również zalecenia firmy Intel odnośnie optymalizacji programów pisanych w OpenCL, są nimi:

- tworzenie wystarczającej liczby grup roboczych dla siatki obliczeniowej, rekomendowane jest ponad 1000 grup roboczych,
- unikanie małych grup roboczych, nie powinno się unikać używania maksimum dostępnego rozmiaru czyli 1024 (maksimum dla modelu Intel Xeon Phi 5120), jak i również trzeba mieć na uwadze, żeby rozmiar grupy roboczej był wielokrotnością liczby 32,
- unikanie kontroli programu na podstawie pierwszej jednostki roboczej w siatce z uwagi na to, że na jej podstawie jest ustalana wektoryzacja,
- zadbanie o to, by dostęp do danych był stały,
- ponowne użycie danych, dzięki pamięci podręcznej w obrębie grupy roboczej,
- używanie wbudowanego prefetchera który przesyła pamięć globalną do cache'u 500-1000 cykli zegarowych przed jej użyciem,
- nie używanie pamięci lokalnej oraz barier.

Na sam koniec warto wspomnieć o rzeczach związanych z samym API OpenCL, aniżeli z tym, w jaki sposób realizuje poszczególne funkcje. Istnieje wiele wrapperów, również w językach wysokopoziomowych, wspomagających pisanie kodu hosta w technologii OpenCL poczynając od C++, a kończąc na językach funkcyjnych typu F# czy Haskell. W przypadku kodu kerneli sprawa wygląda inaczej i została już opisana podczas omawiania modelu wykonawczego. Wraz z najnowszą wersją jest możliwość pisania kodu jądra w języku zgodnym ze standardem C99 albo C++14. Warto nadmienić również to, że OpenCL zapewnia nam własne typy danych, które są tożsame z typami dostępnymi w standardzie C99.

W najbliższej przyszłości technologia OpenCL będzie się szybko rozwijać, co można zaobserwować po czynnej współpracy z takimi firmami jak Intel, AMD, Nvidia przy tworzeniu standardu OpenCL 2.1. Jak i również po tym, że technologie heterogeniczne, które zapewniają bardzo wysoką wydajność, są coraz bardziej poszukiwane.

5. OPIS IMPLEMENTOWANYCH ALGORYTMÓW

W poniższym rozdziale opisano zaimplementowane algorytmy, które reprezentują trzy obszary problemów związanych z przetwarzaniem równoległym. Każdy z nich na podstawie przedstawionych charakterystyk został przełożony na język C++, który został rozszerzony o wykorzystanie technologii MPI, OpenMP oraz OpenCL.

5.1. Duża liczba prostych obliczeń (algorytm generujący obraz zbioru Mandelbrota)

W celu zaprezentowania aplikacji, która miałaby wykonywać dużo operacji obliczeniowych zdecydowano się na implementację algorytmu Mandelbrota. Było to spowodowane znajomością problemu z zajęć projektowych na przedmiocie „Przetwarzanie rozproszone CUDA”.

Do implementacji wybrano podstawowy algorytm tworzący obraz Mandelbrota, a nie jego uproszczoną wersję, taką jak obliczanie zbioru Julii, czy też wersje zmodyfikowane takie jak algorytm Buddhy. Napisany program polega na zobrazowaniu zbioru Mandelbrota. Jako dane wejściowe dla programu przyjęto wysokość i szerokość obrazu wyjściowego, które w dalszym opisie algorytmu są oznaczane odpowiednio jako h i w .

Algorytm ten został naszym wyborem dla problemu „dużej liczby obliczeń” z uwagi na to, że podając programowi zakładany obraz wyjściowy o rozmiarze $h \times w$ dla każdego piksela otrzymujemy k obliczeń.

Odnosnie samego algorytmu, by zdefiniować zbiór Mandelbrota dla każdego punktu na płaszczyźnie zespolonej (p) definiujemy nieskończony ciąg liczb zespolonych, o następujących właściwościach

$$z_{n+1} = z_n^2 + p \quad (5.1)$$

gdzie:

z_0 – jest równe wartości 0,

p – punkt na płaszczyźnie zespolonej, stała wartość.

Zbiór takich punktów, dla których powyższy ciąg nie dąży do nieskończoności, nazywamy zbiorem Mandelbrota. Można również w prosty sposób wykazać, że dzięki powyższym własnościom oraz wiedzy, że ciąg nie dąży do nieskończoności, otrzymujemy zależność

$$n \in N, |z_n| < 2 \quad (5.2)$$

posłuży ona przy implementacji algorytmu. W notacji matematycznej cały zbiór Mandelbrota można przedstawić jako zależność

$$M = \{p \in C : n \in N, |z_n| < 2\} \quad (5.3)$$

gdzie w zależnościach (5.2 i 5.3):

p – punkt na płaszczyźnie zespolonej,

n – należy do zbioru liczb naturalnych i oznacza liczbę początkowych wyrazów ciągu.

z_n – oznacza wartość ciągu dla pojedynczego piksela.

Oznacza to, że opisywany zbiór jest grupą liczb całkowitych takich, że dla każdego n należącego do zbioru liczb naturalnych zachodzi taka nierówność, że moduł z_n oznaczający właściwość punktu jest mniejszy od dwóch.

Mając na uwadze wszystkie powyższe fakty matematyczne oraz zagadnienia odnośnie samego zbioru Mandelbrota, nasz algorytm można przedstawić za pomocą przedstawionego pseudokodu.

```
Complex z = (0.0, 0.0)
Complex p = wartość zależna od położenia piksela
int iteration = 0;
const int MAX_ITERATION = 200;
while(z.Length() < 2.0 && iteration++ < MAX_ITERATION)
    z = (z*z) + p
return iteration
```

W przypadku realizowanej implementacji p oznacza przesunięcie względem początku obrazu (lewego, górnego rogu), natomiast obliczenia wykonywane na z_n są zgodne z obliczeniami na liczbach zespolonych, które opisano poniżej.

Liczba zespolona składa się z części rzeczywistej oraz części urojonej, co prezentuje zależność (5.4)

$$z = z_r + iz_i \quad (5.4)$$

Dodawanie liczb urojonych zdefiniowane jest jako zależność (5.5)

$$z_a + z_b = (z_{ar} + z_{br}) + i(z_{bi} + z_{ai}) \quad (5.5)$$

Potęgowanie jako zależność (5.6)

$$z^2 = (z_r^2 - z_i^2) + i(2 * z_r * z_i) \quad (5.6)$$

Natomiast moduł z liczby zespolonej jako zależność (5.7)

$$|z| = \sqrt{z_r^2 - z_i^2} \quad (5.7)$$

gdzie w powyższych wzorach:

z – liczba zespolona,

z_r – część rzeczywista liczby zespolonej,

z_i – część urojona liczby zespolonej,

i – jednostka urojona.

Dlatego też, dzięki powyższym działaniom, warunek, że moduł liczby z powinien być mniejszy od liczby 2, możemy zastąpić nierównością $z_r^2 + z_i^2 < 4$, gdzie z_r oraz z_i są wyjaśnione w powyższej zależności (5.7).

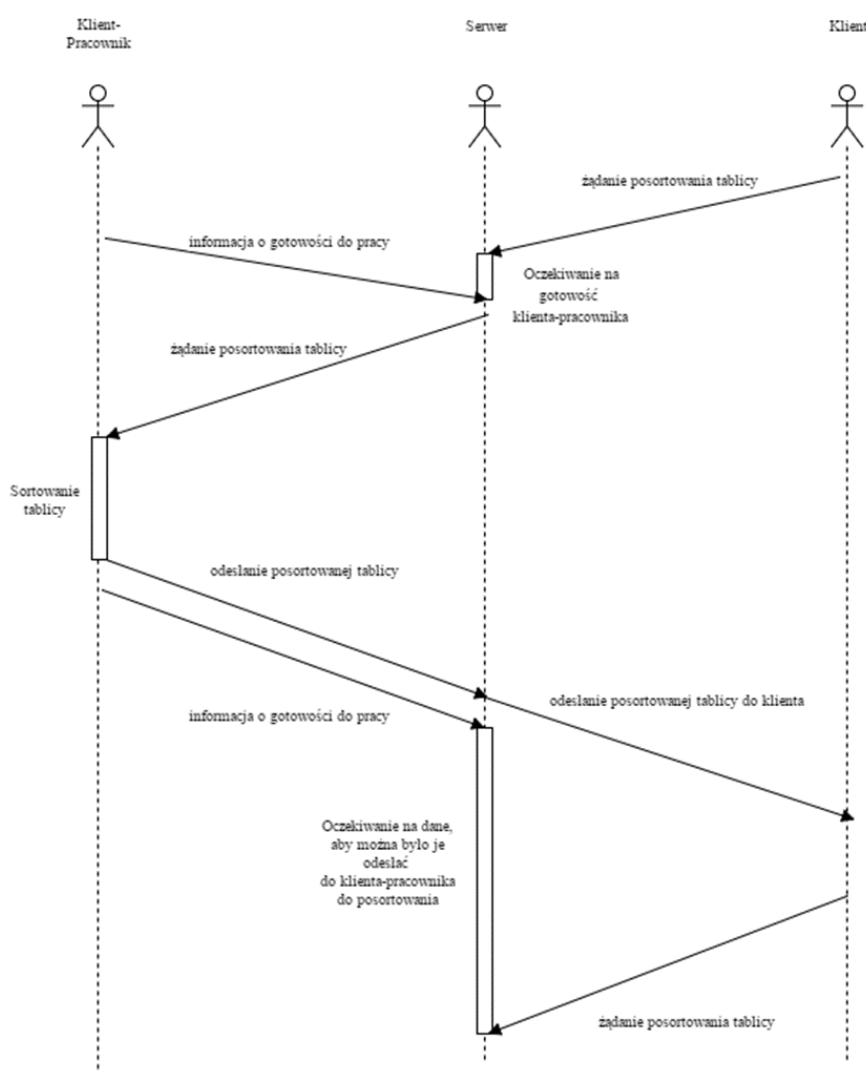
Algorytm zaimplementowano zgodnie z powyższymi wzorami, warto jedynie wspomnieć, że zwracana przez główną funkcję liczba iteracji, definiuje poziom jasności piksela obrazu wyjściowego. Tak powstaje wizualizacja zbioru Mandelbrota, który reprezentuje jeden z trzech rozwiązywanych problemów.

5.2. Duża liczba komunikacji (aplikacja typu klient-serwer)

W celu symulacji dużej liczby komunikacji zdecydowano się na implementację aplikacji podzielonej na serwer, czyli aplikację odbierającą i wysyłającą żądania od klientów oraz na „klientów-pracowników”, którzy odbieraliby żądania od serwera, obsługiwali je i wysyłali wynik z powrotem.

Stronę serwerową stanowi pojedyncza aplikacja, która jest podzielona na stronę przekierowującą żądania oraz na wykonującą właściwe obliczenia. W opisanym przypadku żądaniami są tablice liczb, które zostają posortowane, algorytmem sortowania przez scalanie, w ramach obsługi żądania.

W celu uwydatnienia w aplikacji problemu komunikacji zdecydowano, że stroną inicjalizującą kontakt jest klient, który początkowo wysyła informację do serwera o tym, że chce, aby dana tablica została posortowana. Natomiast serwer może taką tablicę odesłać do posortowania tylko wtedy, gdy otrzyma wiadomość od klienta-pracownika, że ten jest już gotów do pracy. Dopiero w momencie gdy serwer ma dwie informacje i od klienta, i od klienta-pracownika dopiero wtedy może zacząć wysyłać dane do klienta-pracownika w celu wykonania pracy. Rys. 5.1. obrazuje opisywany algorytm.



Rys. 5.1. Przedstawienie zaimplementowanej komunikacji w aplikacji symulującej działanie klient-serwer

Algorytm MergeSort [20] stosowany jest do sortowania danych przesłanych w żądaniu. W pierwszym kroku tablica jest dzielona na zbiory jednoelementowe. Następnie zwiększamy liczebność szeregu za każdym razem dwukrotnie, jednocześnie wykonując ich sortowanie. Poniżej przedstawiono dokładniej kroki związane z działaniem algorytmu.

Mając zbiór wejściowy:

[2,5,6,1,10,8,7,4]

Dzieli się ją na tablice jednoelementowe:

[2], [5], [6], [1], [10], [8],[7],[4]

Następnie łączy szereg tworząc zbiory $2*n$ elementowe (pamiętając, że nowy rozmiar nie może być większy od rozmiaru tablicy wejściowej), gdzie n określa aktualny rozmiar pojedynczego szeregu, jednocześnie je sortując, a więc otrzymujemy następujące zbiory:

[2,5], [1,6], [8,10], [4,7]

Krok ten wykonywany jest, tak długo, aż rozmiar struktury wynikowej będzie równy rozmiarowi tablicy wejściowej. Dzięki temu otrzymujemy rezultat sortowania:

[0,1,2,4,5,6,7,8,10]

Pseudokod algorytmu wygląda przedstawiony został na listingu 5.1.

Listing 5.1. Algorytm sortowania przez scalanie

```
mergesort(tab)
    firstTab, secondTab;
    if tab.size() == 1
        return tab
    else
        firstTab = tab[0..tab.size()/2]
        secondTab = tab[(tab.size()/2) + 1, tab.size()]
        mergesort(firstTab)
        mergesort(secondTab)
    return merge(firstTab, secondTab)

merge(firstTab, secondTab)
    returnArray
    while(firstTab.isnotempty() && secondTab.isnotempty())
        if(firstTab[iter] > secondTab[iter])
            returnArray.addToTheEnd(secondTab[iter])
            secondTab.removeAt(iter)
        else
            returnArray.addToTheEnd(firstTab[iter])
            firstTab.removeAt(iter)
    while(firstTab.isnotempty())
        returnArray.addToTheEnd(firstTab[iter])
        firstTab.removeAt(iter)
    while(secondTab.isnotempty())
        returnArray.addToTheEnd(secondTab[iter])
        secondTab.removeAt(iter)
    return returnArray
```

Choć algorytm ciężko jest opisać, jego pseudokod, przedstawiony rysunek zawierający diagram oraz przykład, powinny wiele wyjaśnić. Jego celem było przedstawienie dużej ilości komunikacji i sprawdzenie w dalszej części pracy, która technologia dla tego typu problemu sprawdza się najlepiej.

5.3. Obliczenia na dużej liczbie danych (algorytm genetyczny)

Celem zobrazowania problemu obliczeń na dużej liczbie danych zdecydowano się na implementację algorytmu genetycznego, którego zadaniem było zwrócenie obrazu złożonego z prostokątów, podobnego do obrazu wejściowego. Dany algorytm uznano za bardzo dobry problem przykładu pracy z dużą liczbą danych, które są m.in. tematem tego rozdziału.

W programie zawsze trzymano wszystkie prostokąty, które są malowane na obrazie, czyli n prostokątów na m obrazach, gdzie pojedynczy przedstawiony był jako struktura z dwoma punktami opisującymi przeciwległe wierzchołki. Zapisywany był również jego kolor. Dodatkowo w pamięci cały czas trzymano tablicę, która miała w sobie informacje o podobieństwie obrazów pomiędzy sobą, jak i również rysunek, na którym malowano wymienione prostokąty dla każdego obrazu w celu porównania otrzymanego z wejściowym.

Użyty algorytm prezentuje pseudokod przedstawiony na listingu 5.2.

Listing 5.2. Algorytm genetyczny

```
Calculate()
    GenerateRandomRectangles.For(Population)
    while(generation > 0)
        CompareRectanglesToInputImage().For(Population)
        SortComparison()
        RemoveImagesWhichAreNotAnElite.Last(Population-Elite)
        MutateEliteToGetNewImages(Elite)
        --generation
    CompareRectanglesToInputImage().For(Population)
    SortComparison()
    return generatedImages.First();
```

Jest to dosyć schematycznie przedstawiony algorytm, z uwagi na to wyjaśniono dokładniej poszczególne jego części.

Metoda `GenerateRandomRectangles.For(Population)` odpowiada za wygenerowanie n zdjęć składających się z k prostokątów, w tym przypadku, n jest populacją czyli liczbą osobników, natomiast liczba k jest stałą ustaloną z góry w programie. W implementowanej wersji jest liczba 50.

Funkcja `CompareRectanglesToInputImage().For(Population)` odpowiada za porównanie każdego pojedynczego obrazu złożonego z prostokątów do obrazu wejściowego. Odbywa się to poprzez obliczenie różnicy wartości każdego piksela dla całego obrazu.

Można to zobrazować za pomocą równania (5.1)

$$w = \frac{\sum_{i=1}^n \sqrt{(p1.r-p2.r)^2 + (p1.g-p2.g)^2 + (p1.b-p2.b)^2}}{\text{wysokość obrazu} * \text{szerokość obrazu} * \sqrt{255*255*3}} \quad (5.1)$$

gdzie:

w – wartość porównania,

h – wysokość obrazu,

w – szerokość obrazu,

p_1 – wartość piksela obrazu porównywanego,

p_2 – wartość piksela obrazu wejściowego,

r, g, b – składowe koloru, odpowiednio czerwony, zielony, niebieski, wartości z $\langle 0;255 \rangle$.

Metoda `SortComparison()` polega na posortowaniu listy zawierającej wszystkie wygenerowane obrazy na podstawie porównania do obrazu wejściowego.

Funkcja `RemoveImagesWhichAreNotAnElite.Last(Population-Elite)` polega na pozostawieniu w liście trzymającej wszystkie wygenerowane obrazy tylko tych, które należą do elity. Jest ona przekazywana jako parametr do programu i z reguły wynosi $0.1 * \text{populacja}$. Obrazy, które nie należą do elity, czyli miały gorsze wyniki porównania w tej funkcji, są usuwane z listy wszystkich wygenerowanych.

Procedura `MutateEliteToGetNewImages(Elite)` polega na wygenerowaniu nowych obrazów w miejscu tych, które zostały usunięte w opisywanej wyżej funkcji. Dodawanie nowych opiera się na mutacji dwóch obrazów z elity, a mutacja opiera się na podanym na listingu 5.3. pseudokodzie.

Listing 5.3. Mutacja

```
Mutate(firstImage, secondImage)
    newImage
    while(newImage.Rectangles.size() != firstImage.Rectangles.size())
        if(rand()%2)
            if(rand%2)
                newImage.Rectangles.Add(firstImage(iter))
            else
                newImage.Rectangles.Add(MutateRectangle(firstImage(iter)))
        else
            if(rand%2)
                newImage.Rectangles.Add(secondImage(iter))
            else
                newImage.Rectangles.Add(MutateRectangle(secondImage(iter)))

MutateRectangle(Rectangle)
    foreach(point : Rectangle.Points)
        point.x = rand() % width
        point.y = rand() % height
```

Można zauważyć, że czasami współrzędne pojedynczego prostokąta też ulegają mutacji.

Algorytm genetyczny jest najbardziej rozbudowany spośród opisywanych programów. Wpłynęło to zarówno na czas jego implementacji, jak i na liczbę mierzonych fragmentów kodu.

6. OPIS IMPLEMENTACJI ALGORYTMU MANDELBROTA

Wśród implementacji znajduje się aplikacja tworząca plik z obrazem zbioru Mandelbrota. Jej dokładna implementacja została opisana w tym rozdziale.

6.1. Szeregowy

Szeregową implementację programu, który wykonuje wizualizację zbioru Mandelbrota na zadanej płaszczyźnie, wykonano w następujący sposób. Przede wszystkim założono, iż program jako argumenty przyjmie dwie wartości. Po pierwsze jest to wysokość owej płaszczyzny, natomiast drugim argumentem jest jej szerokość. Z kolei wynikiem pracy aplikacji jest graficzny plik wynikowy w formacie PNG o nazwie `mandelbrot_cl`, który zawiera wygenerowany obraz zawierający odzwierciedlenie owego zbioru. Drugą ważną rzeczą, jeśli chodzi o przebieg programu, jest czas jego działania. Będzie on ważny przy analizie wydajności aplikacji zrównoleglonych. Przed przejściem do analizy właściwych wyników, należy zaznaczyć, iż w trakcie wykonywania obliczeń, obraz jest reprezentowany w pamięci jako obszar o rozmiarze wysokość razy szerokość bajtów. Wynika to z założenia, iż w celu optymalizacji dostępu do pamięci, wykorzystano, jako nośnik informacji o pojedynczym punkcie obrazu, typ `char`. Przyjęto, że obraz wynikowy będzie zawierał tylko stopnie jasności, od 0 do 255, a zatem potrzebny był tylko jeden bajt pamięci na punkt płaszczyzny.

Obszar ten został zaalokowany przy pomocy funkcji `_mm_malloc`.

```
char* results = (char*)_mm_malloc(sizeof(char) * width * height, ALLOC_ALIGN);
```

Stała `ALLOC_ALIGN` przyjęła wartość 64. Parametr ten wykorzystano w celu wyrównania początkowego adresu tablicy do wartości będącej wielokrotnością liczby 64, tak, aby z jednej strony zoptymalizować wczytywanie linii danych z pamięci głównej do pamięci podręcznej, a z drugiej, dla polepszenia stopnia wektoryzacji kodu. Należy jednak zauważyć, że dla drugiej kwestii wystarczyłoby wyrównanie do 32 bajtów, gdyż obecnie w procesorach nie są stosowane instrukcje wektorowe pracujące na rejestrach dłuższych, niż 256 bitów, a zatem 32 bajtów.

Przechodząc do obliczeń, należy zwrócić uwagę na następujące rzeczy. Po pierwsze, obliczenia wykonywane są wewnątrz podwójnej pętli `for`.

```
for (int y = 0; y < height; ++y)
{
    for (int x = 0 ; x < width; ++x)
    {
        [...]
    }
}
```

Zewnętrzna pętla indeksuje po wierszach, natomiast wewnętrzna po kolumnach. Zostało to zastosowane w celu uzyskania lokalności danych, a zatem by ponownie wykorzystać linie

danych, znajdujące się już w pamięci podręcznej poziomu L1, które zostały do niej wczytane w trakcie poprzednich iteracji. Potęgowane jest to faktem, iż całe obliczenia są zrobione w ten sposób, a co za tym idzie procesor jest w stanie rozpoznać schemat dostępu do pamięci, i sprzętowo zaczyna pobierać kolejne linie danych z pamięci, zanim jeszcze program dojdzie do instrukcji wymagających, bądź pobierających owe dane.

W ramach przytoczonej pętli wykonywane są obliczenia dla każdego punktu płaszczyzny. Z powodu wykorzystywania przez opisany wcześniej algorytm wyliczania zbioru Mandelbrota, konieczne było wykorzystanie struktury, która będzie reprezentowała takową liczbę w pamięci. Ze względu na wiedzę, w jaki sposób zaimplementowane są poszczególne operacje na liczbach zespolonych, zamiast wyboru istniejącego w wersji języka C++11 typu `Complex`, utworzono własną klasę, również o tej samej nazwie, która odwzorowuje liczbę zespoloną w pamięci, w postaci dwóch zmiennych typu `float`. Wynikało z wystarczającej dokładności takiej reprezentacji liczby zmiennoprzecinkowej dla rozwiązania problemu.

```
class Complex
{
    float Real;
    float Imaginary;
    [...]
};
```

W kwestii obliczeń ważne jest zastosowanie ograniczenia maksymalnej liczby iteracji algorytmu, która może zostać wykonana dla pojedynczego punktu. Przytaczając listing 6.1. jest to zmienna typu `int` o nazwie `bailout`, która ustawiona została na wartość 200.

Listing 6.1. Funkcja `BoundedOrbit` klasy `Complex`

```
float Complex::BoundedOrbit(Complex *startingPoint, float bound, int bailout)
{
    Complex multiply = startingPoint->multiply(*startingPoint);
    Complex z = this->add(multiply);
    for (int k = 0 ; k < bailout; ++k) {
        if (z.length() > bound)
        {
            return z.normalizedIterations(k, bailout);
        }
        multiply = z.multiply(z);
        z = this->add(multiply);
    }
    return FLT_MIN;
}
```

Po wykonaniu opisanych operacji oraz zwrocie z przytoczonej procedury, wynik jest konwertowany na typ `char`, w celu zmiany osiągniętej wartości na stopień jasności, a następnie zapisywany do wcześniej zaalokowanej dynamicznie tablicy. Wymagało to wykorzystania

poprawnej indeksacji, biorącej pod uwagę wartość zmiennej *y* jako numeru wiersza i zmiennej *x* jako przesunięcia w aktualnym rzędzie.

```
output[width * y + x] = GrayValue(count);
```

Po wykonaniu wszystkich obliczeń, zostaje wykonana konwersja obrazu do formatu PNG oraz zapis do pliku, co zostaje wykonane przy pomocy biblioteki `stb_image_write`. Na zakończenie aplikacji zostają zwolnione zasoby zaalokowane na potrzeby przechowywania płaszczyzny. Ze względu na zaalokowanie ich za pomocą funkcji `_mm_malloc`, konieczne jest zwolnienie ich za pomocą analogicznej procedury o sygnaturze `_mm_free(void* ptr)`.

```
_mm_free(results);
```

Na zakończenie opisu implementacji należy również wspomnieć o położeniu miejsc wywołania funkcji, które mierzą czas. Rozpoczynają się one przed inicjalizacją wspomnianego obszaru, a kończą tuż po wykonanych przez program obliczeniach.

```
mainTimer.Start();
char* results = (char*)_mm_malloc(sizeof(char) * width * height, ALLOC_ALIGN);
Mandelbrot(results, 0, width, height);
mainTimer.Stop();
```

6.2. MPI

Przy opisie implementacji w technologii MPI skupiono się głównie na realizacji komunikacji w środowisku, uznając jednocześnie, że ze względu na podobieństwo obliczeń do wersji szeregowej algorytmu, można pominąć w tym miejscu ich analizę.

Opisując program warto wyjaśnić początkowo zdefiniowane wartości, które później używane są w trakcie przesyłania wiadomości. Są to wartości tagów komunikatu, o których więcej jest napisane w rozdziale 4. Opis porównywanych technologii.

```
#define PARAMETERS 0
#define RESULTS_DATA 1
#define STARTING 2
```

Chcąc omówić je kolejno, należy wyjaśnić, że wartość `PARAMETERS` jest używana tylko podczas wysyłania komunikatów odpowiedzialnych za wysłanie samych parametrów. Stała `RESULTS_DATA` jest używana w komunikatach odpowiedzialnych za odbieranie danych z jednostek obliczeniowych. Wartość `STARTING` w chwili rozsyłania danych potrzebnych do rozpoczęcia obliczeń.

Oczywiście, pisząc o aplikacji napisanej w technologii MPI, należy pamiętać o inicjalizacji całego środowiska oraz o pobraniu informacji o identyfikatorze procesu oraz liczbie procesów, na których będzie pracował dany program. Jest to realizowane kolejno przez następujące funkcje.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &commSize);
MPI_Comm_rank(MPI_COMM_WORLD, &commRank);
```

Implementacja w MPI różni się od implementacji szeregowej tym, że na początku programu trzeba rozesłać wszystkie parametry wejściowe do każdego procesu MPI, z uwagi na to, że każdy z nich ma swój obszar pamięci, który nie jest współdzielony pomiędzy nimi. Jest to realizowane za pomocą danego fragmentu kodu przedstawionego na listingu 6.2.

Listing 6.2. Fragment odpowiedzialny za rozsyłanie parametrów wejściowych

```
if (commRank == 0)
{
    width = atoi(argv[1]);
    height = width;
    singlePackageOffset = (width * height)/((commSize-1));
    PACKAGE_AMOUNT = (width * height) / singlePackageOffset;

    int sendParameters[3];
    sendParameters[0] = width;
    sendParameters[1] = height;
    sendParameters[2] = singlePackageOffset;

    for (int i = 1; i < commSize; i++) {
        MPI_Isend(sendParameters, 3, MPI_INT, i, PARAMETERS, MPI_COMM_WORLD, &request[i-1]);
    }
    MPI_Waitall(commSize-1, request, statuses);
}
else
{
    int receiveParameters[3];
    MPI_Recv(receiveParameters, 3, MPI_INT, 0, PARAMETERS, MPI_COMM_WORLD, &status);
    width = receiveParameters[0];
    height = receiveParameters[1];
    singlePackageOffset = receiveParameters[2];
}
```

Gdzie proces host, którego zmienna commRank posiada wartość 0, zaczytuje parametry wejściowe i następnie wysyła je asynchronicznie poprzez funkcję MPI_Isend do wszystkich wątków, których liczba zapisana jest w zmiennej commSize. Jednocześnie następuje dopisanie zmiennej typu event do danej metody rozsyłającej w celu późniejszej weryfikacji, czy wysyłanie zostało zakończone. Po rozesłaniu wywołana jest metoda MPI_Waitall, która czeka na

zakończenie komunikacji asynchronicznej. Przekazywana w tym przypadku jest tablica request, która zawiera obiekty żądania, które to były przypisane do funkcji wysyłających.

Po stronie procesów posiadających wartość zmiennej commRank innej niż 0, wywoływana jest blokująca metoda MPI_Recv, która odbiera wysyłane parametry i następnie przypisuje je do zmiennych globalnych dla danego procesów.

Po rozesłaniu oraz odebraniu parametrów wejściowych następuje właściwa część programu, którą należy analizować biorąc pod uwagę każdą komunikację wykonaną pomiędzy procesem gospodarza, a procesami odpowiedzialnymi za liczenie. Główna pętla, oprócz inicjalizacji zmiennych czy alokacji tablic, rozpoczyna się poprzez rozesłanie początkowych danych do wszystkich procesów obliczeniowych, przy użyciu asynchronicznej metody MPI_Isend, co ukazuje poniższy fragment kodu:

```
for (int j = std::min(commSize-1, totalNumberOfIterations); j > 0 ; --j) {
    currentIndexes[j-1].indexes[0] = j-1;
    currentIndexes[j-1].currentlyCalculating = 0;
    MPI_Isend(currentIndexes[j-1].indexes, 1, MPI_INT, j, DATA, MPI_COMM_WORLD, &request);
}
```

Patrząc na ten fragment przedstawiający pętlę, można zauważyć, że jest to proste rozesłanie danych początkowych do procesów liczących z zapisaniem informacji, który proces otrzymał jakie przesunięcie. Jest ono wykorzystane przy obliczaniu otrzymanego fragmentu obrazu. Po wysłaniu pierwszych paczek zostają przesłane kolejne do procesów tak, aby te mogły zacząć odbierać dane już w trakcie wykonywania obecnych obliczeń. W innym przypadku ten fragment kodu nie zostanie wykonany:

```
for (int j = std::min(commSize-1, totalNumberOfIterations-(commSize-1)); j > 0 ; --j) {
    currentIndexes[j-1].indexes[1] = commSize-1+j-1;
    MPI_Isend(currentIndexes[j-1].indexes + 1, 1, MPI_INT, j, DATA, MPI_COMM_WORLD, &request);
}
```

Jak można zauważyć fragment kodu jest analogiczny do poprzedniego.

Po rozesłaniu wszystkich zadań, czyli w tym przypadku przesunięcia potrzebnego do obliczenia fragmentu obrazu, następuje wywołanie fragmentów kodu zbierających wyniki od procesów roboczych. Zrealizowane jest to za pomocą kodu przedstawionego na listingu 6.3.

Listing 6.3. Odbieranie wyników od wątków liczących

```
for (int i = PACKAGE_AMOUNT-1; i >= 2*(commSize-1); --i) {
    MPI_Probe(MPI_ANY_SOURCE, DATA, MPI_COMM_WORLD, &status);
    MPI_Recv(results+indexProvider(currentIndexes[...].indexes[...].currentlyCalculating)),
    singlePackageOffset, MPI_CHAR, status.MPI_SOURCE, DATA, MPI_COMM_WORLD, &status);
    currentIndexes[...].indexes[currentIndexes[...].currentlyCalculating] = i;

    MPI_Isend(currentIndexes[...].indexes + currentIndexes[...].currentlyCalculating, 1,
MPI_INT,
    status.MPI_SOURCE, DATA, MPI_COMM_WORLD, &request);

    currentIndexes[...].currentlyCalculating = (currentIndexes[...].currentlyCalculating
+1)%2;
}
```

Można zauważyć, że w tym przypadku została zastosowana komunikacja blokująca po stronie hosta z uwagi na to, że po stronie procesów obliczeniowych została wykorzystana komunikacja asynchroniczna. Natychmiast po odebraniu wyniku od procesu, który zakończył obliczenia, jest mu wysyłana paczka z danymi dotyczącymi przesunięcia tak, by wątek obliczeniowy mógł rozpocząć dalszą pracę. Po wykonaniu powyższej pętli, uruchamiany jest kolejny fragment aplikacji, który odpowiada za ostateczne zebranie wszystkich wyników od procesów, które liczyły swoje ostatnie paczki (listing 6.4.).

Listing 6.4. Ostateczne zebranie wszystkich wyników od procesów

```
for (int i = std::min(2*(commSize-1), totalNumberOfIterations); i > 0; --i) {
    MPI_Probe(MPI_ANY_SOURCE, DATA, MPI_COMM_WORLD, &status);
    MPI_Recv(results +
indexProvider(currentIndexes[...].indexes[currentIndexes[...].currentlyCalculating]),
    singlePackageOffset, MPI_CHAR, status.MPI_SOURCE, DATA, MPI_COMM_WORLD, &status);
    currentIndexes[...].currentlyCalculating = (currentIndexes[...].currentlyCalculating
+1)%2;
}
```

Widać, że tak jak poprzednio, użyto tutaj blokującej komunikacji odbierającej jedynie wynik. Nie ma żadnego odesłania danych do procesu, który skończył obliczenia. Następnym krokiem jest wysłanie do wszystkich procesów informacji, że program kończy swoje działanie, w związku z czym wysyłana jest informacja o zakończeniu działania. Jest to realizowane za pomocą następującego fragmentu kodu:

```
for (int i = commSize-1; i > 0; --i) {
    MPI_Isend(nullptr, 0, MPI_INT, i, ENDGEN, MPI_COMM_WORLD, &request);
}
```

Jest to koniec kodu gospodarza, aczkolwiek warto omówić jeszcze, co dzieje się w wątku obliczeniowym. Rozpoczyna się on od ustawienia zmiennych oraz alokacji tablicy. Przed główną

pętla, wykonywaną na wątku obliczeniowym, można zobaczyć rozpoczęcie komunikacji z hostem w postaci:

```
MPI_Recv(inValue, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, status);
```

Powyższa linia ma za zadanie pobranie pierwszej wartości przesunięcia potrzebnej do rozpoczęcia liczenia. Zrealizowane jest to w podany sposób, aby spowodować nakładanie się obliczeń i komunikacji. Poniżej mamy już główną pętlę procesu obliczeniowego, która kończy się w przypadku otrzymania informacji o końcu przetwarzania, czego realizację można zauważyć w warunku znajdującym się w pętli while (listing 6.5.).

Listing 6.5. Główna pętla procesu obliczeniowego

```
while (status->MPI_TAG != ENDGEN) {
    MPI_Irecv(inValue+((currentlyCalculating+1)%2), 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
        request+((currentlyCalculating+1)%2));

    Mandelbrot(results+currentlyCalculating*singlePackageOffset,
        singlePackageOffset*inValue[currentlyCalculating], width, height, singlePackageOffset);

    MPI_Isend(results+currentlyCalculating*singlePackageOffset, singlePackageOffset, MPI_CHAR,
        0, DATA, MPI_COMM_WORLD, request + currentlyCalculating);
    currentlyCalculating = (currentlyCalculating+1)%2;

    MPI_Wait(request + currentlyCalculating, status);
};
```

Jak można zauważyć, wewnątrz pętli rozpoczyna się od pobierania kolejnej wartości przesunięcia w sposób asynchroniczny. W tym samym czasie wywoływana jest funkcja, która odpowiada za obliczenie wartości pikseli dla części obrazu, która jest zaimplementowana analogicznie do funkcji z wersji szeregowej. Po wykonanych obliczeniach następuje odesłanie wyniku do hosta, a następnie bariera, która czeka na zakończenie metody odbierającej nowe przesunięcie.

6.3. OpenMP

Omawiając implementacje algorytmu Mandelbrota w technologii OpenMP, trzeba mieć na uwadze, że implementacje funkcji liczących nie zostały w żadnym stopniu zmodyfikowane. Jedyną rzeczą, wartą wspomnienia, są dyrektywy preprocesora, które pozwoliły napisać aplikację w sposób zrównoleglony na urządzeniu Intel Xeon Phi.

Wykonując opis implementacji, należy na wstępie wyjaśnić znaczenie zdefiniowanych wartości:

```
#define ALLOC alloc_if(1) free_if(0)
#define FREE alloc_if(0) free_if(1)
#define REUSE alloc_if(0) free_if(0)
#define ALLOCANDFREE alloc_if(1) free_if(1)
```

Oznaczają one kolejno, że przy użyciu dyrektyw z rodziny offload oraz użyciu stałej ALLOC, oznaczony obszar zostanie zaalokowany na początku bloku oznaczonego ową dyrektywą, a jednocześnie nie zostanie on zwolniony na jej końcu. Odwrotny efekt ma wartość FREE, która nie wykonuje alokacji, lecz tylko zwolnienie pamięci. REUSE z kolei, nie wykonuje żadnej operacji, a ALLOCANDFREE wykonuje obie te operacje.

Analizując dalszą część kodu, można zaobserwować dyrektywy, którymi oznaczone są definicje funkcji.

```
#pragma offload_attribute (push, target(mic))
#pragma offload_attribute (pop)
```

Oznaczają one, że dana procedura będzie dostępna na koprocesorze poprzez wygenerowanie odpowiedniego kodu dla architektury Intel MIC. Bez tych dyrektyw, otaczających funkcje, program skończyłby swoje działanie z błędem, ponieważ koprocesor nie miałby możliwości ich wywołania.

Następne dyrektywy znajdują się już w głównej liczącej funkcji, czyli w Mandelbrot(). Pierwsza z nich

```
#pragma offload target(mic) mandatory out(output:length(width*height) ALLOCANDFREE){...}
```

odpowiada za przesłanie na koprocesor tablicy o nazwie output oraz rozmiarze, wyrażonym jako iloczyn dwóch zmiennych width i height. Natomiast po zakończeniu tej pragmy, dany obszar pamięci powinien zostać wyczyszczony. Można zwrócić w tym miejscu uwagę na wykorzystany atrybut mandatory, który gwarantuje, że w przypadku braku możliwości uruchomienia oznaczonego bloku kodu po stronie koprocesora, program zostanie przerwany i zwróci błąd. Został on wykorzystany ze względu na temat pracy, wymagający użycia koprocesora.

Kolejną dyrektywą jest

```
#pragma omp parallel
```

która oznacza, jak to było już opisane w przypadku OpenMP w rozdziale 4. Opis porównywanych technologii, że dana sekcja będzie wykonywana równolegle na wszystkich wątkach. Ostatnią pragmatą OpenMP jest

```
#pragma omp for collapse(2)
```

Oznacza ona, że, przy dystrybucji iteracji do wątków, będą brane iteracje obu pętli for, które poprzedza wspomniana dyrektywa. Pozwala zwiększyć to granulację zadań, a co za tym idzie polepszyć zbalansowanie obciążenia poszczególnych wątków.

Implementacja algorytmu Mandelbrota w OpenMP, jak można zauważyć, nie jest zbyt skomplikowana, ponieważ polega głównie na wstawieniu odpowiednich dyrektyw odpowiedzialnych za zrównoleglenie kodu.

6.4. OpenCL

W przypadku implementacji algorytmu rozwiązującego problem przedstawienia zbioru Mandelbrota, w technologii OpenCL, należy wziąć pod uwagę zmianę kontekstu parametrów wejściowych programu. Podobnie, jak w pozostałych implementacjach ze zrównoległym kodem obliczeniowym, wyłącznie pierwszy argument programu decyduje o rozmiarze rozwiązywanego problemu. Dokładniej decyduje on zarówno o wysokości, jak i szerokości generowanego obrazu, gdyż tworzony jest on w postaci kwadratu. Drugim parametrem przekazywanym do programu jest rozmiar siatki obliczeniowej wykorzystywanej przez technologię OpenCL. Implementacja tego algorytmu została zmieniona w stosunku do wersji szeregowej tak, aby trzecim argumentem była nazwa pliku, do którego zostanie zapisany wygenerowany obraz. Oprócz owego produktu wyjściowego aplikacji, ważna w kontekście całej pracy jest linia, wypisywana na konsolę, zawierająca zarówno informację, o parametrach programu odpowiedzialnego za wyliczenia zbioru, jak i poszczególne czasy, dające informacje o czasie trwania poszczególnych konstrukcji.

```
OpenCL,64,512,80362000,1823722000
```

Oddzielone przecinkami wartości, odpowiadają kolejno technologii, w jakiej zaimplementowano dany program, rozmiarowi siatki obliczeniowej, długości jednego boku obrazu, czasu trwania aplikacji (bez tworzenia kontekstu i kompilacja kodu kernela), oraz czasowi trwania całego programu.

Przechodząc do analizy głównej części implementacji, można zauważyć rozległy kod inicjalizujący środowisko, w którym wykonywane są zrównoleglone obliczenia. Wynika to z faktu, iż charakterystyka technologii OpenCL wymaga bezpośredniej kontroli nad wszystkimi jej konstrukcjami. Tak też odpowiadający za to obszar kodu rozpoczyna się od wywołania funkcji pobierającej dostępne platformy OpenCL'owe na maszynie, na której uruchamiany jest program.

```
std::vector<cl::Platform> platforms;  
cl::Platform::get(&platforms);
```

Następnie w celu otrzymania uchwytu do urządzenia typu koprocessor, a zatem do karty Intel Xeon Phi, została wywołana metoda pobierająca z wybranej platformy urządzenia o zadanym typie.

```
std::vector<cl::Device> devices;  
platforms[0].getDevices(CL_DEVICE_TYPE_ACCELERATOR, &devices);
```

Należy zauważyć, iż nie zostało wykonane tutaj żadne sprawdzenie dostawcy wybranej platformy ze względu na to, że na wykorzystywanej maszynie nie znajdowały się środowiska innych producentów niż dostarczane przez firmę Intel.

W następnym kroku został stworzony obiekt typu context, który jest potrzebny w kolejnych funkcjach. Między innymi dotyczy to fragmentu odpowiedzialnego za zbudowanie kodu kernela, który odpowiada za implementację właściwej funkcji liczącej. W celu kompilacji kodu z żądaniem przeprowadzenia częściowej optymalizacji zastosowano następujące dwie flagi:

```
char *buildOptions = "-cl-finite-math-only -cl-no-signed-zeros";
```

Odpowiadają one kolejno za założenie, iż w trakcie obliczeń zmiennoprzecinkowych nie zostaną wykorzystane wartości odpowiadające NaN, ani liczbie nieskończenie dużej ze znakiem plus, bądź minus. Druga flaga odpowiada za traktowanie zer z oboma znakami jako tę samą liczbę.

Budowa programu, a następnie pobranie obiektu kernela przebiega w sposób ukazany na listingu 6.6.

Listing 6.6. Budowa programu oraz pobranie obiektu kernela

```
auto program = cl::Program(context, kernelSource, false, &error);
program.build(devices, buildOptions);
if(error != CL_SUCCESS) {
    printf("Error at creating program : %d\n", error);
    printf("Build log %s \n",
        program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0]));
}
auto kernel = cl::make_kernel<cl::Buffer&, cl::Buffer&> (program, "MainKernel", &error);
if(error != CL_SUCCESS) printf("Error at getting kernel : %d\n", error);
```

Należy zwrócić uwagę na sposób sygnalizacji błędów funkcji biblioteki OpenCL, realizowanej jako parametr wyjściowy owych funkcji. Został on wykorzystany w programie jako zabezpieczenie, przed omyłkowym uznaniem działania błędnego programu za poprawny. Dodatkowo, w razie wystąpienia błędu, zostaje wyświetlony zrzut informacji związanych z niepoprawnym budowanym kodem.

Kolejnymi tworzonymi strukturami jest kolejka rozkazów.

```
auto commandQueue = cl::CommandQueue(context, 0, &error);
if(error != CL_SUCCESS) printf("Error at creating command queue : %d\n", error);
```

oraz następnie bufory (listing 6.7.), czyli obszary pamięci dostępne dla urządzeń związanych z kontekstem.

Listing 6.7. Utworzenie buforów

```
auto inBuffer = cl::Buffer(context, CL_MEM_READ_ONLY, sizeof(MandelbrotKernelParameters),
    NULL, &error);
if(error != CL_SUCCESS) printf("Error at creating input buffer : %d\n", error);
const auto sizeOfOutput = parameters.width * parameters.height * sizeof(char);
auto outBuffer = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeOfOutput, NULL, &error);
if(error != CL_SUCCESS) printf("Error at creating output buffer : %d\n", error);
```

Można tutaj zauważyć zarówno wykorzystanie, w przypadku bufora `outBuffer`, pamięci globalnej, jak i pamięci typu stałego, `const`, dla obiektu `inBuffer`. Sterowane jest to za pomocą typu wyliczeniowego `cl_mem_flags`, który w przypadku wartości `CL_MEM_READ_ONLY` prosi o alokację pamięci typu `const`, która jest niemodyfikowalna przez kod kernela.

Przed wykonaniem samego algorytmu konieczne jest jeszcze zajęcie obszaru pamięci na obraz.

```
auto output = (char*)_mm_malloc(sizeofOutput, ALLOC_ALIGN);
if(!output) printf("Error when allocating memory on host\n");
```

Pamiętając przy tej operacji o konieczności sprawdzenia poprawności alokacji pamięci.

Ostatnią sprawą jest zainicjalizowanie struktury `MandelbrotKernelParameters` oraz zapisanie jej stanu do bufora (listing 6.8.), dzięki czemu przekazywane są argumenty do kodu wykonywanego na koprocessorze.

Listing 6.8. Zapisanie parametrów do bufora

```
parameters.width = atoi(argv[1]);
parameters.height = parameters.width;
parameters.maxIterations = 200;
[...]
auto totalLength = parameters.height*parameters.width;
parameters.totalLength = totalLength;
commandQueue.enqueueWriteBuffer(inBuffer, CL_FALSE, 0, sizeof(MandelbrotKernelParameters),
&parameters, NULL, &event);
```

Dosyć ważnym aspektem wywołania funkcji `enqueueWriteBuffer` jest przekazanie adresu zmiennej `event`, względem której zostanie wykonana synchronizacja tak, aby wywołanie kernela w następnym kroku zostały wykonane dopiero po zapisie danych. Ma to miejsce dzięki przekazaniu jako drugi argument zmiennej w konstruktorze klasy `EnqueueArgs`.

```
auto kernelEvent = kernel(cl::EnqueueArgs(commandQueue, event, cl::NDRange(numberOfThreads),
cl::NDRange(numberOfThreads == 16 ? 16 : 32)), inBuffer, outBuffer);
```

W tym miejscu należy zauważyć dwie rzeczy. Pierwszą jest sposób przekazania argumentów do kernela. Zostaje to dokonane za pomocą przekazania do funktora dwóch ostatnich argumentów. Drugą kwestią jest rozmiar pojedynczej grupy, czego wielokrotnością jest rozmiar siatki obliczeniowej. Z tego też powodu, konieczne jest sprawdzenie, czy rozmiar siatki nie jest mniejszy niż 32, a jeśli tak właśnie jest, ustalenie rozmiaru grupy na 16 zamiast 32.

Ostatnimi operacjami, po stronie aplikacji hosta, są kolejno zlecenie odczytania wartości wyniku obliczeń oraz wywołanie metody gwarantującej wykonanie wszystkich żądań w kolejce.

```
commandQueue.enqueueReadBuffer(outBuffer, CL_FALSE, 0, sizeofOutput, output, &kernelEvents, &event);
commandQueue.finish();
```

Przechodząc do kodu realizującego właściwe algorytmowi obliczenia, należy przede wszystkim zauważyć, iż rozszerzenie pliku, zawierającego ów kod, różni się od rozszerzeń

pozostałych plików. Mianowicie jest to plik z końcówką .cl, a zatem symbolizuje on, iż jest to kod napisany zgodnie z wymaganiami i standardem technologii OpenCL, i może zostać zbudowany oraz wykorzystany w jego infrastrukturze. Jak już wcześniej zostało wspomniane, jest to standard zbliżony do C99, rozszerzony o konstrukcje realizujące m.in. wskazanie położenia w pamięci wybranej zmiennej. Przykładem może być atrybut __global sygnalizujący pole znajdujące się w pamięci globalnej. Drugim z najważniejszych atrybutów jest słowo kluczowe __kernel, który sygnalizuje procedura, która ma być dostępna z poziomu programu hosta w celu wykonania obliczeń.

Listing 6.9. Zebranie parametrów wejściowych w jądrze obliczeniowym

```
__kernel void MainKernel (__global Parameters *parameters, __global char *outputBuffer)
{
    int lid = get_local_id(0), gid = get_global_id(0), globalCount = get_global_size(0);
    int threadCount = get_local_size(0);
    int pixelsToCount = parameters->totalLength / globalCount;
    [...]
}
```

Przede wszystkim funkcja ta (listing 6.9.) przyjmuje dwa parametry, będące wskaźnikami na obszary pamięci zawierające kolejno parametry programu, oraz wolną pamięć przeznaczoną na wynik obliczeń. Aby zrównoleglić algorytm, konieczne było pobranie informacji na temat położenia na siatce obliczeniowej danego kernela, a także jej rozmiar, dzięki czemu możliwe zostało określenie, jaki zakres danych ma zostać zrealizowany w danym zadaniu.

Drugą ważną decyzją, była zmiana przebiegu pętli (listing 6.10.). Pętla podwójna została zastąpiona pojedynczą ograniczoną wyliczoną wcześniej wartością zmiennej pixelsToCount.

Listing 6.10. Główna pętla obliczeniowa

```
for (int j = 0; j < pixelsToCount; ++j) {
    starting.x = -2.5f + 3.5f*((gid*pixelsToCount+j)%parameters->width)/parameters->width;
    starting.y = -1.25f + 2.5f*((gid*pixelsToCount+j)/parameters->width)/parameters->height;
    outputBuffer[(gid*pixelsToCount) + j] = (char)(CalculateSinglePoint(starting, 200));
}
```

Jak można zauważyć, opisana wcześniej charakterystyka kerneli OpenCL wymusiła zmianę indeksacji w tablicach, co spowodowało konieczność wzięcia pod uwagę przesunięcia początkowego dla danego wywołania procedury.

Ze względu na krytyczność wydajności aplikacji, zdecydowano się na wykorzystanie atrybutu inline w przypadku funkcji wykonujących obliczenia na liczbach zespolonych, aby zmniejszyć liczbę rozkazów skoku w trakcie pracy kernela.

```
inline Complex Add([...]) { return (Complex)(left.x+ right.x, left.y+ right.y); }
```

Ostatnią kwestią, którą należy poruszyć, jest realizacja liczby zespolonej w przypadku kernela. Ze względu na brak klas, a zatem niemożliwość ponownego wykorzystania

zaimplementowanego w wersji szeregowej kodu, zdecydowano się na wykorzystanie istniejącego typu `float2`, który jest parą dwóch liczb zmiennoprzecinkowych pojedynczej precyzji. Dla zwiększenia czytelności nadano temu typowi alias `Complex`. Obliczenia zostały zaimplementowane w dokładnie ten sam sposób, co w wersji oryginalnej.

7. OPIS IMPLEMENTACJI APLIKACJI TYPU KLIENT-SERWER

Jednym ze stworzonych programów jest aplikacja typu klient-serwer. Jej dokładny opis implementacyjny został przedstawiony w tym rozdziale.

7.1. Szeregowy

Zadaniem programu jest posortowanie tablic generowanych na jego początku. Analizę szeregowej implementacji problemu przetwarzania żądań należy rozpocząć od przedstawienia parametrów wejściowych oraz wyjściowych wykonanego programu. Wartościami, które przekazywane są do programu są kolejno rozmiar pojedynczego żądanie, definiujący rozmiar nieposortowanej tablicy, a także liczba żądań do przetworzenia. Wynikiem działania programu jest wartość, mediana, czasu, który został poświęcony na posortowanie pojedynczej tablicy.

Omawiając szeregową implementację, należy zwrócić uwagę na jej bardzo prostą realizację (listing 7.1.), co wynika z faktu, iż przy jednowątkowym podejściu, wystarczy kolejno przetwarzać zadane żądania, czyli w tym przypadku sortować kolejne tablice wejściowe.

Listing 7.1. Główna pętla programu klient-serwer

```
int* array = (int*)_mm_malloc(sizeof(int*) * numberOfArrays * arraySize, ALLOC_ALIGN);
for(int arrayCounter = 0; arrayCounter < numberOfArrays; arrayCounter++)
{
    GenerateArray(array + (arrayCounter * arraySize), arraySize);
    [...]
    MergeSort(array + (arrayCounter * arraySize), arraySize);
    [...]
}
_mm_free(array);
```

Podczas, gdy zadaniem funkcji `GenerateArray()` jest generowanie tablicy składającej się z losowych liczb, należy zwrócić uwagę na implementację algorytmu sortującego, która została zrealizowana w ramach procedury `MergeSort()`. Jest to realizacja algorytmu sortowania przez scalanie w wersji iteracyjnej, co jest wynikiem braku wsparcia funkcji rekurencyjnych w później omawianej technologii OpenCL w wersji 1.2, a co za tym idzie, ze względu na charakter pracy, wymusiło ujednolicenie implementacji algorytmu sortującego. Należy tutaj zaznaczyć, iż realizacja owego algorytmu zakłada, że w tablicy sortowanej nie znajdą się liczby równe maksymalnej wartości dla typu `int`, gdyż względem tej liczby następuje scalenie tablic składowych (listing 7.2.).

W opisywanym rozdziale nie skupiono się, na dokładnym opisie algorytmu `MergeSort`, gdyż jest to tematem rozdziału 5. Opis implementowanych algorytmów, w którym wyjaśniono w jaki sposób przebiega jego działanie.

Początkowo prosty algorytm, ulega skomplikowaniu podczas przełożenia na testowane technologie, co zostało opisane w dalszych rozdziałach.

Listing 7.2. Scalanie posortowanych podtablic

```
leftArray[leftArraySize - 1] = INT_MAX;
[...]
```

```
rightArray[rightArraySize - 1] = INT_MAX;
for (int k = startLeftIndex, i = 0, j = 0; k < endRightIndex; ++k)
{
    array[k] = leftArray[i] <= rightArray[j] ? leftArray[i++] : rightArray[j++];
}
```

7.2. MPI

Analizując implementację aplikacji, zrealizowaną w technologii MPI, skupiono się na realizacji komunikacji, która miała w niej kluczową rolę. Oczywiście warto wspomnieć o inicjalizacji środowiska poprzez funkcję `MPI_Init_Thread` oraz o deklaracji stałych odpowiedzialnych za oznaczanie różnych typów przesyłanych wiadomości.

```
#define PARAMETERS 0
#define RESULTS_DATA 1
#define STARTING 2
#define STATUS 3
#define FINISH 4
```

W tym przypadku było dostępnych pięć typów danych wysyłanych pomiędzy procesami. Kolejno są to parametry, wyniki, tablica startowa, która rozpoczynała obliczenia na wątku przeznaczonym do tego, status informujący o tym, czy wątki odpowiedzialne za liczenie są gotowe do przyjęcia danych, oraz stała `FINISH`, która sygnalizowała zakończenie obliczeń.

Przechodząc do analizy programu można zauważyć, że początkowo są ustawiane zmienne, które będą potrzebne do działania programu, przez proces hosta, a następnie rozsyłane są one do pozostałych procesów. Jest to zrealizowane poprzez funkcje `MPI_Send` i `MPI_Recv`, czyli zwykłe funkcje blokujące, co ukazuje listing 7.3.

Listing 7.3. Przesyłanie parametrów początkowych

```
if (me == 0) {
    [...]
    arraySize = atoi(argv[1]);
    numberOfArrays = atoi(argv[2]);

    int sendParameters[2];
    sendParameters[0] = arraySize;
    sendParameters[1] = numberOfArrays;

    for (int i = 1; i < numberOfProcesses; ++i) {
        MPI_Send(sendParameters, 2, MPI_INT, i, PARAMETERS, MPI_COMM_WORLD);
    }
    array = (int *) _mm_malloc(sizeof(int) * numberOfArrays * arraySize,
ALLOC_ALIGN_TRANSFER);
    for (int arrayCounter = 0; arrayCounter < numberOfArrays; arrayCounter++) {
        GenerateArray(array + (arrayCounter * arraySize), arraySize);
    }
}
else {
    int receiveParameters[2];
    MPI_Recv(receiveParameters, 2, MPI_INT, 0, PARAMETERS, MPI_COMM_WORLD, &status);
    arraySize = receiveParameters[0];
    numberOfArrays = receiveParameters[1];
}
```

Następnie można zauważyć podział strumienia działania kodu hosta na dwa oddzielne wątki poprzez użycie sekcji z API OpenMP.

```
#pragma omp parallel private(status)
{
    #pragma omp sections
    {
        #pragma omp section
        {...}
        #pragma omp section
        {...}
    }
}
```

Było to spowodowane modelem tego rozwiązania, gdzie założone istnienie klienta, który wysyła żądania posortowania tablicy, natomiast serwer odbiera je i oddelegowuje wykonanie przetwarzania na koprocesor, a następnie, w jak najkrótszym czasie, odbiera posortowaną tablicę i odsyła ją klientowi.

Analizując po kolei kod (listing 7.4.) możemy zauważyć, że, podobnie jak w przypadku algorytmu Mandelbrota, następuje tutaj rozesłanie do procesów liczących danych początkowych, na których mają się opierać ich obliczenia. Z tą różnicą, że są one wysyłane dopiero po

otrzymaniu wiadomości typu STATUS, a same dane są wysyłane do konkretnie tego procesu, który wysłał taką wiadomość.

Listing 7.4. Rozsyłanie żądań do procesów obliczeniowych

```
while (actualStartingPoint + arraySize <= numberOfArrays * arraySize) {int ready = 1;
    MPI_Recv(&ready, 1, MPI_INT, MPI_ANY_SOURCE, STATUS, MPI_COMM_WORLD, &status);
    if (ready == 1 && (actualStartingPoint < numberOfArrays * arraySize)) {
        startingForProcess[status.MPI_SOURCE - 1] = actualStartingPoint;
        howManyRequests = howManyRequests % (numberOfProcesses-1);
        MPI_Isend(&(array[actualStartingPoint]), arraySize, MPI_INT, status.MPI_SOURCE,
        STARTING, MPI_COMM_WORLD, &request[howManyRequests++]);
        actualStartingPoint = actualStartingPoint + arraySize;
    }
}
MPI_Waitall(howManyRequests, request, statuses);
finishStatus = 0;
for (int i = 1; i < numberOfProcesses; ++i)
    MPI_Isend(&finishStatus, 1, MPI_INT, i, FINISH, MPI_COMM_WORLD, &request[i-1]);
MPI_Waitall(numberOfProcesses-1, request, statuses);
Barrier(finishWork) = true;
```

Jeżeli zostały wysłane wszystkie żądania, ustawiona zostaje blokada MPI_Waitall, która czeka, aż każdy proces odbierze do końca dane wejściowe. Następnie rozsyłany jest komunikat o zakończeniu obliczeń do procesów obliczeniowych i ustawiana jest zmienna statyczna, dla wątku o identyfikatorze równym zero, sygnalizująca zakończenie pracy. Flaga ta pozwala na zakończenie obu sekcji działających na pierwszym wątku.

Listing 7.5. Kod sekcji drugiej

```
int howManyRequests = 0;
while (1) {
    howManyRequests = howManyRequests % (numberOfProcesses-1);
    MPI_Probe(MPI_ANY_SOURCE, RESULTS_DATA, MPI_COMM_WORLD, &status);
    MPI_Irecv(&(array[startingForProcess[status.MPI_SOURCE - 1]]), arraySize, MPI_INT,
    status.MPI_SOURCE, RESULTS_DATA, MPI_COMM_WORLD, &request[howManyRequests++]);
    if(Barrier(finishWork))
    {
        MPI_Waitall(howManyRequests, request, statuses);
        break;
    }
}
```

Jak można zauważyć, w drugiej sekcji (listing 7.5.) wykonywanej w procesie gospodarza (id = 0) następuje jedynie odbiór danych od jednostek obliczeniowych. Natomiast jeżeli zmienna finishWork zostanie ustawiona na wartość prawda, zostanie przerwana praca sekcji numer dwa. Przez to program będzie mógł działać dalej i nie zablokuje się w nieskończonej pętli w tym fragmencie kodu. W ten sposób można uzyskać pewność, że wszystkie obliczenia w procesach

obliczeniowych zostały zakończone, ponieważ w sekcji numer jeden, po wysłaniu zakończenia pracy, ustawiana jest bariera, a procesy obliczeniowe, przed otrzymaniem wiadomości o końcu pracy, kończą swoje wszystkie obliczenia (listing 7.6.).

Listing 7.6. Kod procesu slave

```
MPI_Send(&finishStatus, 1, MPI_INT, 0, STATUS, MPI_COMM_WORLD);
finishStatus = 0;
while (true) {
    if(statusRequest != 0)
        MPI_Wait(&statusRequest, &singleStatus);
    if(sendRequest != 0)
        MPI_Wait(&sendRequest, &singleStatus);
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &singleStatus);
    if (singleStatus.MPI_TAG == STARTING) {
        MPI_Irecv(smallArray, arraySize, MPI_INT, 0, STARTING, MPI_COMM_WORLD, &receiveRequest);
        MPI_Wait(&receiveRequest, &singleStatus);

        MergeSort(smallArray, arraySize);

        MPI_Isend(smallArray, arraySize, MPI_INT, 0, RESULTS_DATA, MPI_COMM_WORLD, &sendRequest);

        finishStatus = 1;
        MPI_Isend(&finishStatus, 1, MPI_INT, 0, STATUS, MPI_COMM_WORLD, &statusRequest);
    }
    else if (singleStatus.MPI_TAG == FINISH) {
        break;
    }
}
```

W fragmencie kodu (listing 7.6.) można zauważyć, że tak jak i w innych implementacjach programów w MPI, realizowane jest nakładanie obliczeń oraz komunikacji poprzez wysłanie wiadomości do hosta o gotowości przed wejściem do głównej pętli, za pomocą MPI_Send, o statusie STATUS. Następnie, w samej pętli, wywoływana jest metoda MPI_Probe, która sprawdza, czy dostarczono już dane od procesu głównego. Następnie zaczyna je odbierać asynchronicznie poprzez MPI_Irecv, po czym następuje bariera MPI_Wait. Po wykonaniu obliczeń, dane są odsyłane do procesu hosta poprzez asynchroniczną funkcję MPI_Isend oraz wysyłany jest status o gotowości do przetwarzania kolejnych paczek danych za pomocą wywołania asynchronicznej metody MPI_Isend. Na początku pętli while znajdują się dwie blokady, które czekają na wysłanie danych, oraz wysłanie statusu w przypadku, jeżeli transfer takich komunikatów został rozpoczęty. W razie odebrania w metodzie MPI_Probe danych o statusie FINISH, praca w procesie obliczeniowym jest kończona.

7.3. OpenMP

W porównaniu do wersji sekwencyjnej, realizacja programu w technologii OpenMP, jak to już było wspomniane przy opisie implementacji algorytmu Mandelbrota, opiera się głównie na

dyrektywach preprocesora. Zatem funkcje realizujące logikę aplikacji, czyli sortowanie oraz generowanie tablic, zostały pominięte w przedstawionym opisie.

Rozpoczynając od implementacji, można zauważyć deklaracje stałych, które są używane w dyrektywach offload, a które oznaczają odpowiednio, alokację pamięci bez jej zwolnienia, zwolnienie pamięci bez alokacji i ponowne jej użycie, bez zwolnienia oraz alokacji. Jest to oczywiście związane z obszarami danych deklarowanymi w dyrektywie offload.

```
#define ALLOC alloc_if(1) free_if(0)
#define FREE alloc_if(0) free_if(1)
#define REUSE alloc_if(0) free_if(0)
```

Kolejną, wartą podkreślenia rzeczą, jest dodatkowy parametr wejściowy numOfTransferredArrays, który odnosi się do liczby przetwarzanych tablic w ramach jednej iteracji programu.

Następnie można przytoczyć dyrektywy preprocesora otaczające funkcje, które są wywoływane po stronie koprocatora:

```
#pragma offload_attribute (push, target(mic))
{...}
#pragma offload_attribute (pop)
```

oznaczają one wygenerowanie kodu funkcji pod architekturę Intel MIC.

Przechodząc do fazy inicjalizacji programu, można zauważyć dyrektywę offload_transfer, która wykonuje tylko alokację obszaru danych, o rozmiarze równym wartości wyrażenia `arraySize * numOfTransferredArrays`, na urządzeniu Intel Xeon Phi. Warto w tym miejscu zauważyć różnicę w zastosowanym modelu, pomiędzy innymi technologiami. Mianowicie z braku możliwości [8], w używanej wersji kompilatora, wywołania wielu sekcji jednocześnie, tak aby, po każdym transferze móc zacząć kolejny wraz z obliczeniami na innych wątkach, zdecydowano się na jednorazowe przesyłanie takiej ilości danych, aby każdy uruchomiony wątek na koprocesorze musiał posortować jedną tablicę. Jest to spójne z założeniem, że żądanie jest przetwarzane tylko przez jedną jednostkę, jednak nie zachowujemy problemu tak częstej komunikacji, jak w przypadku innych technologii.

Opisując dalej dyrektywę otaczającą funkcję MergeOnMic(), należy zwrócić uwagę, że nie zostaje na początku wykonany transfer danych pomiędzy pamięcią, ze względu na użycie konstrukcji `array[0:0]`, która to oznacza kopiowanie, od indeksu 0, zerowej liczby elementów. Chodzi po prostu o alokację odpowiedniej ilości pamięci na koprocesorze.

```
#pragma offload_transfer target(mic)
in(array[0:0]:alloc(array[0:arraySize * numOfTransferredArrays]) ALLOC)
```

Analizując w następnej kolejności implementację funkcji MergeOnMic(), należy zwrócić uwagę na dyrektywę realizującą wykonanie obliczeń na koprocesorze. Można wyróżnić tutaj dwie rzeczy, które są dla niej charakterystyczne. Mianowicie jest to, po pierwsze, omówione w przypadku pozostałych algorytmów, ponowne użycie zaalokowanej pamięci na koprocesorze, co odbywa się za pomocą atrybutów REUSE. Po drugie, unikalne dla omawianej implementacji,

jest zastosowanie klauzuli `into`. Powoduje ona, w połączeniu z `in`, odwzorowanie przesuniętego o podaną ilość adresu na hoście na adres na koprocesorze. Dzięki temu zabiegowi możliwe jest odpowiednie skopiowanie danych z większej tablicy spod adresu, który to jest równy wartości wyrażenia `array + arrayCounter*arraySize`, pod właściwy w tablicy na Intel Xeon Phi. `into`, również w przypadku `out`, umożliwia odpowiednie odwzorowanie w drugą stronę.

```
#pragma offload target(mic:0)
in(array[arrayCounter*arraySize:arraySize*numOfTransferredArrays] :
into(array[0:arraySize*numOfTransferredArrays]) REUSE)
out(array[0:arraySize*numOfTransferredArrays] :
into(array[arrayCounter*arraySize:arraySize*numOfTransferredArrays]) REUSE)
```

Wewnątrz wyżej opisanej sekcji możemy zauważyć już tą, która rozdziela wykonywanie kodu na wątki poprzez następującą pragmę:

```
#pragma omp parallel for
```

Na końcu warto podkreślić, iż w opisywanej implementacji zastosowano małą liczbę dyrektyw, są one bardzo ciekawe i pokazują wiele cech i możliwości modelu `offload`.

7.4. OpenCL

Rozpatrując implementację algorytmu w ostatniej z technologii, OpenCL, należy podkreślić zmianę znaczenia ostatniego parametru wejściowego w porównaniu do pozostałych technologii. Mianowicie stanowi on informację dotyczącą nie ile procesów, bądź wątków zostanie utworzonych na koprocesorze, lecz liczbę utworzonych kolejek OpenCL, a zatem liczbę jednoczesnych utrzymywanych połączeń z kartą Intel Xeon Phi. Z kolei wynikiem programu jest linia tekstu zawierająca czasy trwania poszczególnych części programu i ma ona format identyczny z pozostałymi implementacjami równoległymi.

Przechodząc do implementacji rozwiązania rozpatrywanego modelu, należy zaznaczyć, że został wykorzystany, odmienny względem pozostałych technologii, model przetwarzania. Mianowicie jest tutaj wykorzystywany model asynchroniczny, który wspierany jest w znaczącym stopniu przez metody biblioteki OpenCL. Jednakże zanim zostanie omówiona jego realizacja, przedstawiona zostanie ogólna struktura programu.

Kod został podzielony na obszar inicjalizujący środowisko.

```
ClEnvironment PrepareEnvironment(int arrays, int size)
```

zostaje tu utworzony obiekt typu `Context` oraz obiekt typu `Kernel`, oraz kod, który zarządza kolejkami rozkazów i przetwarza żądania. Jego wywołanie znajduje się w procedurze

```
void MakeCalculations(ClEnvironment &env,int numOfArrays,int arraySize, int numOfRunners)
```

Ten obszar programu składa się z obiektu klasy `RequestProcessor`, który posiada kontener przechowujący instancje klasy `ClRunner`, które są obiektami odpowiadającymi za pojedyncze kolejki rozkazów. Wspomniana klasa `RequestProcessor` odpowiada za przyjmowanie żądań, a

następnie wybór kolejki, która może obecnie przyjmować zadania. Przetwarzanie ich rozpoczyna się w metodzie `RequestProcessor::StartProcessing()`

```
for (int i = 0; i < parameters.kernelParameters.arrayCounter; ++i) {  
    auto request = requestFetcher();  
    DispatchRequest(request);  
}
```

W tym miejscu należy nadmienić, że `requestFetcher` jest to instancja klasy `RequestFetcher`. Jest to funktor, który po wywołaniu zwraca kolejne przychodzące żądanie. Z kolei funkcja `RequestProcessor::DispatchRequest(int *request)` wybiera dostępny obiekt klasy `ClRunner` i zleca mu przetworzenie żądania (listing (7.7)).

Listing 7.7. Zlecenie przetwarzania żądania

```
void RequestProcessor::DispatchRequest(int *request) {  
    auto user_data = new UserDataForCallback;  
    user_data->statuses = statuses;  
    int selectedRunnerIndex = getAvailableRunner();  
    user_data->runnerIndex = selectedRunnerIndex;  
    ClRunner& selectedRunner = runners[selectedRunnerIndex];  
    selectedRunner.WriteTo(request, parameters.kernelParameters.arraySize);  
    selectedRunner(parameters.environment.kernel, [(cl_event e, cl_int i, void*  
user_data) {  
        ((UserDataForCallback*)((RunnerArgs*)user_data)->PassedData))  
        ->statuses[((UserDataForCallback*)((RunnerArgs*)user_data)->PassedData))  
        ->runnerIndex]=true;  
        _mm_free(((RunnerArgs*)user_data)->OutputData);  
        delete (((RunnerArgs*)user_data)->PassedData);  
    }, user_data);  
}
```

Analizując kolejne linie przytoczonej metody (listing 7.7.), należy zwrócić uwagę na jej przebieg. Przede wszystkim wywołana jest metoda `getAvailableRunner()` w celu pobrania indeksu wolnej kolejki. Ze względu na testowanie wydajności obliczeń na karcie Intel Xeon Phi, zdecydowano się na prostą implementację tej metody, która ciągle sprawdza, czy któryś z obiektów `ClRunner` zgłosił gotowość poprzez ustawienie odpowiadającej jej zmiennej logicznej na wartość `prawda` (listing 7.8.).

Listing 7.8. Pobranie wolnej kolejki

```
int RequestProcessor::getAvailableRunner() const {
    int selected = -1;
    while(selected == -1) {
        selected = checkForAvailability();
    }
    return selected;
}

int RequestProcessor::checkForAvailability() const {
    for (int i = 0; i < parameters.numberOfRunners; ++i) {
        if(statuses[i]) {
            statuses[i] = false;
            return i;
        }
    }
    return -1;
}
```

Ponadto następuje zlecenie zapisu do bufora wybranego obiektu oraz zlecenie przeprowadzenia obliczeń. Ważnym argumentem wywoływanej funkcji operator() klasy CIRunner, jest funkcja anonimowa, która pełni rolę odwołania. Zostanie one uruchomione po wykonaniu obliczeń oraz pobraniu danych z urządzenia do pamięci komputera hosta. W funkcji tej, w przypadku rzeczywistej aplikacji, nastąpiłoby odesłanie danych do klienta. Aby przekazać dane użytkownika do wspomnianej funkcji, zostaje, jako ostatni argument, przekazany adres do obszaru pamięci, w którym znajdują się owe wartości.

Aby zrozumieć dokładnie przepływ sterowania w zrealizowanej aplikacji, należy przeanalizować kod klasy CIRunner. Aby zapewnić niezależność od pozostałych kolejek, klasa ta, jako jej pola, przechowuje obiekty bufora

```
cl::CommandQueue CommandQueue;
cl::Context Context;
cl::Buffer LocalMemoryBuffer;
cl::Buffer OutBuffer;
cl::Buffer InBuffer;
```

Inicjalizacja opisanych obiektów została tutaj pominięta, ze względu na implementację zbliżoną do pozostałych programów wytworzonych w tej technologii (listing 7.9.).

Listing 7.9. Kod funkcji `ClRunner::operator()`

```
void ClRunner::operator()(cl::make_kernel<cl::Buffer &, cl::Buffer &, cl::Buffer &>
Kernel, void (CL_CALLBACK *pfn_notify)(cl_event event, cl_int command_exec_status, void *
user_data), void *pVoid) {
    auto output = (int*)_mm_malloc(sizeof(int) * Parameters.arraySize, ALLOC_ALIGN);
    auto callbackArgs = new RunnerArgs();
    [...]
    cl::Event event;
    event = Kernel(cl::EnqueueArgs(CommandQueue, previousCommandEvent, cl::NDRange(1),
cl::NDRange(1)), LocalMemoryBuffer, InBuffer, OutBuffer);
    event.setCallback(CL_SUCCESS, [] (cl_event event_in, cl_int command_exec_status, void *
user_data) {
        cl::Event event;
        ((RunnerArgs*)user_data)->CommandQueue
        ->enqueueReadBuffer(*(((RunnerArgs*)user_data)->OutputBuffer), CL_FALSE, 0,
        (((RunnerArgs*)user_data)->ElementsNumber)*sizeof(int),
        ((RunnerArgs*)user_data)->OutputData, NULL, &event);
        event.setCallback(CL_SUCCESS, [] (cl_event event, cl_int command_exec_status, void *
user_data) {
            ((RunnerArgs*)user_data)->Callback(event, command_exec_status, user_data);
            delete user_data;
        }, user_data);
    }, callbackArgs);
}
```

Należy zwrócić uwagę na następujące dwie kwestie. Po pierwsze obliczenia wywoływane są na siatce obliczeniowej o rozmiarze 1, przy rozmiarze lokalnym mającym taką samą wartość. Nie jest to optymalne rozwiązanie, lecz zapewnia możliwość jednoczesnego uruchomienia wielu kerneli z poziomu różnych kolejek. Drugą rzeczą jest fakt, iż wspomniane wcześniej odwołanie związane jest w pośredni sposób, za pomocą funkcji `event.setCallback()`, z obiektem zdarzenia, które skojarzone jest z wykonywanymi obliczeniami przez kernel. Odwołanie to zostanie uruchomione asynchronicznie, względem głównego przepływu programu, po prawidłowym wykonaniu kernela. Tutaj należy nadmienić, że konieczne było zastosowanie podobnego rozwiązania we właściwym odwołaniu na zdarzenie kernela, gdzie konieczne było ponowne, asynchroniczne pobranie danych z urządzenia i dopiero wówczas wywołanie funkcji przekazanej do metody `operator()`.

W kontekście kodu wykonującego przetwarzanie żądania, który jest kodem kernela OpenCL, należy nadmienić dwie rzeczy. Po pierwsze, ze względu na brak wsparcia, w wersji OpenCL 1.2, wywoływania rekursywnego funkcji, została zaimplementowana wersja iteracyjna algorytmu sortującego przez scalanie. Drugą ważną rzeczą był fakt, iż przez brak możliwości dynamicznej alokacji pamięci, konieczne było przekazanie, jako argumentu kernela, nadmiarowo zaalokowanego obszaru danych, w postaci wskaźnika na pamięć globalną `localMemory`,

8. OPIS IMPLEMENTACJI ALGORYTMU GENETYCZNEGO

Wśród stworzonych programów jest aplikacja tworząca obrazy z wykorzystaniem algorytmu genetycznego. Jej dokładny opis implementacyjny został przedstawiony w tym rozdziale.

8.1. Szeregowy

Implementacja szeregowy polegała na stworzeniu aplikacji generującej obraz podobny do rysunku wejściowego za pomocą algorytmu genetycznego, gdzie obraz wynikowy miał być zbudowany z samych prostokątów.

Jako argumenty programu przyjęto kolejno rozmiar populacji, liczbę generacji, liczbę osobników w elicie oraz ścieżkę do pliku wejściowego. Program został zaimplementowany w ten sposób, że składa się z obiektów odpowiadających za różne aspekty algorytmu, zatem plik wejściowy programu zawiera jedynie stworzenie instancji głównej klasy `GenericAlgorithm`, która to będzie odpowiadać za wywołania kolejnych faz algorytmu.

```
GeneticAlgorithm *geneticAlgorithm = new GeneticAlgorithm(population, generation, elite,
imageFile);
```

Następnie zostaje wywołana funkcja odpowiedzialna za wykonanie zadanej liczby generacji algorytmu.

```
geneticAlgorithm->Calculate();
```

Przechodząc do implementacji, wspomnianej wcześniej, klasy `GeneticAlgorithm`, należy zauważyć, że ze względu na wykorzystanie biblioteki `Cxxtest`, odpowiadającej za testy, konieczne było oznaczenie pól oraz funkcji prywatnych atrybutem `protected`, co było spowodowane chęcią przetestowania tych metod. W przypadku, gdyby pozostały one oznaczone modyfikatorem dostępu `private`, nie byłoby możliwości przeprowadzenia owych testów.

Analizując kod klasy `GeneticAlgorithm` warto zwrócić uwagę na deklaracje wartości stałych, które znajdują się na początku pliku nagłówkowego `GeneticAlgorithm.h`:

```
#define MAX_NUMBER_OF_RECTANGLES 500
#define ALLOC_ALIGN 64
#define ALLOC_TRANSFER_ALIGN 4096
```

Pierwsza z nich oznacza liczbę prostokątów, które wchodzi w skład pojedynczego osobnika. Z kolei następne dwie wartości są związane z drugim argumentem funkcji `_mm_malloc`, której deklaracja znajduje się w pliku nagłówkowym `mm_malloc.h`. Pozwala ona na alokację pamięci z wyrównaniem adresu początkowego do zadanej liczby bajtów. Mianowicie zgodnie z ustaleniami firmy Intel, co do programowania na Intel Xeon Phi, wszystkie dane powinny być wyrównane do 64 bajtów, natomiast tablice, które są przenoszone pomiędzy urządzeniami, powinny być wyrównane do 4096 bajtów w celu wykorzystania DMA.

Przechodząc do implementacji samej funkcji `Calculate()`, która wykonuje wszystkie obliczenia, można zaobserwować działanie wszystkich funkcji oraz obiektów. Zaczynając analizę

od początku funkcji Calculate(), widać w niej wywołanie prywatnej metody generateRectangles(), która odpowiada za wygenerowanie n obrazów złożonych, jak to zostało wcześniej wspomniane, z 500 losowo rozstawionych prostokątów. Z uwagi na optymalizację rozwiązania, zdecydowano się na trzymanie w pamięci jedynie listy prostokątów, które są malowane na obrazach. Zrezygnowano natomiast z przechowywania w pamięci listy wszystkich wygenerowanych obrazów, zamiast tego w pamięci znajduje się jeden obraz.

W tym miejscu należy przytoczyć realizację struktury odpowiedzialnej za reprezentację pojedynczego prostokąta w programie

```
struct Rectangle
{
    Rectangle();
    Rectangle(Point leftUp, Point rightDown, Color color);

    Point rightDown, leftUp;
    Color color;
};
```

gdzie typ Point jest zaimplementowany następująco:

```
struct Point
{
    unsigned int x, y;
};
```

Jest to reprezentacja optymalna, gdyż do opisu prostokąta wystarczają współrzędne dwóch przeciwległych wierzchołków, przez co wszystkie prostokąty zajmują łącznie wartość, opisaną przez zależność (8.1)

$$(4B + 4B) * 2 * MAX\ NUMBER\ OF\ RECTANGLES * Population \quad (8.1)$$

gdzie:

m – maksymalna liczba prostokątów

p – wartość populacji

W następnych liniach kodu można zauważyć pętlę, która stanowi główną część programu, a w jej wnętrzu są wykonywane fazy oceny, mutacja oraz krzyżowania. Można zauważyć, że pętla odpowiada za kontrolowanie wykonywania wszystkich obliczeń w zależności od wcześniej zadeklarowanej liczby generacji.

```
for (GenerationsLeft = generation; GenerationsLeft > 0; --GenerationsLeft){...}
```

Wewnątrz pętli for znajdują się czasomierze odpowiedzialne za liczenie czasów trwania pojedynczej generacji oraz oceny obrazów. Następnie zostają wykonane wywołania funkcji odpowiedzialnych za ocenę osobników populacji, co wykonywane jest na zasadzie wygenerowania obrazu na podstawie prostokątów danego osobnika, po czym utworzony obraz porównywany jest z wejściowym. Należy w tym miejscu zauważyć, iż wszystkie obliczenia przeprowadzone są przy pomocy obszaru pamięci odpowiadającemu wyłącznie jednemu obrazowi, co jest kluczowe dla efektywnego zarządzania pamięcią, zwłaszcza w przypadku

implementacji algorytmu w wersji równoległej z uwagi na zaledwie 8 GB pamięci dostępnej na urządzeniu Intel Xeon Phi 5120. Wracając do analizy kodu, funkcja odpowiedzialna za wyliczanie ocen osobników opiera się na wyczyszczeniu obszaru pamięci odpowiedzialnego za reprezentację obrazu, następnie namalowaniu na nim prostokątów, które należały do danego osobnika, i ostatecznie na porównaniu obrazu wejściowego z tym wygenerowanym.

Listing 8.1. Porównanie obrazów

```
double GeneticAlgorithm::compare(Image *first, Image *second) {
    double maxDiff = this->width * this->height * validsqrt;
    auto diff = 0.0;
    for (int y = this->height - 1; y >= 0; --y) {
        for (int x = this->width - 1; x >= 0; --x) {
            auto firstPixel = first->Area[y * width + x];
            auto secondPixel = second->Area[y * width + x];
            diff += sqrtl(
                (firstPixel.r - secondPixel.r) * (firstPixel.r - secondPixel.r)
                + (firstPixel.g - secondPixel.g) * (firstPixel.g - secondPixel.g)
                + (firstPixel.b - secondPixel.b) * (firstPixel.b - secondPixel.b));
        }
    }
    return (maxDiff - diff) / maxDiff;
}
```

Na listingu 8.1. można zauważyć, że porównywane były wszystkie wartości pikseli, a na koniec była zwracana wartość porównania, która zawsze była wyrażana jako liczba z zakresu $<0.0; 1.0>$.

Po wykonaniu porównań obrazów następowało zatrzymanie czasomierza odpowiedzialnego za liczenie porównania, a następnie zostało wykonane sortowanie otrzymanych wyników w celu przygotowania ich do późniejszej obróbki przez algorytm. Sam kod klasy porównującej znajduje się poniżej:

```
struct Comparison
{
    Comparison(int i, double d);
    static bool CompareComparison (const Comparison& first, const Comparison& second);

    unsigned int index;
    double value;
};
```

przy czym implementacja funkcji porównującej wygląda następująco:

```
bool Comparison::CompareComparison (const Comparison& first, const Comparison& second)
{
    return first.value > second.value;
}
```

Po posortowaniu wyników następowało uruchomienie kolejnego czasomierza, który odpowiadał za pomiar czasu mutacji i krzyżowania obrazów.

Mutacje, czyli funkcja `mutateElite()` (listing 8.2.), wywołana zaraz po uruchomieniu wspomnianego wyżej czasomierza, były odpowiedzialne za stworzenie nowych osobników na podstawie najlepiej ocenionych z obecnej populacji, tzw. elity, której liczebność była podawana na wejściu programu. Przechodząc do metody `mutateElite()` można zauważyć, że wybierała ona dwa różne obrazy z elity i na ich podstawie wywoływała metodę `mutation(...)`

Listing 8.2. Mutacja elity

```
void GeneticAlgorithm::mutateElite() {
    for (int i = this->population - 1; i >= this->elite; --i) {
        unsigned long i1 = rand() % this->elite;
        int i2 = i % this->elite;
        mutation(this->imagesRectangles +
            comparisonResults[i].index*MAX_NUMBER_OF_RECTANGLES,
            this->imagesRectangles +
            comparisonResults[i2].index*MAX_NUMBER_OF_RECTANGLES,
            this->imagesRectangles +
            comparisonResults[i1].index*MAX_NUMBER_OF_RECTANGLES);
    }
}
```

Sama metoda `mutation` jest krótką funkcją, która, w pętli `for`, zapełnia tablicę osobnika wynikowego losowymi prostokątami pochodzącymi od rodziców oraz wykonującą, z niskim stopniem prawdopodobieństwa, losową mutację wybranych prostokątów.

```
for (int i = 0; i < MAX_NUMBER_OF_RECTANGLES; ++i) {...}
```

Po zakończeniu głównej pętli w funkcji `Calculate()`, która wykonywała się tyle razy, ile zostało zadeklarowane na wejściu programu poprzez ustawienie argumentu liczby generacji, następowało zatrzymanie wszystkich działających jeszcze czasomierzy oraz wybranie najlepszego wyniku, co w tym przypadku było pobraniem listy prostokątów osobnika o najwyższej ocenie i namalowaniu ich na obraz wyjściowy, a następnie zapisywaniu na dysku jako plik w formacie PNG i nazwie `result`. Po wyjściu z funkcji `Calculate()`, następowało zwolnienie zasobów poprzez wywołanie destruktoru obiektu `GeneticAlgorithm`.

8.2. MPI

W przypadku implementacji w technologii MPI w kontekście parametrów wejściowych oraz wyników programu, w porównaniu do wersji szeregowej programu, jedyną zmianą, która została wykonana, jest zmiana formatu wyjściowych statystyk czasu działania programu.

```
MPI,16,1000,250,601540804000,119822072000,119807136000,15942000,601540975000,601800279000
```

Są to wartości odpowiadające kolejno technologii, w której została wykonana dana implementacja, ilość procesów stworzonych po stronie koprocatora, rozmiar obrazu

wejściowego, rozmiar populacji, całkowity czas wykonywania algorytmu, czas pojedynczej generacji, czas pojedynczego wyliczania ocen populacji, czas pojedynczej fazy mutacji i krzyżowania populacji, czas działania programu bez czasu inicjalizacji środowiska oraz całkowity czas programu.

Przechodząc do wytworzonego kodu, należy rozpocząć omówienie programu od ogólnej struktury programu. Dzięki pobraniu na samym początku dwóch parametrów ze środowiska MPI, mianowicie rozmiaru komunikatora globalnego oraz tzw. wartości, która jest identyfikatorem procesu w obrębie komunikatora

```
MPI_Comm_size(MPI_COMM_WORLD, &numberOfProcesses);  
MPI_Comm_rank(MPI_COMM_WORLD, &commRank);
```

możliwy jest podział działania programu na część wykonywaną przez proces główny typu master, oraz procesy typu slave działające na koprocesorze. Podział ten realizowany jest poprzez porównanie pobranej wartości identyfikatora z wartością 0, która odpowiada głównemu procesowi.

```
if(commRank == 0)
```

Kolejną konstrukcją wartą omówienia, jest wykonane w trakcie inicjalizacji pracy algorytmu, wywołanie funkcji komunikacji zbiorczej, mianowicie jest to realizowane przez następujący kod.

```
MPI_Bcast(&height, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Dzięki tej funkcji przekazywane zostają parametry pracy z procesu głównego do pozostałych.

Najważniejszą strukturą, która została wykonana, jest część kodu odpowiadająca za dystrybucję pracy pomiędzy procesy typu slave z poziomu procesu głównego, co zaimplementowane zostało w metodzie prezentowanej poniżej.

```
DistributeCalculations(int starting, int ending, std::function<int(int)> indexProvider)
```

Odpowiada ona za przydzielenie zadań do konkretnych procesów. Liczba tych zadań determinowana jest poprzez dwa pierwsze argumenty, a zatem parametr starting wskazuje od którego indeksu, w tablicy obrazów, należy rozpocząć rozsyłanie danych, aż do osiągnięcia wartości parametru ending. W trakcie wysyłania danych, obecny indeks jest konwertowany za pomocą przekazanej do tej metody funkcji anonimowej, która zwraca odpowiedni adres, pod którym znajdują się dane do wysłania.

```
MPI_Isend(imagesRectangles + indexProvider(ending + j-1) * MAX_NUMBER_OF_RECTANGLES,  
MAX_NUMBER_OF_RECTANGLES, mpi_rectangle_type, j, DATA, MPI_COMM_WORLD, &request);
```

W kontekście dystrybucji zadań (listing 8.3.) należy podkreślić fakt, iż zrealizowane jest to w sposób gwarantujący, że każdy z procesów slave będzie miał paczkę danych do liczenia, a w dodatku obliczenia oraz komunikacja będą się nakładały.

Listing 8.3. Dystrybucja paczek danych pomiędzy procesy slave

```
for (int j = std::min(commSize-1, totalNumberOfIterations); j > 0 ; --j) {
    [...]
    MPI_Isend(imagesRectangles + indexProvider(ending + j-1) * MAX_NUMBER_OF_RECTANGLES,
    MAX_NUMBER_OF_RECTANGLES, mpi_rectangle_type, j, DATA, MPI_COMM_WORLD, &request);
}

for (int j = std::min(commSize-1, totalNumberOfIterations - (commSize-1)); j > 0 ; --j) {
    [...]
    MPI_Isend(imagesRectangles + indexProvider(ending + commSize-1+j-1) *
    MAX_NUMBER_OF_RECTANGLES, MAX_NUMBER_OF_RECTANGLES, mpi_rectangle_type, j, DATA,
    MPI_COMM_WORLD, &request);
}

for (int i = starting; i >= ending+2*(commSize-1); --i) {
    MPI_Recv(&value, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    [...]
    MPI_Isend(imagesRectangles + indexProvider(i) * MAX_NUMBER_OF_RECTANGLES,
    MAX_NUMBER_OF_RECTANGLES, mpi_rectangle_type, status.MPI_SOURCE, DATA, MPI_COMM_WORLD,
    &request);
    [...]
}

for (int i = std::min(2*(commSize-1), totalNumberOfIterations); i > 0; --i) {
    MPI_Recv(&value, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    [...]
}
}
```

Jak można zauważyć, na początku, do każdego procesu wysyłane jest asynchronicznie po jednym osobniku, a następnie wysyłana jest kolejna partia danych tak, aby w momencie zakończenia odbierania pierwszej paczki, proces slave mógł od razu rozpocząć pobieranie kolejnych danych. Po wykonaniu wspomnianych pierwszych dwóch pętli następuje faza dalszej dystrybucji, lecz tym razem biorąc pod uwagę obciążenie poszczególnych procesów. Mianowicie zanim do procesu zostanie wysłana kolejna porcja danych, musi on najpierw przysłać wynik przetworzenia poprzedniej paczki. Na koniec tej fazy zostają zebrane wyniki ze wszystkich procesów typu slave.

W przytoczonym kodzie można zauważyć, że do transmisji danych, został wykorzystany ich własny typ (listing 8.4.), mianowicie jest to `mpi_rectangle_type`, który został utworzony za pomocą metody `InitCustomTypes()`, która inicjalizuje typy danych użytkownika.

Listing 8.4. Definiowanie własnego typu

```

const int nitensRectangle=3;
int      blocklengthsRectangle[3] = {1,1,1};
MPI_Datatype typesRectangle[3] = {mpi_point_type, mpi_point_type, mpi_color_type};
MPI_Aint    offsetsRectangle[3];
offsetsRectangle[0] = offsetof(Rectangle, rightDown);
offsetsRectangle[1] = offsetof(Rectangle, leftUp);
offsetsRectangle[2] = offsetof(Rectangle, color);
MPI_Type_create_struct(nitensRectangle, blocklengthsRectangle, offsetsRectangle,
typesRectangle, &mpi_rectangle_type);
MPI_Type_commit(&mpi_rectangle_type);

```

Jak widać (listing 8.4.), aby utworzyć własny typ, konieczne jest podanie z jakich typów prostszych będzie się on składał oraz w jakiej liczbie będą one składowane. Następnie podając przesunięcia położenia danych pól, można już utworzyć własny typ, co realizowane jest przez dwie ostatnie przytoczone wywołane procedury.

8.3. OpenMP

Chcąc dokonać analizy implementacji algorytmu genetycznego w kolejnej technologii równoległej, mianowicie OpenMP, należy, ze względu na wysoki stopień podobieństwa do kodu szeregowego w kontekście właściwych obliczeń, skupić się głównie na zmianach. W większości polegają one na wykonanych konstrukcjach realizowanych za pomocą dyrektyw preprocesora, jak również na mniejszych modyfikacjach związanych ze zrównolegleniem algorytmu.

Jednak należy zaznaczyć, iż do argumentów początkowych programu dodano parametr, który określa liczbę wątków tworzonych w grupie wątków na koprocessorze Intel Xeon Phi. Jest to parametr wykorzystany tylko przy wypisywaniu statystyk związanych z działaniem programu, gdyż użyta liczba wątków określana jest przez zmienną środowiskową OPM_NUM_THREADS. Format tych statystyk jest następujący:

```
OpenMP,16,1000,250,602849529000,120260396000,119280333000,980044000,602853978000,602853978000
```

Wartości oddzielone przecinkami określają kolejno nazwę technologii, w której został wykonany dany program, liczbę zadanych wątków, rozmiar zadanego obrazu wejściowego, rozmiar populacji, całkowity czas działania algorytmu, czas trwania pojedynczej generacji, czas trwania pojedynczego liczenia ocen osobników w danej generacji, czas pojedynczej mutacji oraz dla spójności z wynikami pozostałych implementacji, dwukrotnie czas działania całego programu.

Przechodząc do zrównoleglenia obliczeń, należy opisać sposób w jaki zostało to zaprojektowane. Mianowicie przyjęto, że obliczenia zostaną podzielone odpowiednio dla funkcji liczącej ocenę na poziomie pojedynczego osobnika, czyli wyliczenie jakości danego obrazu nie jest już podzielne. Natomiast mutacja i krzyżowanie realizowane są na poziomie wybranej pary osobników, dla których zachodzi krzyżowanie i jednocześnie mutacja dla osobnika wynikowego. Z tego faktu przyjęto, iż każdy z wątków potrzebuje tylko i wyłącznie jednego obszaru, na którym

będzie malował prostokąty osobnika, a następnie wyliczał na podstawie wygenerowanego obrazu jego jakość. Aby zoptymalizować realizację owego wymagania dokonano tylko pojedynczej alokacji pamięci na początku działania całego algorytmu (listing 8.5.).

Listing 8.5. Alokacja pamięci na koprocesorze

```
int numberOfThreadsOnPhi;  
#pragma offload target(mic) mandatory  
#pragma omp parallel  
{  
    #pragma omp single  
    numberOfThreadsOnPhi = omp_get_num_threads();  
}  
  
#pragma offload mandatory target(mic)  
in(outputImagesPhi[0:0]:alloc(outputImagesPhi[0:numberOfThreadsOnPhi]) alloc_if(1)  
free_if(0) align(64))  
in(comparisonResultsPhi[0:0]:alloc(comparisonResultsPhi[0:population]) alloc_if(1)  
free_if(0) align(64))  
in(imagesRectanglesPhi[0:sizeOfRectangleTablePhi]:alloc(imagesRectanglesPhi[0:sizeOfRectangleTablePhi]) alloc_if(1) free_if(0) align(64))  
in(nativeImagePhi[0:1]:alloc(nativeImagePhi[0:1]) alloc_if(1) free_if(0) align(64))  
in(nativeImageAreaPhi[0:width*height]:alloc(nativeImageAreaPhi[0:width*height])  
alloc_if(1) free_if(0) align(64))  
{  
    nativeImagePhi->Area = nativeImageAreaPhi;  
    for (int i = 0; i < numberOfThreadsOnPhi; ++i) {  
        Image::InitImage(outputImagesPhi + i, height, width);  
    }  
}
```

Należy zauważyć wykorzystanie w tym przypadku (listing 8.5.) dwóch osobnych bloków, które zostały odesłane do wykonania na koprocesorze. Pierwszy z nich ma na celu pobranie aktualnej liczby wątków działających w obrębie grupy na koprocesorze. Posiadając już tę informację, można przystąpić do alokacji pamięci, co zostało podzielone na dwa etapy. Struktury, które posiadają reprezentację ciągłą, zostały zaalokowane za pomocą dyrektywy offload. Pojedynczym zarezerwowaniu pamięci może być następująca linia

```
in(outputImagesPhi[0:0]:alloc(outputImagesPhi[0:numberOfThreadsOnPhi]) alloc_if(1) free_if(0)  
align(64))
```

Oznacza ona, że wystąpiło żądanie alokacji pamięci o rozmiarze podanym jako wartość zmiennej numberOfThreadsOnPhi, przy założeniu, że alokujemy od indeksu 0. Dzięki pierwszej zaobserwowanej konstrukcji [0:0] zablokowano kopiowanie zawartości obszaru pamięci pomiędzy hostem, a koprocesorem. Następnie zostały zadane parametry alloc_if(1) oraz free_if(0), które powodują, że obszar ten zostanie zaalokowany, lecz nie będzie zwolniony po zakończeniu działania danej dyrektywy offload. Zastosowano również atrybut align(64) ze

względem na optymalizację w kierunku wektoryzacji na urządzeniu Intel Xeon Phi, które do realizacji instrukcji wektorowych wykorzystuje rejestry 64 bajtowe.

Ze względu na nieliniowy charakter struktury typu Image, konieczne była najpierw alokacja tablicy samych struktur opakowujących ten typ za pomocą dyrektywy, a następnie, w kodzie wykonanym na koprocesorze, wykonana została inicjalizacja każdej z tych struktur (listing 8.5.). Należy jeszcze wspomnieć, iż zwolnienie wcześniej zaalokowanych obszarów została wykonana w analogiczny sposób.

Przechodząc do analizy konstrukcji realizującej zrównoleglenie i uruchomienie na koprocesorze, należy początkowo przytoczyć następujących fragment kodu pokazany na listingu 8.6.

Listing 8.6. Wykonanie obliczeń na koprocesorze

```
#pragma offload mandatory target(mic) in(outputImagesPhi[0:0]:alloc(outputImagesPhi[0:0])
alloc_if(0) free_if(0))
inout(comparisonResultsPhi[0:populationPhi]:alloc(comparisonResultsPhi[0:0]) alloc_if(0)
free_if(0))
in(imagesRectanglesPhi[0:sizeOfRectangleTablePhi]:alloc(imagesRectanglesPhi[0:0])
alloc_if(0) free_if(0))
in(nativeImagePhi[0:0]:alloc(nativeImagePhi[0:0]) alloc_if(0) free_if(0)) \
in(nativeImageAreaPhi[0:0]:alloc(nativeImageAreaPhi[0:0]) alloc_if(0) free_if(0))
#pragma omp parallel private(i, th_id)
{
    th_id = omp_get_thread_num();
    #pragma omp for
    for (i = populationPhi - 1; i >= 0; --i) {
        comparisonResultsPhi[i] = Comparison(i, CalculateValueOfImage(
            imagesRectanglesPhi + i * MAX_NUMBER_OF_RECTANGLES, width, height,
            outputImagesPhi+th_id, nativeImagePhi));
    }
}
```

Można wyszczególnić tutaj parę najważniejszych konstrukcji. Po pierwsze, aby wykorzystać wcześniej zaalokowane obszary danych, należało użyć następującej części dyrektywy.

```
in(nativeImagePhi[0:0]:alloc(nativeImagePhi[0:0]) alloc_if(0) free_if(0))
```

Powoduje ona, iż nie zostanie wykonana żadna alokacja, ani transfer danych pomiędzy danymi obszarami pamięci. Jednakże konieczne było zastosowanie tutaj atrybutu in w celu wykorzystania mechanizmów biblioteki Offload. Mianowicie dzięki tej konstrukcji, wartość wskaźnika po stronie hosta, zostaje zamieniona na adres wcześniej zaalokowanego obszaru danych, co wykonane jest dzięki wewnętrznej implementacji wykorzystanych konstrukcji. Drugą trudnością z wykorzystaniem dyrektywy offload w kontekście ponownego użycia obszaru danych, a która została rozwiązana w realizowanym projekcie, była niemożliwość zastosowania pól klasy jako argumentów w obrębie dyrektywy offload. Korzystając z nabytej wiedzy o działaniu tej

konstrukcji, została podjęta decyzja o tworzeniu zmiennych lokalnych, które stanowią kopię wartości pól klasy, a które też mogą zostać wykorzystane jako argumenty dyrektywy, np:

```
auto elitePhi = this->elite;
```

Ostatnim szczegółem implementacyjnym, o którym należy wspomnieć jest dyrektywa `omp parallel for`, odpowiedzialna za zrównoleglenie kodu, a która jest wykonana na koprocesorze, dzięki zastosowanej poprzedzającej ją dyrektywie `offload`. Została ona rozłączona na dwie oddzielne dyrektywy `omp parallel` oraz `omp for` w celu pobrania dla każdego wątku jego numeru. Podział pracy pomiędzy wątki został wykonany w domyślny sposób.

8.4. OpenCL

W odróżnieniu od implementacji szeregowej, w przedstawionym poniżej opisie nie zostały opisane funkcje i metody, które nie zostały zmienione. Zaprezentowano jedynie cechy i metody, charakterystyczne dla implementacji algorytmu w technologii OpenCL.

Wejście programu wygląda analogicznie do wersji szeregowej, dodany został jednak kolejny argument, który określa liczbę wątków, która ma odzwierciedlenie w rozmiarze globalnym.

Pierwszą rzeczą, jaką należy opisać, jest pobieranie urządzeń i ustawianie całego kontekstu dla algorytmu, co wiąże się z późniejszym uruchomieniem jąder obliczeniowych na urządzeniach wybranych właśnie w tym etapie programu. Samo jego ustawienie, jak i ściągnięcie urządzeń, zostało wykonane w sposób analogiczny do implementacji algorytmu Mandelbrota w technologii OpenCL.

Jednak w tym miejscu warto opisać definicje konkretnych kerneli, oraz to, za co odpowiadają. Jak można zauważyć po nazwie, pierwsze jądro obliczeniowe odpowiada za obliczenie wartości obrazów, drugie za mutację elity. Z uwagi na specyficzny sposób przesyłania struktur do kodu jądra obliczeniowego, zdecydowano się zadeklarować, dla obu kerneli, wszystkie potrzebne struktury w jego wnętrzu. Zatem w przypadku kernela odpowiedzialnego za liczenie wartości obrazów, na początku pliku znajdują się deklaracje struktur danych, które są analogiczne do tych z implementacji szeregowej, jednakże warto wymienić, że są wśród nich `Point`, `Color`, `Rectangle`, `Comparison`, `Pixel`, `Image`. Strukturą dodatkową, która nie wystąpiła w implementacji szeregowej, jest struktura posiadająca listę wszystkich parametrów.

```
typedef struct type_parameters
{
    int generation, population, width, height, elite, numberOfRectangles;
} Parameters;
```

Po deklaracji wszystkich struktur następuje definicja jądra obliczeniowego oznaczona słowem kluczowym `__kernel` znajdującym się przed typem zwracanym przez funkcję oraz jej nazwą. Parametrami przekazywanymi do funkcji są parametry globalne, takie jak elita, populacja, lista prostokątów nanoszonych na obrazach, lista z wartościami porównania, obraz wejściowy, oraz lista obrazów, na których będą malowane prostokąty. Dla każdego wątku technologii OpenCL przekazywany jest oddzielny obszar pamięci, co zostało zrealizowane, aby uniknąć

miejsz synchronizacji typu bariera, które są niezalecane przez firmę Intel podczas tworzenia oprogramowania na urządzenia Intel Xeon Phi w opisywanej technologii. Sama deklaracja funkcji jądra obliczeniowego wygląda następująco:

```
__kernel void ValueOfImage(__global Rectangle* rectangles, __global Parameters* parameters,
__global Comparison* comparisonTable, __global Pixel* validImage, __global Pixel*
imageToWriteOn)
```

Skupiając się na opisie kernela (listing 8.7.), można zauważyć, że następuje w nim pobranie identyfikatora globalnego oraz rozmiaru globalnego, który będzie potem użyty w pętli wykonującej porównania, z uwagi na to, że w pewnych przypadkach pojedyncza jednostka obliczeniowa będzie wykonywała więcej, niż jedno porównanie obrazów. Po pobraniu zmiennych związanych ze środowiskiem, następuje wyliczenie liczby obrazów, które zostaną porównane.

Listing 8.7. Ustawienie parametrów wejściowych jądra obliczeniowego

```
int globalId = get_global_id(0);
int globalSize = get_global_size(0);
int rectanglesPerThread = 1;
float validsqr = sqrt(255.0 * 255.0 * 3.0);
float maxDiff = (float)parameters->width * (float)parameters->height * validsqr;
float diff = 0.0;
if(parameters->population >= globalSize)
    rectanglesPerThread = parameters->population / (globalSize -1);
int imageSize = parameters->width * parameters->height;
int downLimit = globalId * rectanglesPerThread, upLimit=(globalId+ 1)*rectanglesPerThread;
```

Można zauważyć (listing 8.7.) używanie zmiennych typu float, które są mniej precyzyjne od zmiennych typu double, aczkolwiek zajmują mniej miejsca w pamięci, co w przypadku tego algorytmu było kolejną kluczową optymalizacją

Analizując dalszą część kodu kernela (listing 8.7), można zauważyć, że w zależności od obrazów liczonych na nim oraz mając na uwadze, że nie powinno się liczyć wartości porównania dla obrazów, których identyfikator jest większy od liczby populacji.

Listing 8.8. Fragment kodu jadra obliczeniowego odpowiedzialny za malowanie prostokątów na obrazie

```
Rectangle rectangle;
int height, width, baseY;
for (int i = parameters->numberOfRectangles; i >= 0; --i) {
    rectangle = rectangles[j * parameters->numberOfRectangles + i];
    height = abs(rectangle.rightDown.y - rectangle.leftUp.y);
    width = abs(rectangle.rightDown.x - rectangle.leftUp.x);
    for (int y = height; y >= 0; --y) {
        baseY = (rectangle.leftUp.y + y) * parameters->width;
        for (int x = width; x >= 0; --x) {
            int linearIndex = baseY + rectangle.leftUp.x + x;
            imageToWriteOn[...].r =
                (imageToWriteOn[...].r * imageToWriteOn[...].drawed + rectangle.color.r) /
                (imageToWriteOn[...].drawed + 1);
            imageToWriteOn[...].g =
                (imageToWriteOn[...].g * imageToWriteOn[...].drawed + rectangle.color.g) /
                (imageToWriteOn[...].drawed + 1);
            imageToWriteOn[...].b =
                (imageToWriteOn[...].b * imageToWriteOn[...].drawed + rectangle.color.b) /
                (imageToWriteOn[...].drawed + 1);
            ++imageToWriteOn[...].drawed;
        }
    }
}
```

Listing 8.9. Fragment kodu jadra obliczeniowego odpowiedzialny za porównanie obrazów

```
Pixel firstPixel, secondPixel;
for (int y = parameters->height - 1; y >= 0; --y) {
    for (int x = parameters->width - 1; x >= 0; --x) {
        firstPixel = imageToWriteOn[imageSize * globalId + y * parameters->width + x];
        secondPixel = validImage[y * parameters->width + x];
        diff += sqrt((float)
            ((firstPixel.r - secondPixel.r) * (firstPixel.r - secondPixel.r)
            + (firstPixel.g - secondPixel.g) * (firstPixel.g - secondPixel.g)
            + (firstPixel.b - secondPixel.b) * (firstPixel.b - secondPixel.b)));
    }
}
comparisonTable[j].value = (maxDiff - diff) / maxDiff;
comparisonTable[j].index = j;
```

Kod (listing 8.9.) odpowiada za porównanie obrazu wejściowego do wygenerowanego. Jedyną różnicą jest zapisanie, po wyliczeniu, takiego porównania w tablicy zawierającej jego wynik. Odpowiadającą metodą w wersji sekwencyjnej była funkcja compare.

Analizując kod drugiego jądra obliczeniowego odpowiedzialnego za mutację elity, można zauważyć podobny schemat. Na początku są tworzone struktury, na których będzie operował algorytm. Te struktury to Point, Color, Rectangle, Parameters, Comparison.

Wartym nadmienienia jest fakt, że w odróżnieniu od implementacji szeregowej, w przypadku OpenCL wymagane było zaimplementowanie własnej funkcji generującej losowe wartości liczbowe. Funkcja ta wygląda następująco:

```
inline int rand(int firstSeed, int secondSeed, int globalID)
{
    int seed = firstSeed + globalID;
    int t = seed ^ (seed << 11);
    return secondSeed ^ (secondSeed >> 19) ^ (t ^ (t >> 8));
}
```

W kolejnych liniach znajduje się kod kernela, który analogicznie do poprzedniego, najpierw pozyskuje wszystkie potrzebne zmienne środowiskowe oraz ustala ile obrazów będzie mutowanych na jednym jądrze obliczeniowym. Następnie podobnie, jak poprzednie, dla każdego obrazu, który ma zostać zmutowany, zostaje wywołany blok kodu odpowiedzialny za mutację (listing 8.10.).

Listing 8.10. Mutacja obrazu

```
for (int i = 0; i < parameters->numberOfRectangles; ++i) {
    if (rand(12, parameters->numberOfRectangles, j) % 2 > 0) {
        rectangles[(comparisonResults[j].index * parameters->numberOfRectangles) + i] =
            rand(2, parameters->numberOfRectangles, j) % 4 > 0 ?
            rectangles[comparisonResults[i2].index * parameters->numberOfRectangles + i] :
            getNewRectangle(parameters, j);
    }
    else {
        rectangles[(comparisonResults[j].index * parameters->numberOfRectangles) + i] =
            rand(1, parameters->numberOfRectangles, j) % 4 > 0 ?
            rectangles[comparisonResults[i1].index * parameters->numberOfRectangles + i] :
            getNewRectangle(parameters, j);
    }
}
```

Blok ten odpowiada funkcji `mutateElite` oraz `mutateSingleImage` z implementacji szeregowej.

Przechodząc z powrotem do kodu gospodarza, który wykonuje jądra obliczeniowe, warto nadmienić w jaki sposób są alokowane tablice w buforze oraz jak wykonywana jest transmisja zarówno programów jak i buforów.

Listing 8.11. Stworzenie obiektu kernela oraz bufora

```
auto valueOfImageKernel = cl::make_kernel<cl::Buffer&, cl::Buffer&, cl::Buffer&,
cl::Buffer&, cl::Buffer&> (valueOfImageProgram, "ValueOfImage", &error);
auto valueOfImageQueue = cl::CommandQueue(context, 0, &error);
auto paramsInBuffer = cl::Buffer(context, CL_MEM_READ_ONLY,
sizeof(GeneticAlgorithmKernelParameters), NULL, &error);
```

Na listingu 8.11. można zauważyć stworzenie kernela wraz z buforem, które będą stanowiły jego argumenty wejściowe, stworzenie kolejki w danym kontekście oraz stworzenie pojedynczego bufora, w tym przypadku dla parametrów wejściowych kernela. Z uwagi na to, że takich buforów jest tworzonych dokładnie pięć, a schemat ich tworzenia jest analogiczny, warto omówić tworzenie go na pojedynczym przykładzie. Dla parametrów tworzony bufor jest pamięcią jedynie do odczytu. Tworzony jest tylko dla danego kontekstu, to znaczy, że jądro obliczeniowe w swojej definicji nie będzie mogło przypisywać żadnej wartości do tych komórek pamięci. W przypadku zapisu, rezultatem byłoby rzucenie wyjątku. Rozmiar bufora wynosi tyle, ile bajtów zajmuje struktura `GenericAlgorithmKernelParameters`.

Funkcja `Calculate()`, w porównaniu do jej implementacji w wersji szeregowej, różni się wyłącznie inicjalizacją środowiska OpenCL, tworzeniem buforów oraz uruchomieniem jąder obliczeniowych.

Listing 8.12. Uruchomienie obliczenia wartości podobieństw obrazów

```
valueOfImageQueue.enqueueWriteBuffer(paramsInBuffer, CL_FALSE, 0,
sizeof(GeneticAlgorithmKernelParameters), &params, NULL, &paramsEvent);
events.push_back(paramsEvent);
valueOfImageQueue.enqueueWriteBuffer(validImageInOutBuffer, CL_FALSE, 0, validImageSize,
this->nativeImage->Area, NULL, &validImageEvent);
events.push_back(validImageEvent);
valueOfImageQueue.enqueueWriteBuffer(validImageToWriteOnInOutBuffer, CL_FALSE, 0,
validImageToWriteOnSize, imagesToWriteOn, NULL, &imageToWriteOnEvent);
events.push_back(imageToWriteOnEvent);

valueOfImageQueue.enqueueWriteBuffer(rectanglesInOutBuffer, CL_FALSE, 0, sizeOfOutput,
this->imagesRectangles, NULL, &rectanglesEvent);
events.push_back(rectanglesEvent);

valueOfImageKernel(cl::EnqueueArgs(valueOfImageQueue, events,
cl::NDRange(numberOfThreads), cl::NDRange(numberOfThreads == 16 ? 16 : 32)),
rectanglesInOutBuffer, paramsInBuffer, comparisonInOutBuffer,
validImageInOutBuffer, validImageToWriteOnInOutBuffer);
valueOfImageQueue.finish();

valueOfImageQueue.enqueueReadBuffer(comparisonInOutBuffer, CL_FALSE, 0, sizeOfComparison,
this->comparisonResults, NULL, &comparisonEvent);
```

W pierwszej linii listingu 8.12. można zaobserwować zapis zmiennej `params` do bufora `paramsInBuffer`. We wszystkich wersach, w których znajduje się wywołanie metody `enqueueWriteBuffer` na obiekcie `valueOfImageQueue` wykonywana jest podobna operacja. Przy każdej akcji skopiowania danych do bufora jest wykonany zapis w tabeli obiektu zdarzenia związanego z tą operacją z uwagi na to, że każdy zapis do bufora jest asynchroniczny, o czym świadczy drugi argument w funkcji `enqueueWriteBuffer`, czyli `CL_FALSE`. Po załadowaniu wszystkich danych wywoływany jest `operator()` obiektu `valueOfImageKernel`, który tworzy siatkę obliczeniową o rozmiarze globalnym równym wartości zmiennej `numberOfThreads`, rozmiarze lokalnym 16 lub 32, oraz ładuje wszystkie buforów do kernela jako jego argumenty. Wartym zauważenia jest, że nie wykonano żadnej funkcji, która czekałaby na wcześniejsze skończenie zapisu danych do buforów przed wykonaniem kernela. Jest to spowodowane tym, że drugi argument w `operatorze()` dla pierwszego argumentu `operatora()` obiektu `valueOfImageKernel` czeka na zakończenie wszystkich eventów zapisanych w tablicy `events`. W tym miejscu należy wspomnieć, że przy implementacji tego rozwiązania dowiedziano się [7], że w przypadku jedynie alokacji miejsca w buforze, bez kopiowania do niego danych, nie powinno się zdarzenia z nim skorelowanego zapisywać do listy `events`, ponieważ w takim przypadku jądro obliczeniowe nie będzie czekało na zakończenie obliczeń, co jest pewnym niedociągnięciem technologii OpenCL.

Po wykonaniu samego kernela następuje odczytanie danych zapisanych w buforze do pamięci zaalokowanej na urządzeniu gospodarza. Dzieje się to poprzez funkcję `enqueueReadBuffer()`, tak jak i w przypadku zapisu do bufora, tak i teraz jest to operacja asynchroniczna, o czym informuje drugi argument, zatem należy poczekać na zakończenie tej operacji przed wykonaniem jakichkolwiek operacji na danych.

```
cl::Event::waitForEvents(events);
```

W przypadku ładowania danych dla kernela odpowiedzialnego za mutację obrazu, operacje są analogiczne, różnią się jedynie liczbą buforów. Warto nadmienić, że w celu optymalizacji programu i zniwelowania czasu wykonania, buforów które się nie zmieniają nie są ładowane w każdej generacji na urządzenie, są ładowane tylko jednokrotnie.

9. WYNIKI PORÓWNANIA TECHNOLOGII

Celem pisanej pracy było zbadanie czasów wykonania programów napisanych w technologiach MPI, OpenMP i OpenCL oraz pokazanie, która z nich okazała się najlepsza dla zaimplementowanych algorytmów. Odzwierciedlają one bowiem najważniejsze obszary problemów przetwarzania równoległego, co wpływa na sens wykonanych testów.

Pierwszą kwestią, na którą warto zwrócić uwagę jest fakt, że przed implementacją programów w opisywanych technologiach, każdy z nich został początkowo napisany w wersji szeregowej z wykorzystaniem języka C++. Takie działanie zapewniło użycie wspólnych fragmentów kodu dotyczących implementowanych algorytmów we wszystkich testowanych programach. Dzięki temu różnice w czasach działania programów spowodowane są jedynie charakterystykami stosowanych technologii.

Opis każdej implementacji stworzono, biorąc pod uwagę skalowalność wyników, czyli czasów wykonania całego programu w zależności od liczby jednostek liczących oraz wpływ uruchomienia środowiska na rezultat, a na ich podstawie wyodrębniono cechy szczególne każdej technologii.

9.1. Algorytm generujący obraz zbioru Mandelbrota

Pierwszym algorytmem, dla którego porównano technologie MPI, OpenMP oraz OpenCL jest algorytm Mandelbrota. Jego szczegółowa charakterystyka została opisana w rozdziale 5. Opis implementowanych algorytmów. Jak już wspomniano we wstępie podczas opisów i wyjaśnień wzięto pod uwagę czasy wykonania całego programu z uwzględnieniem jego części składowych indywidualnie do opisywanych technologii.

9.1.1. MPI

Porównanie zaczęto od charakterystyki rezultatów napisanego programu w technologii Message Passing Interface. Analizę oparto głównie o czas wykonania całego programu wraz z inicjalizacją środowiska.

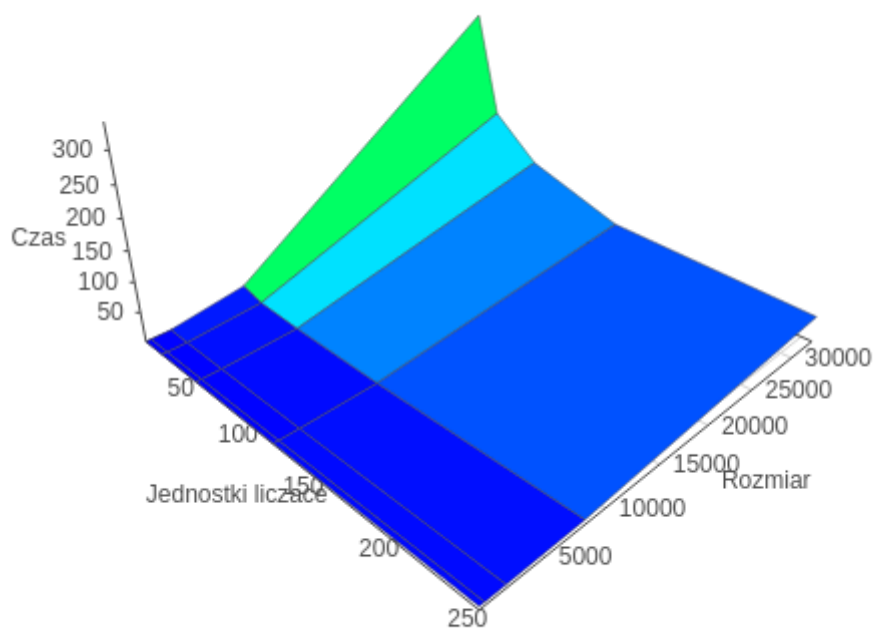
Jedną z pierwszych rzeczy, którą można zauważyć jest fakt, iż MPI uzyskuje oczekiwaną skalowalność do 128 procesów. Można wyraźnie zauważyć, że przy zwiększaniu dwukrotnie liczby pracujących procesów czas maleje dwukrotnie. Natomiast przy zwiększaniu z 128 do 256 procesów efekt jest różny w zależności od wielkości generowanego obrazu. Dla małych rozmiarów obrazu o długości boku mniejszym niż 8192 pikseli czasy wykonania stają się dłuższe. Natomiast dla dużych, których długość i szerokość wynoszą 32 tysięcy pikseli, zwiększanie liczby procesów nadal przynosi korzyść, jednak stosunek czasów maleje z wartości 2 do około 1,3. Spowodowane jest tym, że przy tak dobranych parametrach łączny czas komunikacji pomiędzy utworzonymi procesami staje się dłuższy, a ruch na złączu PCI Express zwiększa się. Z tego faktu wynika zaobserwowana zależność pomiędzy liczbą procesów, a wielkością generowanego obrazu, iż wraz ze zwiększaniem wielkości problemu szczyt wydajności zostaje osiągnięty dla większej liczby tworzonych procesów. W tym miejscu należy wspomnieć o zaobserwowanych wynikach analizy przebiegu działania programów w aplikacji VTune. Czas działania funkcji

systemowych stanowi dużą część wykonywanych programów i rośnie on wraz z liczbą tworzonych procesów, co wpływa na opisane wcześniej zjawisko.

Jedną z rzeczy, o których należy pamiętać przy analizie wyników działania programu w MPI, jest fakt, że obliczenia częściowo nakładają się z komunikacją. Wpływa to na korzyść rezultatów badanej technologii, jednak ten aspekt wzięto pod uwagę w rozdziale 9. Wyniki porównania technologii, gdzie porównano działanie wszystkich implementacji.

Należy się spodziewać, że dla większej liczby procesów czasy wykonania programów powinny rosnąć dla każdego rozmiaru danych wejściowych, co spowodowane jest zbyt dużą ilością komunikacji pomiędzy tworzonymi procesami i częstym przełączeniem kontekstu na poszczególnych rdzeniach. Zmiana ta, w przeciwieństwie do przełączania kontekstu wątku łączy się z wymianą większej ilości zasobów, związanej chociażby z mechanizmami pamięci wirtualnej.

Warto również wspomnieć, że na podstawie zebranych rezultatów można stwierdzić, iż narzut związany z uruchomieniem środowiska MPI wynosi około 3-30% w stosunku do czasu działania całego programu. Duży narzut jest charakterystyczny dla wywołania algorytmów z dużą liczbą procesów, jak i zadanym małym rozmiarem obrazu. Wynika to z faktu, iż czas uruchomienia środowiska zwiększa się wraz z wzrostem liczby uruchamianych procesów, przy czym ogólny czas obliczeń zmniejsza się. Natomiast narzut MPI_Init oraz MPI_Finalize wynosi około 2-9,5%.



Rys. 6.1. Wykres zależności czas, rozmiaru oraz jednostek liczących technologii Message Passing Interface dla algorytmu z dużą liczbą prostych obliczeń

9.1.2. OpenMP

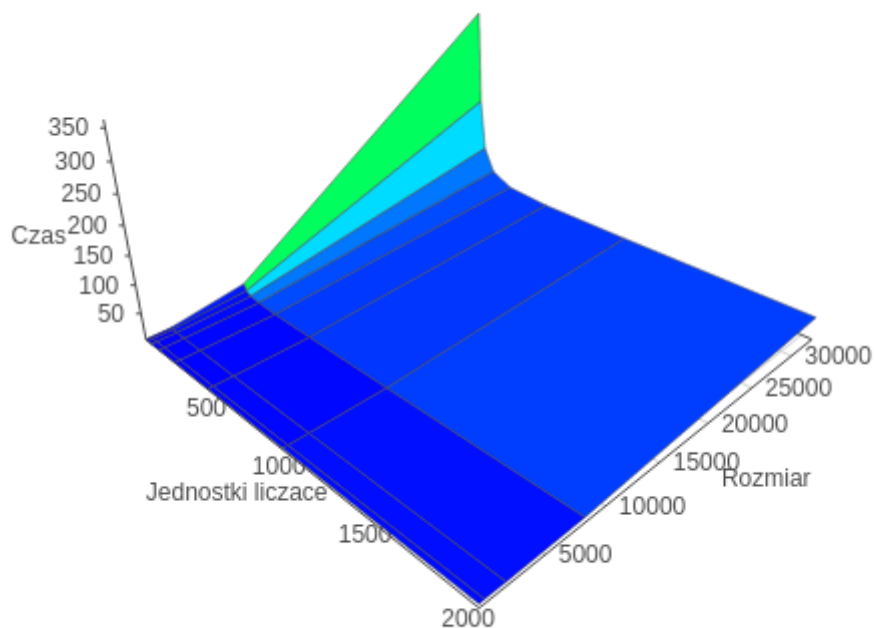
Kolejną porównywaną technologią dla algorytmu Mandelbrot jest OpenMP. W tym przypadku ogólną analizę również oparto na czasach wykonania całości programu, a jej rezultaty opisano w poniższym rozdziale.

Na wstępie należy zauważyć, że program skaluje się zgodnie z oczekiwaniami dla większych rozmiarów danych. Widać to w szczególności dla rozmiaru boku obrazu około 32 tysiące pikseli, który zyskuje na czasie wykonania, aż do wykorzystania 512 wątków, gdzie osiąga swój najlepszy czas, 37 sekund. Przy użyciu większej liczby jednostek obliczeniowych dla tak dużego obrazu zaobserwujemy spadek wydajności. Spowodowane jest to faktem przełączania kontekstu wątków, co znacząco wpływa na jakość pracy poszczególnych jednostek. Warto również tutaj wyjaśnić ciekawy fakt z tym związany, iż mimo zalecenia tworzenia 236 wątków na koprocesorze Intel Xeon Phi, najlepszy czas uzyskujemy dla 512 wątków. Wynika to z tego, iż duża część wątków wykonuje małą liczbę obliczeń, przez co po krótkim czasie działa już ich mniejsza liczba, lecz obliczenia są rozłożone równomiernie w pozostałym czasie wykonania programu.

W kontraście do czasów wykonania dużych obrazów warto opisać rezultaty pracy dla problemów o małym rozmiarze. Zauważono, że wraz ze wzrostem rozmiaru boku, ogólny czas działania programu nie ulega dużym zmianom. Można to wyjaśnić dużym narzutem inicjalizacji środowiska w stosunku do czasów obliczeń. Ustalono, że jest on stały i nie zależy od wielkości problemu, wynosi około 3 sekundy. Wpływa to na fakt, że dla małych problemów udział procentowy środowiska uruchomieniowego jest bardzo duży i wynosi około 90%.

W kontekście opisywanej technologii warto również napisać o wynikach przy wykorzystaniu różnych wartości Thread Affinity. Opcja ta została wcześniej dokładnie opisana w rozdziale 4. Opis Porównywanych technologii, tutaj skupiono się jedynie na porównaniu czasów wykonania algorytmu Mandelbrot. Pomiary zrobiono dla dwóch typów Thread Affinity, scatter oraz compact, przy ustawieniu granulacji na wartość „fine”, która oznacza wzięcie pod uwagę wszystkich logicznych procesorów (również tych będących efektem działania technologii HyperThreading) przy dystrybucji wątków.

Z analizy rezultatów widać, iż typ Scatter uzyskał znacznie lepsze wyniki, do około 1,5x, dla rozmiaru obrazu o boku 32 tysiące pikseli. Wynika to z faktu, iż wątki są rozłożone równomiernie na wszystkich rdzeniach, co powoduje, że zwłaszcza dla wykorzystania liczby wątków mniejszej lub równej 118 wszystkie wykonują obliczenia równocześnie. Natomiast warto zauważyć, że dla małych problemów czasy wykonania są bardzo podobne. Nikłą różnicę, około 0.3 sekundy na korzyść opcji compact, można próbować wytłumaczyć lokalnością danych na pojedynczym rdzeniu.



Rys. 6.2. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenMP dla algorytmu z dużą liczbą prostych obliczeń

9.1.3. OpenCL

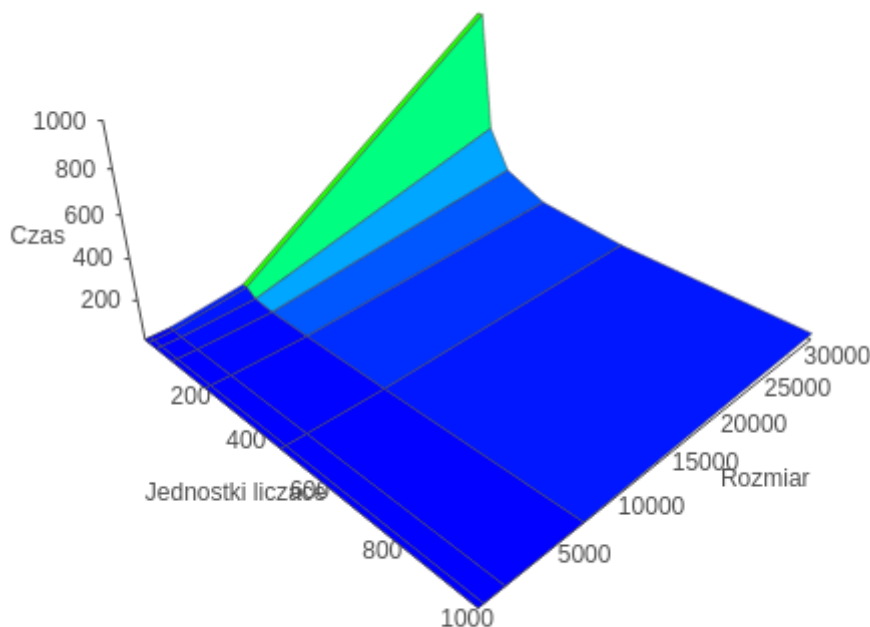
Ostatnią technologią, względem której porównywane jest działanie implementacji algorytmu Mandelbrot jest OpenCL. Również jak dla poprzednich algorytmów wzięto pod uwagę czas wykonania całego programu.

Pierwszą rzeczą, którą należy zauważyć są bardzo duże rezultaty czasowe przy małej liczbie wątków w grupie roboczej. Może wynikać to ze specyfiki implementacji technologii OpenCL na XeonPhi, która oparta jest na Intel Threading Building Blocks. Działa na zasadzie dystrybucji zadań na uruchomione 236 wątków.

Innym faktem, który można zaobserwować jest ciągła redukcja czasu wykonania, aż do obranej maksymalnej liczby wątków. Wynika to z opisywanej wcześniej implementacji OpenCL, gdyż poprzez tę zmianę zrównoważono obciążenie poszczególnych rdzeni. Należałoby przypuszczać, iż w którymś momencie czas związany z przełączaniem zadań przewyższy korzyść związaną ze zwiększaniem granulacji problemu, a zatem lepszym zbalansowaniem obciążenia.

Na koniec warto wspomnieć o czasie związanym z inicjalizacją środowiska uruchomieniowego. Bez pomiaru pobrania urządzeń wynosił on od 9% do 60% działania programu. Można wnioskować, że czas ten jest stały i wynosi około 3,1 sekundy, dlatego dla małych rozmiarów problemu czas ten często stanowi większą część programu niż obliczenia. Patrząc na udział procentowy czasu inicjalizacji wraz z pobraniem całego kontekstu wynosi od 4,5% do 40%. Przy czym również jest to związane z wielkością rozwiązywanego problemu, dlatego jak łatwo się domyślić, tak samo w tym przypadku, większy udział procentowy związany

z narzutem uruchomienia środowiska i pobrania kontekstu jest związany z generowaniem obrazu o małym boku.



Rys. 6.3. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenCL dla algorytmu z dużą liczbą prostych obliczeń

9.1.4. Porównanie wyników każdej technologii

Opisano już wszystkie technologie i wyjaśniono ich rezultaty, jednak celem pracy inżynierskiej jest porównanie między sobą MPI, OpenMP oraz OpenCL. W poniższym rozdziale przedstawiono, która z testowanych technologii okazała się być najlepszą i dla jakich wielkości problemu. Starano się również wyjaśnić różnice czasowe w rezultatach testowanych programów.

Ze względu na charakterystykę analizowanego problemu, czyli dużą intensywność obliczeń w porównaniu z pozostałymi implementowanymi algorytmami najlepsze rezultaty pod względem wydajności zostały uzyskane dla OpenCL. Wynika to z opisanego wyżej faktu, iż implementacja najlepszej technologii zamiast uruchamiać zadaną liczbę wątków określoną jako rozmiar globalny, rozdysponowuje zadania między istniejącymi 236 jednostkami roboczymi. Dzięki temu marginalizowany jest efekt przełączania kontekstu pracy wątków, co ma miejsce w przypadku technologii OpenMP. Jeszcze większy efekt ma to w przypadku MPI, gdzie nie tylko przełączany jest kontekst wątków, ale również procesów.

Należy jednak zauważyć, że dla małych rozmiarów problemów najlepszy okazuje się być MPI. Spowodowane jest to modelem komunikacji pomiędzy hostem, a koprocesorem Xeon Phi, gdzie proces główny komunikuje się z każdym z pozostałych węzłów oddzielnie, co powoduje naturalne nakładanie się obliczeń i transferu danych. Spotęgowane jest to nierównomiernym rozłożeniem obliczeń, w przypadku statycznego podziału zadań dla problemu Mandelbrota. W

innych technologiach komunikacja następuje dopiero po zakończeniu wszystkich obliczeń na koprocesorze.

Biorąc pod uwagę wszystkie aspekty można wnioskować, że dla danego problemu dużej intensywności obliczeń, technologia OpenCL jest najbardziej optymalna.

9.2. Aplikacja klient-serwer

Drugim z porównywanych programów jest aplikacja nastawiona na intensywną komunikację pomiędzy hostem, a koprocesorem. Na wstępie warto zauważyć, że analizę oparto, jak przy pozostałych algorytmach, na czasie działania całego algorytmu wraz z komunikacją. Należy również podkreślić, że model zastosowany w implementacji OpenMP różni się nieco od tego, zastosowanego w pozostałych technologiach, który został opisany w rozdziale 5. Opis implementowanych algorytmów. Zmiany w modelu opisano w odpowiednim rozdziale poniżej.

9.2.1. MPI

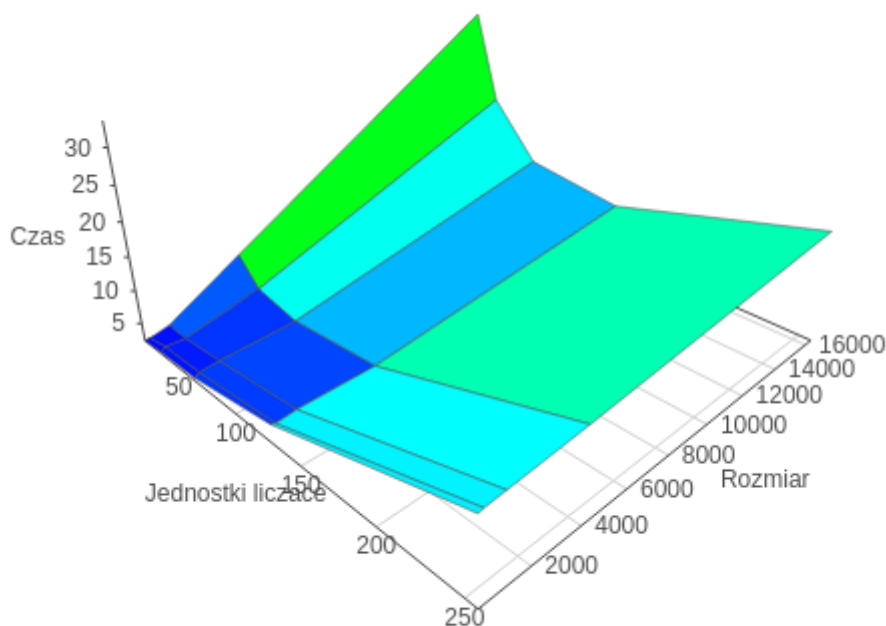
Pierwszą z analizowanych technologii jest Message Passing Interface. Jego charakterystyka jest bardzo zbliżona do analiz prowadzonych w przypadku innych algorytmów, jednak w poniższym rozdziale starano się wskazać różnice lub jeszcze raz podkreślić cechy szczególne technologii, które wystąpiły również w tej implementacji.

Pierwszą rzeczą, którą można zauważyć, jest fakt, że dla małych wartości liczby żądań do 1024, zwiększanie liczby procesów nie powoduje zwiększenia wydajności, jest wręcz przeciwnie. Spowodowane jest to bardzo krótkim czasem obliczeń w stosunku do komunikacji. Z analizy przeprowadzonej w aplikacji VTune wynika, iż w przypadku małej liczby poleceń, czas poświęcony na obliczenia wykonane w ramach przetwarzania, trwają zaledwie 2,7% całego wykonania programu w porównaniu do 70,3% zajętych przez komunikację. Po szesnastokrotnym zwiększeniu rozmiaru żądań stosunek ten zmienia się. Czas obliczeń rośnie niemal pięciokrotnie do wartości 12,9%, natomiast czas komunikacji zmniejsza się do 22,6% czasu wykonania. Należy podkreślić, że w przypadku tej analizy, procesy spędzają większość swojej pracy na wykonywaniu funkcji biblioteki libc, nie udało się jednak określić dokładniejszych informacji.

Opisany wcześniej efekt wyrównania czasu obliczeń i komunikacji dla dużych rozmiarów problemu, ma również odzwierciedlenie w wynikach, gdzie widać, że czasy wykonania zmniejszają się wraz ze wzrostem wykorzystanej liczby procesów. Warto zauważyć, że w takim wypadku dla opisywanego programu szczyt wydajności jest osiągany dla 128 procesów, gdzie czas wykonania wynosi niecałe 9 sekund.

Należy również wspomnieć o udziale inicjalizacji środowiska w wykonanych pomiarach. Wynika z tego, iż narzut środowiska uruchomieniowego w mpirun wynosi około 6-8% czasu działania aplikacji, natomiast narzut związany z MPI_Init_thread() oraz MPI_Finalize() wynosi od 11% do 22%. Wyższa wartość procentowa odpowiada programom, w których użyto większą liczbę procesów, gdyż takie działanie powoduje wzrost czasu związanego z ich utworzeniem i zamknięciem.

Po raz kolejny wyróżniono zachowania Message Passing Interface, podobne do opisywanych w poprzednich rozdziałach. Szczególnie ten algorytm pokazuje, iż narzut związany z komunikacją pomiędzy procesami ma znaczący wpływ w porównaniu z czasem wykonywanych obliczeń. Reasumując należy tak dopasować liczbę wykorzystanych jednostek obliczeniowych do rozmiaru problemu, aby czas komunikacji stanowił mniejszy procent wykonania programu, niż praca nad obliczeniami.



Rys. 6.4. Wykres zależności czas, rozmiaru oraz jednostek liczących technologii Message Passing Interface dla algorytmu z dużą liczbą komunikacji

9.2.2. OpenMP

Kolejną technologią, która została opisana jest OpenMP. W tym przypadku należy podkreślić, iż zastosowany model różni się od implementacji w pozostałych technologiach, ograniczenie z tym związane zostało opisane w pierwszej części analizy. Dalsza skupia się na wyjaśnieniu uzyskanych rezultatów.

W implementacji tej technologii zdecydowano, że paczki danych przesyłane będą do koprocatora grupami. Jej wielkość była zależna od wykorzystanych wątków, gdyż każdy z nich był odpowiedzialny za posortowanie jednej tablicy. Ten model został uproszczony o brak komunikacji asynchronicznej, tak, aby gdy, któryś z wątków skończy obliczenie dostał od hosta, kolejną porcję danych, gdyż nie można było wysłać kolejnej paczki danych, podczas gdy host nie otrzymał poprzednich rezultatów.

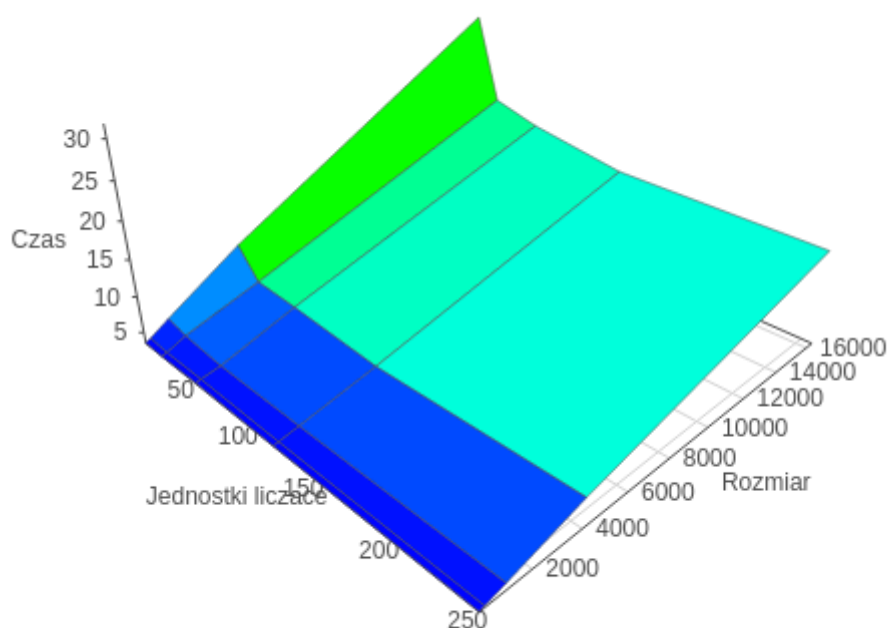
Zastosowanie takiego modelu spowodowane było ograniczeniem związanym z przestarzałą wersją kompilatora OpenMP na akceleratorze Xeon Phi. Nie dało to możliwości

użycia technologii Stream, która najprawdopodobniej pozwoliłaby na rozwiązanie komunikacji asynchronicznej, pomiędzy hostem, a koprocesorem.

W kontekście skalowania warto zauważyć, że dla opisywanego problemu czas związany z wykorzystaniem obszaru offload i komunikacją jest dużo większy niż czas obliczeń na koprocesorze. Przez to maksimum wydajność dla największego testowanego problemu, osiąga się dla wartości 128 wątków, jednak w porównaniu z czasem osiągniętym dla ośmiokrotnie mniejszej liczby jednostek jest zaledwie 2,12 razy dłuższy.

Przedstawiając statystyki, warto również napisać, że po zwiększeniu żądań z rozmiaru 1024 do 16384, udział obliczeń w działaniu programu wzrasta z poziomu 24% do 37,4% całości wykonania programu. Można zauważyć, że czas związany z działaniem funkcji implementujących dyrektywę offload stanowi 11,4% działania programu dla rozmiaru żądania 16 tysięcy. Natomiast analizując udział procentowy startu środowiska w opisywanej technologii można zauważyć, że jest on stały i wynosi około 2,6 sekundy.

Podsumowując charakterystykę technologii OpenMP dla opisywanego problemu, warto podkreślić, że wykonywanie zbyt wielu osobnych regionów na koprocesorze wiąże się z dużym narzutem. Aczkolwiek można mieć nadzieję, że wykorzystanie konstrukcji funkcji z rodziny Stream poprawiłoby osiągi dla zadanego problemu.



Rys. 6.5. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenMP dla algorytmu z dużą liczbą komunikacji

9.2.3. OpenCL

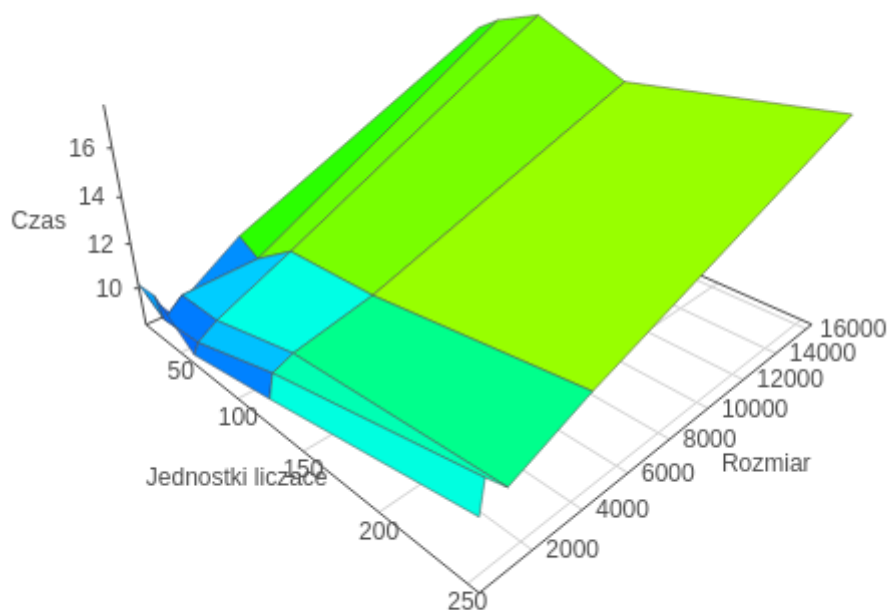
W tym rozwiązaniu można zauważyć mniejszą skalowalność czasu wykonania z uwagi na duży narzut wywołany utrzymaniem dużej liczby kolejek, co wiąże się z utrzymywaniem dużej ilości połączeń na poziomie COI (Coprocesor Offload Infrastructure).

Warto zauważyć, że średni czas pojedynczego przetwarzania żądania rośnie wraz ze zwiększaniem liczby kolejek. Może się to wiązać również z niespełnieniem zaleceń firmy Intel co do optymalizacji aplikacji OpenCL pisanych na urządzenia Intel Xeon Phi, które zakładają użycie 32 jednostek roboczych w pojedynczej grupie. Wymagania nie zostały spełnione z uwagi na realizację modelu, który miał implementować aplikacja. W tym przypadku każda grupa robocza posiada pojedynczą jednostkę roboczą.

Podczas analizy w narzędziu VTune stwierdzono, że czas poświęcony na zarządzanie środowiskiem przez TBB, wynosi około 65% dla rozmiaru problemu 1024 przy wykorzystaniu 64 jednostek roboczych. Natomiast obliczenia zajmują jedynie poniżej 1% całości wykonania programu.

Jak w przypadku opisu innych technologii, tutaj również warto przestawić czasy związane z narzutem środowiska. Samo uruchomienie wynosi około 3,1 sekundy, natomiast narzut czasowy związany z tworzeniem kontekstu nie jest stały i wynosi od 10 do 34% czasu wykonania programu, co przekłada się na 1,5 sekundy do 3,5. Wiaże się to z innym sposobem pomiaru czasu, niż w implementacjach poprzednich problemów. Z uwagi na to, że w tym rozwiązaniu w czasie tworzenia był również zawarty czas tworzenia buforów, dlatego czasy te różnią się w zależności od rozmiaru problemu.

Zastosowany model asynchroniczny OpenCL w przypadku opisywanej technologii okazał się bardzo wygodnym modelem, aczkolwiek z powodu konieczności utrzymywania bardzo wielu połączeń między hostem, a koprocesorem nie została osiągnięta oczekiwana wydajność.



Rys. 6.6. Wykres zależności czas, rozmiaru oraz jednostek liczących technologii OpenCL dla algorytmu z dużą liczbą komunikacji

9.2.4. Porównanie wyników

Z wyniku analizy pomiarów wyszło, iż technologia Message Passing Interface okazała się być najlepsza, z powodu nastawienia na komunikację pomiędzy osobnymi procesami, co ma duże znaczenie w testowanym problemie na zastosowany podział pracy. W jednym momencie jeden proces odpowiada za obsługę jednego żądania, podczas gdy inne technologie nastawione są na komunikację zbiorczą o małej granulacji. Dlatego przy założeniu innego modelu, czyli obliczania żądań w grupach po 16 tablic, zwłaszcza technologia OpenCL mogłaby zyskać z powodu mocnego nastawienia na wykorzystanie technologii wektoryzacji.

Warto w tym miejscu zauważyć, że dla małych wielkości problemu najlepszą technologią jest OpenMP. Jednak z uwagi na zastosowanie w niej innego modelu, pominięta jej wyróżnienie w stosunku do MPI.

9.3. Algorytm genetyczny

Ostatnim programem, który porównano jest algorytm genetyczny. Tak jak w przypadku innych implementacji, również tutaj wzięto pod uwagę głównie czas wykonania całości programu, według którego stworzono poniższą charakterystykę.

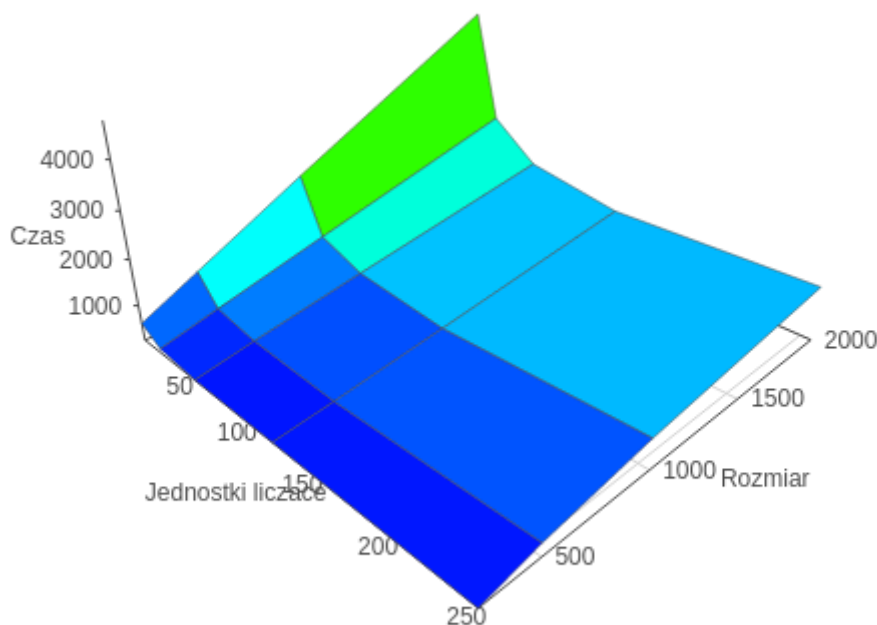
9.3.1. MPI

Opisując skalowalność, można zauważyć podobne zachowanie, jak te zaobserwowane w przypadku problemu Mandelbrota. Z kolejnymi krokami zwiększania liczby procesów zmniejsza

się poziom skalowalności z wartości około 2x (przy zmianie z 16 procesów na 32) do 1,3x (przy zmianie z 64 procesów na 128). Warto nadmienić, że również w tym wypadku zwiększenie ilości procesów do 256 powoduje wzrost czasu wykonania programu. Szczyt wydajności zaobserwowano przy wykorzystaniu 128 procesów. Przy czym można zauważyć, że biorąc pod uwagę zrównolegloną część programu, czyli liczenie oceny obrazów, widać, że dla populacji równej 250 najbardziej wydajną liczbą wątków było 256. Możliwe, iż wynikało to z wykonywania zapisów do osobnych procesów na koprocessorze.

Ze względu na niedostosowanie modelu o rozproszonej pamięci do zrównoleglenia części programu odpowiadającej za mutację, nie została ona wykonana równolegle. Aczkolwiek stanowiła ona małą część całości działania programu. Analizując ogólne rezultaty implementacji algorytmu genetycznego, można zauważyć, że stanowi ona około 1‰ w stosunku do czasu wykonania całego programu.

W trakcie analizy aplikacji w narzędziu VTune, zauważono, że właściwe obliczenia zajmują zaledwie 60% efektywnego czasu pracy procesora, natomiast narzut konstrukcji technologii MPI wyniósł, aż 26,6%. Przy czym trzeba wziąć pod uwagę, iż wykorzystana została wersja biblioteki MPI w wersji debug. Trzecią główną składową są wykonania funkcji systemowych, które w dużej mierze odpowiadają za przełączanie kontekstu procesów, trwają one 12,6% czasu wykonania całego programu.



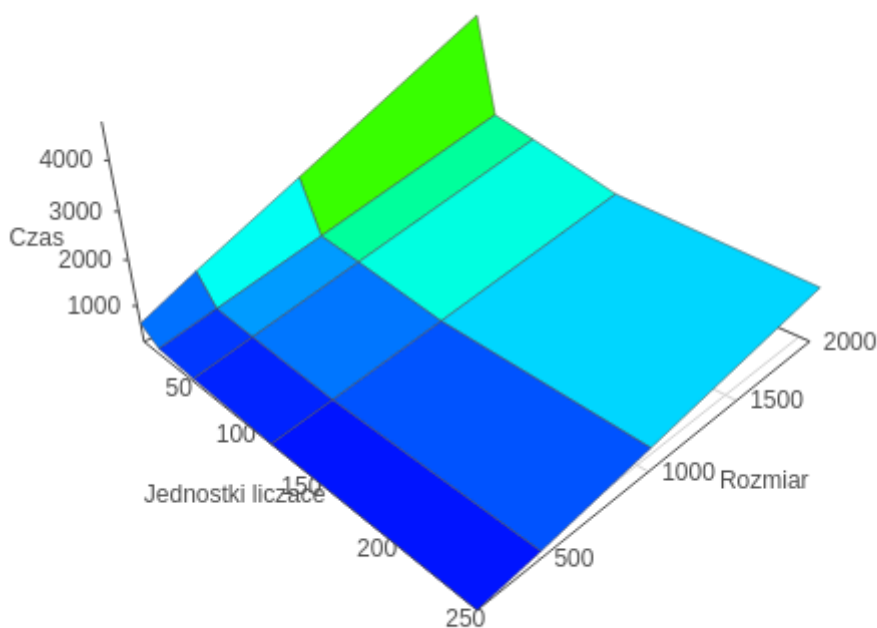
Rys. 6.7. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii MPI dla algorytmu z dużą liczbą obliczeń na dużej ilości danych

9.3.2. OpenMP

W przypadku tej technologii można zauważyć dobrą skalowalność czasu w odniesieniu do liczby wątków, aczkolwiek jest to widoczne do 64 wątków wyłącznie. Dla reszty testów nie można określić jednoznacznie optimum czasu wykonania, z uwagi na to, że dla wszystkich uzyskano polepszenie rezultatów.

W trakcie analizy wyników zauważono bardzo wysoki czas fazy mutacji, w porównaniu do implementacji szeregowej, w związku z czym wykonano analizę w programie VTune. Porównując czas wykonania pojedynczej fazy mutacji z czasem spędzonym na obliczeniach, zauważono, że narzut wynikający z offloadu jest relatywnie wysoki w stosunku do pełnego czasu wykonania tej części algorytmu, gdyż wynosi około 94%.

Wnioskując po wynikach uzyskanych w programie VTune można stwierdzić, że rezultaty związane z funkcjami systemowymi stanowiły niewielką część czasu wykonania algorytmu. Wynosiły one bowiem jedynie 10,6% wyniku finalnego. Ze względu na statyczny podział pracy również narzut związany ze zrównoleglaniem pętli jest bardzo mały. Warto tutaj zauważyć, że w trakcie analizy nie udało się określić narzutu związanego z konstrukcjami offloadu w skali całego programu, lecz można go uzyskać badając pojedynczą zrównolegloną sekcję, co było opisane akapit wyżej.



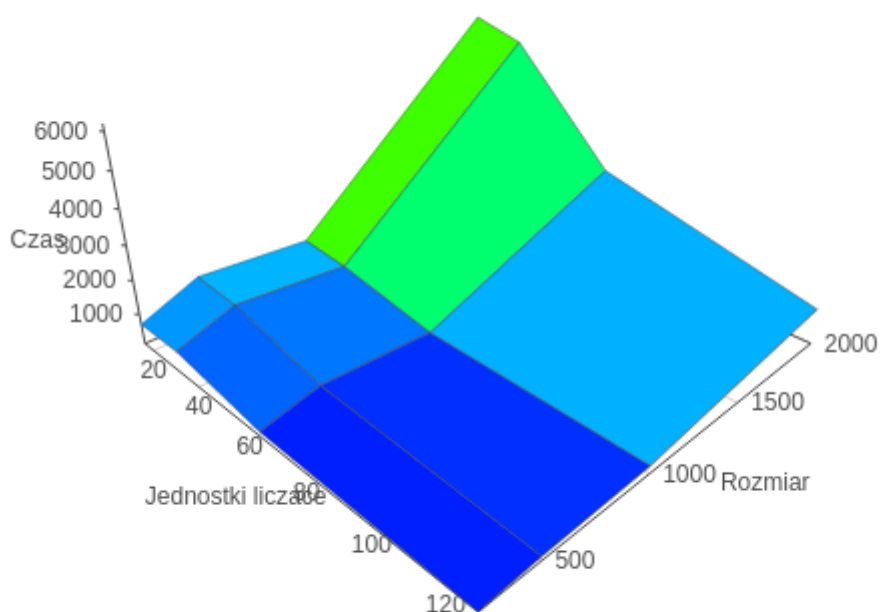
Rys. 6.8. Wykres zależności czas, rozmiaru oraz jednostek liczących technologii OpenMP dla algorytmu z dużą liczbą obliczeń na dużej ilości danych

9.3.3. OpenCL

Zauważono, że w przypadku tej technologii skalowalność jest zgodna z zakładaną. Wraz ze wzrostem liczby użytych jednostek roboczych zmniejsza się dwukrotnie czas wykonania programu. Należy jednak podkreślić, iż wyniki przy użyciu 16 i 32 jednostek obliczeniowych są bardzo podobne, z uwagi na użycie odpowiednio szesnastu i trzydziestu dwóch jednostek roboczych w pojedynczej grupie roboczej.

Stwierdzono pewne odchylenie wyników od zakładanych dla populacji o rozmiarze 1000. Niektóre z nich są wyjątkowo niskie, stąd zostały pominięte w analizie czasu wykonania programu. Należy jednak podkreślić, iż nie wszystkie wyniki dla opisywanego parametru wydają się błędne, stąd możliwość odrzucenia kilku z nich, bez konieczności powtarzania pomiarów.

Z uwagi na to, że program VTune nie mógł rozpoznać jednego z dwóch jąder obliczeniowych w zaimplementowanym algorytmie. Niemożliwym jest podanie udziału procentowego wykonania funkcji systemowych w aspekcie działania całego programu.



Rys. 6.9. Wykres zależności czas, rozmiaru oraz jednostek liczących technologii OpenCL dla algorytmu z dużą liczbą obliczeń na dużej ilości danych

9.3.4. Porównanie wyników każdej technologii

W analizach czasów wykonania został pominięty czas inicjalizacji środowiska z uwagi na wysokie skomplikowanie zaimplementowanych rozwiązań, a co z tego wynika, długi czas obliczeń realizowanych w ramach algorytmu. W porównaniach brane pod uwagę były również chybieńia linii w pamięci podręcznej zarówno poziomu L1 jak i L2. Po otrzymanych wynikach w programie VTune, stwierdzono, że najlepszy stosunek nie trafiać pomiędzy pamięciami

poziomów L1 i L2 miała technologia OpenCL gdzie relacja ta wynosiła około 1 do 5, natomiast w OpenMP oraz MPI było to na poziomie 1 do 3.

W porównaniu do algorytmu Mandelbrota jeszcze bardziej uwydatnia się przewaga technologii OpenCL z uwagi na większe skomplikowanie obliczeń, jak i również większy rozmiar danych. W kontraście do opisywanego wcześniej algorytmu Mandelbrota, w tym przypadku technologia OpenCL, okazała się również lepsza dla małej liczby procesów i małego rozmiaru problemu. Wynika to z opisanego wcześniej faktu, iż korzysta ona z technologii Intel Threading Building Blocks. Aczkolwiek dla dużego rozmiaru problemu i małej liczby wątków najlepszą technologią było OpenMP.

10. OPIS PRAC IMPLEMENTACYJNYCH

Aby zrealizować cel pracy inżynierskiej i porównanie poszczególnych technologii miało sens, początkowo należało każdy algorytm zaimplementować w sposób szeregowy. Użyto do tego języka C++, który potem rozszerzono o odpowiednie biblioteki, aby móc zastosować testowane technologie. Opis algorytmów, implementacje, a także charakterystyka stworzonych programów są tematami innych rozdziałów tej pracy. Tutaj skupiono się na użytych do implementacji narzędziach, a także dodatkowych technologiach lub bibliotekach, które pomogły w napisaniu zakładanych programów.

10.1. *Wykorzystane narzędzia i technologie*

Na początku warto omówić wykorzystane narzędzia oraz technologie, które umożliwiły i ułatwiły realizację projektu inżynierskiego. Wśród nich znajdują się środowiska programistyczne, oprogramowania kontroli wersji, aplikacje umożliwiające monitorowanie działania stworzonych programów, biblioteki ułatwiające implementację oraz stworzone do prezentacji narzędzia, które również były częścią projektu inżynierskiego.

Warto na wstępie omówić używane środowiska programistyczne i edytory tekstowe, które ułatwiały pisanie i pracę z kodem. Jednym z użytych programów był CLion, czyli środowisko programistyczne przeznaczone dla języków C/C++ produkowane przez spółkę JetBrains. Udostępnia ono szereg narzędzi i opcji pomocnych w implementacji, w tym takie podstawowe jak kompilacja, uruchamianie oraz debugowanie programów. Warto zauważyć, iż program ten jest narzędziem komercyjnym, jednak firma JetBrains udostępnia go studentom za darmo.

Innym programem użytym podczas pisania kodu był Visual Studio Code. Jest to prosty, darmowy edytor tworzony przez firmę Microsoft. Wspiera ono kolorowanie składni w większości popularnych obecnie języków, jednak w chwili obecnej debugowanie jest możliwe jedynie dla środowisk Node.js, TypeScript oraz JavaScript. Warto również podkreślić, że opisywany program posiada także oprogramowanie GIT służące do kontroli wersji i dzięki odpowiedniej konfiguracji daje możliwość przesyłania kodu do dowolnej usługi zdalnej.

Wśród narzędzi służących do tworzenia kodu warto również wyróżnić stosowany przy implementacji edytor VIM. Jest to rozszerzony klon edytora tekstu VI, napisany przez programistę Bramę Moolenaar, który należy do grupy oprogramowania o otwartym kodzie źródłowym. Warto zauważyć, że podczas realizacji projektu narzędzie to było używane w terminalu, podczas potrzeby szybkiej zmiany kodu lub modyfikacji prowadzonych z maszyny zdalnej. Wspiera ono kolorowanie składni dla języka C++ i jako, iż jego podstawowa wersja nie jest edytorem graficznym charakteryzuje się szybkością działania.

Kolejnym pomocnym oprogramowaniem jakie należy omówić jest GIT, czyli rozproszony system kontroli wersji stworzony przez Linusa Torvaldsa jako narzędzie wspomagające rozwój jądra Linux. Podobnie jak wcześniej opisywany VIM jest częścią otwartego oprogramowania.

Opisując GIT, warto podkreślić jego najważniejsze cechy, takie jak:

- silne wsparcie dla rozgałęzionego procesu tworzenia oprogramowania – oprogramowanie pozwala na tworzenie tzw. branchy, czyli oddzielnych ścieżek rozwijanego oprogramowania, które w każdej chwili w łatwy sposób można scalić z główną wersją projektu,
- praca off-line – dla wielu jest to największa korzyść z pracy z opisywanym oprogramowaniem, bowiem każdy programista ma na swoim komputerze lokalną kopię repozytorium, do którego może zatwierdzać wprowadzane zmiany, następnie te zmiany mogą być przenoszone pomiędzy innymi repozytoriami,
- kompatybilność z istniejącymi protokołami sieciowymi – repozytoria mogą być publikowane poprzez HTTP, FTP, rsync, SSH,
- efektywna praca z dużymi projektami – według twórcy wydajność opisywanego oprogramowania jest dużo wyższa od innych podobnych rozwiązań stosowanych na rynku,
- prywatne repozytorium – inaczej niż SVN, którego każda wersja musi być przesłana na serwer.

Opisując GIT, warto przejść do krótkiej charakterystyki używanego podczas implementacji serwisu BitBucket tworzonego przez firmę Atlassian. Służył on bowiem do trzymania aktualnych wersji repozytorium, gdyż opisywane wcześniej oprogramowania można w łatwy sposób połączyć z wymienioną usługą. Warto zauważyć, iż BitBucket umożliwia założenie darmowego konta, dzięki któremu pozwala na wgląd do zmian w utworzonym projekcie, a także na dodawanie komentarzy i ocenianie zmian nie tylko w głównej ścieżce projektu, ale także we wspomnianych wcześniej branchach.

Kolejnym narzędziem, który został wykorzystany podczas realizacji projektu, był Intel VTune Amplifier XE 2016, który pozwala na całościową analizę działania wybranego programu. Dostarcza on możliwość dynamicznej analizy przebiegu programu, poprzez monitorowania szeregu parametrów, jak chociażby całej gamy zdarzeń sprzętowych, takich jak np. liczba wykonanych instrukcji, chybionych odczytów, bądź zapisów w pamięci podręcznej, czy też wydajność wektoryzacji wybranych instrukcji. Przy wykonaniu prac została wykorzystana głównie możliwość analizy działania programów nie tylko na samym hoście, lecz głównie na samym koprocesorze Intel Xeon Phi. Owe narzędzie dostarcza również możliwość zbadania zjawiska niebalansowania obciążenia pomiędzy danymi wątkami. Wspiera on nie tylko analizę pojedynczego procesu, lecz też, w przypadku technologii MPI, rozpoznaje ją i zbiera informacje ze wszystkich pochodnych procesów. Został on również wykorzystany do wykrycia funkcji i modułów, w których wykonane programy spędzały najwięcej czasu. Pozwoliło to również na zapoznanie się z implementacją wewnętrzną wybranych technologii, co wobec braku dostępnej dokładnej dokumentacji, okazało się krytyczne w zrozumieniu działania programów.

Narzędziem, które zostało wykorzystane do prezentacji zebranych wyników, była napisana aplikacja webowa, która została wykonana przy użyciu wyłącznie języka JavaScript. Do jej realizacji zostały wykorzystane następujące narzędzia. Node.js wraz z biblioteką

ExpressJs jako serwer aplikacji internetowej, natomiast do samej prezentacji wyników została wykorzystana biblioteka Graph3D. Wykonana aplikacja pozwala na wyświetlenie zebranych wyników działania programów, umożliwiając wybór interesującego użytkownika algorytmu, a także jednej z dostępnych zmierzonych metryk w kontekście wybranego wcześniej algorytmu. Same wyniki prezentowane są zarówno w formie trójwymiarowego wykresu, jak i w postaci tabelarycznej. Oprócz wartości bezwzględnych czasu wykonania programów, wyświetlane są również różnice czasowe w wykonaniu programów, przy tych samych parametrach, pomiędzy wykorzystanymi technologiami, co prezentowane jest w sposób analogiczny do wyników bezwzględnych.

Ostatnią rzeczą wartą opisaną w tym podrozdziale jest nakładka na bibliotekę OpenCL dla języka C++. Zdecydowano się na jej użycie z uwagi na czystość kodu oraz na to, że implementacja w czystym OpenCL C byłaby bardzo rozległa i w przypadku tworzonych algorytmów powodowałaby problem w zrozumieniu kodu. Warto zauważyć, że użyto najnowszej obecnie wersji nakładki, mianowicie 1.2. Podsumowując, użyta biblioteka miała ułatwić pisanie programów jak i zwiększyć czytelność kodu.

Podczas pisania pracy skorzystano z dużej liczby narzędzi i bibliotek ułatwiających realizację projektu, których wykorzystanie było niezbędne lub dobrowolne.

W powyższym rozdziale skupiono się na wyczerpującym opisie, jednak uwagi związane z użyciem opisanych narzędzi w praktyce zostało opisane w rozdziale 12. Zebrane doświadczenia.

10.2. Ograniczenia

Podczas realizacji projektu napotkano na trudności, które uniemożliwiły zastosowanie rozwiązań prowadzących do przedstawienia planowanych modeli. Było to głównie spowodowane przestarzałymi technologiami znajdującymi się na maszynie apl12. Napotkane problemy, nieco dokładniej opisano w obecnym rozdziale.

Rozwiązanie problemu z wywołaniem kilku dyrektyw offload na koprocesorze Xeon Phi, mogłoby zapewnić wykorzystanie technologii Stream. Jednak ta opcja jest dostępna w 16-tej wersji kompilatora OpenMP, natomiast podczas realizacji projektu wymuszono skorzystanie z wersji 14.0.3, która była zainstalowana na wykorzystywanej maszynie.

Warto również napisać o ograniczeniach związanych z implementacją algorytmów w OpenCL. Znowu utrudnieniem okazała się przestarzała technologia dostępna na maszynach apl12. Z uwagi na to, użyto bibliotek w wersji 1.2. i nie było możliwości pisania kodu kerneli w C++14. Zamiast tego należało używać C99. Przetestowanie jak ma się względem wydajności obliczeń pamięć współdzielona pomiędzy kernelem, a hostem wprowadzona w standardzie OpenCL 2.0 również nie była możliwa z powodu wykorzystywanej wersji.

Opisane ograniczenia, które wyniknęły podczas realizacji projektu nie uniemożliwiły jego realizacji, jednak wymusiły zastosowanie innych rozwiązań, niż zakładano początkowo.

10.3. Testy jednostkowe

Z uwagi na charakter pracy inżynierskiej nie było okazji, by zastosować w niej testy akceptacyjne czy też integracyjne. Jest to spowodowane brakiem warstwy interfejsu graficznego. Jedyne testy, jakie zostały wykonane w pracy, są to testy jednostkowe i dotyczą one jedynie implementacji sekwencyjnych poszczególnych algorytmów. Dotyczą tylko ich z uwagi na to, że mają one charakter testów regresyjnych, ponieważ miały ustrzec przed ewentualną zmianą kodu, która powodowałaby niepoprawne działanie algorytmów.

Zważając na fakt, iż implementacje sekwencyjne były wykonane w języku C++, zgodnie z najnowszym standardem C++14, biblioteka, która posłużyła za wykonywanie testów to CxxTest. Wybór padł na nią, z uwagi na jej lekkość i łatwość w użyciu.

Stworzenie testu polega na stworzeniu klasy, która będzie dziedziczyła po innej, którą chcemy testować, jest to zaprezentowane na poniższym fragmencie kodu:

```
class ComplexTest : public CxxTest::TestSuite, public Complex{};
```

Jak widać w celu utworzenia testu, powinno się stworzyć jakąś klasę, która będzie dziedziczyła po TestSuite znajdującej się w zbiorze nazw CxxTest, jak i również po klasie, którą chcemy przetestować. Trzeba mieć na uwadze, że aby przetestować metody, które nie są publiczne powinny one być oznaczone przynajmniej jako protected.

Struktury klasy testowej zawiera dwie metody, które możemy przeładować, są to setUp() oraz tearDown(). Pierwsza z nich zajmuje się inicjalizacją wszystkich potrzebnych tablic, atrybutów, rzeczy potrzebnych do testowania, natomiast tearDown() odpowiada za sprzątanie po wykonaniu testów.

Pojedyncze testy tworzymy jako metody publiczne we wcześniej zadeklarowanej klasie, pamiętając o tym, że każda z metod powinna być bezargumentowa oraz musi zaczynać się od słowa test, tak jak to widać poniżej w poniższym przykładzie.

```
void testMultiply()
{
    Complex result = this->multiply(*second);
    TS_ASSERT_EQUALS(-3.0, result.Real);
    TS_ASSERT_EQUALS(2.0, result.Imaginary);
}
```

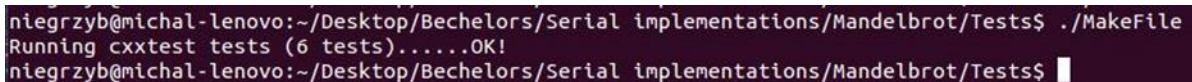
Można też tutaj zauważyć przykłady gwarantów, czyli porównań, wykorzystywanych w każdym znanym frameworku testowym. CxxTest daje nam możliwość porównania wartości na 18 różnych sposobów, aczkolwiek w pracy została użyta tylko część z nich.

Samo uruchomienie testów polega na uprzednim skompilowaniu ich, tak by otrzymać skrypt, który je uruchomi. Służy do tego zwykły kompilator g++. Komenda może wyglądać następująco (przykład dla algorytmu Mandelbrota):

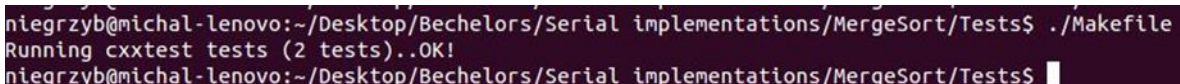
```
cxctestgen --error-printer -o runner.cpp *.h
g++ runner.cpp ../Structures/Complex.cpp ../Structures/mandelbrot.h -std=c++11 -o runner -
I/usr/include/cxctestgen
chmod 777 runner
./runner
rm -f ../Structures/*.o
rm -f runner.cpp
rm -f runner
```

Patrząc po kolei na komendy, dzięki cxctestgen tworzymy swój generator testów wraz z informacją o tym, aby wszelkie błędy były wyświetlane na konsoli. Skrypt uruchomieniowy będzie się nazywał runner.cpp, natomiast wszystkie testy będą znajdowały się w plikach nagłówkowych. Następnie następuje kompilacja wszystkich źródeł testów, oraz plików zależnych kompilatorem g++, potem uruchomienie testów i usunięcie skompilowanych plików.

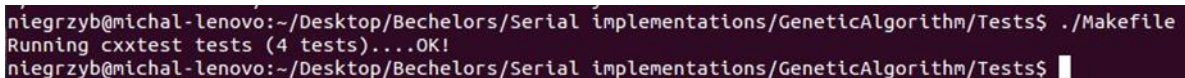
Na rys. 7.1., 7.2. oraz 7.3. zostały przedstawione zrzuty ekranu z wykonanych testów dla wersji sekwencyjnych programów. W kolejności są to algorytm Mandelbrota, klient-serwer, algorytm genetyczny.

A terminal window showing the execution of tests for the Mandelbrot algorithm. The prompt is 'niegrzyb@michal-lenovo:~/Desktop/Bechelors/Serial implementations/Mandelbrot/Tests\$'. The command './Makefile' is entered, followed by the output 'Running cxctest tests (6 tests).....OK!'. The prompt then returns to 'niegrzyb@michal-lenovo:~/Desktop/Bechelors/Serial implementations/Mandelbrot/Tests\$'.

Rys. 7.1. Zrzut ekranu przedstawiający wynik uruchomienia testów dla programu implementującego algorytm Mandelbrota

A terminal window showing the execution of tests for the MergeSort algorithm. The prompt is 'niegrzyb@michal-lenovo:~/Desktop/Bechelors/Serial implementations/MergeSort/Tests\$'. The command './Makefile' is entered, followed by the output 'Running cxctest tests (2 tests)..OK!'. The prompt then returns to 'niegrzyb@michal-lenovo:~/Desktop/Bechelors/Serial implementations/MergeSort/Tests\$'.

Rys. 7.2. Zrzut ekranu przedstawiający wynik uruchomienia testów dla algorytmu MergeSort

A terminal window showing the execution of tests for the GeneticAlgorithm. The prompt is 'niegrzyb@michal-lenovo:~/Desktop/Bechelors/Serial implementations/GeneticAlgorithm/Tests\$'. The command './Makefile' is entered, followed by the output 'Running cxctest tests (4 tests)...OK!'. The prompt then returns to 'niegrzyb@michal-lenovo:~/Desktop/Bechelors/Serial implementations/GeneticAlgorithm/Tests\$'.

Rys. 7.3. Zrzut ekranu przedstawiający wynik uruchomienia testów dla programu implementującego algorytm genetyczny

Oprócz testów jednostkowych, stosowane były również najprostsze możliwe sposoby sprawdzenia czy zaimplementowany algorytm, w każdej z technologii opisanych we wstępie pracy, działa poprawnie. Było to zazwyczaj wykonane za pomocą zwykłych pętli, jeżeli był znany wynik dla jakiegoś zbioru wyjściowego, albo był po prostu porównywany obraz wejściowy z obrazem, który zwracał napisany program.

11. OPIS REALIZACJI PROJEKTU

Projekt realizowano przez ponad pięć miesięcy zgodnie z metodyką zwinną Scrum, jednak zmodyfikowano niektóre jej aspekty w celu dopasowania planu komunikacji w zespole do zajęć na uczelni oraz wykonywanej pracy zawodowej. Stosowana metodyka zakłada częsty kontakt pomiędzy członkami zespołu, jak i klientem, którym w przypadku pisanej pracy, był jej opiekun, doktor habilitowany Paweł Czarnul. W tym rozdziale opisano dokładniej plan komunikacji w zespole oraz jego realizację z wykorzystaniem metodyki Scrum oraz opis przedsięwzięcia, jakim był tworzony projekt i pisana praca.

11.1. *Plan komunikacji w zespole i jego realizacja*

Scrum zakłada, że praca nad pojedynczym wydaniem projektu będzie podzielona na tzw. sprinty. Planowane iteracje, gdyż tak są również one nazywane, powinny być krótkie, trwające około dwóch tygodni. Wszystkie cykle zaczyna się od spotkania, na którym planowane są zadania do wykonania oraz przydziela się je do poszczególnych członków zespołu. Każdego dnia sprintu powinny odbywać się krótkie zebrania, na których każdy członek zespołu opowiada o wykonanej pracy, a także o celach, jakie chce wykonać do następnego krótkiego spotkania, nazywanego stand-up. Wraz z końcem wydania powinno odbyć się spotkanie zwane retrospektywą, na którym członkowie zespołu rozmawiają nad plusami i minusami zastosowanych rozwiązań pracy i komunikacji w zespole, tak, aby praca nad kolejnym wydaniem produktu była przyjemniejsza i efektywniejsza. Dla potrzeb pracy inżynierskiej zdecydowano się zrezygnować z typowych spotkań stand-up, jednak starano się, aby komunikacja w zespole występowała każdego dnia realizacji projektu. Osobiste spotkania zastępowano programami służącymi do komunikacji, takimi jak Skype lub Facebook Messenger. Mimo uproszczenia codziennych zebrań, nie zdecydowano się na ułatwienie dotyczące planowania iteracji. Takie spotkania miały miejsce co dwa tygodnie, na początku każdego sprintu. Rozmawiano na nich o niezrealizowanych zadaniach, które były przypisane podczas ostatniego planowania oraz przydzielano członkom zespołu nowe obowiązki.

Do pracy z metodyką zwinną posłużono się środowiskiem deweloperskim RallyDev, umożliwiającym założenie projektu, podział pracy na iteracje, tworzenie tzw. UserStory, czyli opisów poszczególnych funkcji projektu oraz dodawanie do nich zadań. Tak definiowane obowiązki przypisywano do zarejestrowanych w projekcie użytkowników, czyli autorów pisanej pracy inżynierskiej. Każde zadanie miało swój aktywny status, który mogli śledzić wszyscy członkowie zespołu. Główne stany zadania to zdefiniowane, w trakcie realizacji, zakończone. Praca nad każdym User Story, polegała na realizacji przynajmniej dwóch zadań, implementacji oraz rewizji, gdyż po zakończeniu pracy przez osobę implementującą, inny członek zespołu zobowiązany był do sprawdzenia poprawności kodu oraz zwróceniu uwagi, o ile były one konieczne. W przypadku, gdy zadania zostały zakończone przez obie osoby, a podczas pracy nad innymi funkcjonalnościami został wykryty błąd, środowisko RallyDev pozwalało na utworzenie opisu defektu. Jest to opis zbliżony formą do UserStory, lecz jasno wskazuje, że dana funkcjonalność powinna już działać. Do takiego defektu również w czasie realizacji projektu

przypisywano osobę implementującą, jak i członka zespołu, który sprawdzał poprawność rozwiązania.

W tym miejscu warto zauważyć, że mimo wprowadzenia niewielkich zmian w stosunku do tradycyjnego podejścia metodyki Scrum, starano jak najczęściej spotykać się pełnym zespołem, stąd spotkania odbywały się przynajmniej raz w tygodniu, a pod koniec projektu nawet cztery razy w ciągu siedmiu dni. Najczęstszym miejscem spotkań była uczelnia, gdyż każda z osób realizujących projekt mogła w łatwy sposób dotrzeć na teren Politechniki Gdańskiej. Drugim powodem miejsca spotkań była możliwość korzystania z laboratorium Katedry Architektury Systemów Komputerowych, gdyż realizowany projekt był realizowany na maszynach, do których był możliwy bezpośredni dostęp z komputerów w katedralnej sali.

Do realizacji projektu niezbędne były również narzędzia służące do kontroli wersji pisanego kodu. Ułatwiały one komunikację w zespole dotyczącą rewizji poszczególnych zmian w realizowanym projekcie. Jako takie narzędzia wykorzystano serwis BitBucket, w którym znajdowało się repozytorium projektu oraz GIT, czyli rozproszony system kontroli wersji. Te narzędzia pozwalały na sprawdzenie wprowadzonych zmian w projekcie przez każdego członka zespołu, tym samym ułatwiały komunikację w przypadku znalezionego błędu, gdyż wszystkie osoby realizujące projekt miały dostęp do aktualnego kodu w tym samym momencie. Niezbędny był jedynie dostęp do Internetu, jednak dla autorów pracy nie był to problem. Zatwierdzenie każdej zmiany w projekcie i przesłanie jej do repozytorium wiązało się z dodaniem komentarza do zmiany. Ta funkcjonalność ułatwiała zrozumienie działania prowadzonych modyfikacji.

Kolejnymi narzędziami ułatwiającymi realizację i komunikację w zespole były narzędzia firmy Microsoft, takie jak Word, OneNote czy technologia dysku umieszczonego w chmurze, OneDrive. Ostatnie wersje programu Word umożliwiają pisanie dokumentów wraz z ich natychmiastową synchronizacją i przesłaniem do każdej zainteresowanej osoby. Również podczas pisania, twórcy pracy mają możliwość podejrzenia, kto edytuje lub edytował dokument i jakie zmiany wprowadził. Taka forma realizacji wspólnie pisanego dokumentu wprowadza dużo udogodnień dla osób ją tworzących. Obecna technologia ogranicza trudne łączenie dokumentów spowodowane brakiem porozumienia w zespole. Należy zauważyć, że taka synchronizacja jak i pisanie pracy w przeglądarce korzystając z narzędzia World Online, jest możliwa dzięki technologii Microsoft, OneDrive. Jest to chmura, która umożliwia przetrzymywanie w niej dokumentów i udostępnianie ich wybranym osobom, co znacznie upraszcza komunikację w zespole. W tym rozdziale warto również krótko opisać program OneNote, który okazał się przydatnym narzędziem w realizacji projektu. Jest to aplikacja działająca jak tradycyjny notes, umożliwiająca użytkownikom tworzenie nowych wpisów i edycję już istniejących. Każdy z członków zespołu miał dostęp do notatnika stworzonego na potrzeby realizacji projektu, dzięki temu była możliwość zachowania ważnych informacji podczas tak długo tworzonego projektu.

Powyższy plan komunikacji w zespole i opisane sposoby realizacji znacznie ułatwiły pracę nad tworzeniem projektem inżynierskim. Dzięki częstym spotkaniom unikano nieporozumień, gdyż każda sprawa dotycząca zadań była na bieżąco omawiana, co umożliwiło zakończenie prac zgodnie z zakładanym planem przedstawionym w następnym podrozdziale.

11.2. Plan przedsięwzięcia

Plan realizowanej pracy zakładał wykonanie projektu w pięć miesięcy, zaczynając od lipca 2015 roku. Jednak warto podkreślić, że chęć współpracy z dr hab. Pawłem Czarnulem oraz realizacja pracy z jego pomocą planowana była na długo przed rozpoczęciem prac. W lutym 2015 roku, czyli na początku szóstego semestru autorzy zdecydowali się objąć Indywidualny Program Studiów i zrealizować przedmioty z pierwszego semestru magisterskiego, takie jak Przetwarzanie Równoległe CUDA oraz Systemy Obliczeniowe Wysokiej Wydajności, aby móc zacząć przygotowywać się do realizacji zaplanowanej pracy, gdyż jej temat wykracza poza zakres studiów inżynierskich.

W czasie szóstego semestru, czyli od marca do czerwca 2015 roku zdobyto niezbędne doświadczenie do realizacji projektu. Nauczono się podstaw takich technologii jak OpenMP, MPI, czy OpenCL. W ramach przedmiotu Przetwarzanie równoległe CUDA, zrealizowano projekt prostej aplikacji korzystającej z testowanych technologii. Zdobywanie podstawowych umiejętności zachęciło do obrania tematu pracy inżynierskiej związanej z przetwarzaniem równoległym, dlatego w kwietniu wraz z opiekunem pracy zdefiniowano go oraz zgłoszono.

Po zakończeniu szóstego semestru ustalono plan przedsięwzięcia. Lipiec przeznaczono na czytanie literatury dotyczącej przetwarzania równoległego oraz wykorzystaniu takiego modelu obliczeń na koprocesorze Intel Xeon Phi. W następnym miesiącu, sierpniu, uzgodniono z opiekunem rodzaj implementowanych algorytmów i zakres pracy, po czym przystąpiono do planowania wydania produktu i najbliższej iteracji. W sierpniu w czasie dwóch sprintów napisano pierwszy testowany program, czyli generowanie obrazu Mandelbrota, w czterech wersjach, szeregowej oraz z wykorzystaniem MPI, OpenMP i OpenCL. We wrześniu zaczęto pracę nad kolejnym programem, który korzystając z algorytmu genetycznego generuje obraz złożony z prostokątów na podstawie pliku wejściowego. Realizacja tego algorytmu okazała się trudniejsza niż zakładano, stąd jego realizacja wykroczyła poza dwie iteracje i zakończyła się pod koniec października. Następnie poświęcono jedną iterację na realizację ostatniego programu, symulującego działanie klient-serwer. Należy zauważyć, że jego implementacja nie sprawiła większych problemów i została zakończona zgodnie z planem. Po zaimplementowaniu algorytmów w każdej z technologii i poprawieniu wykrytych błędów przystąpiono do pomiarów czasu wykonania każdego z nich oraz pisania pracy. Ten etap realizowano głównie w listopadzie. Pod koniec tego miesiąca oraz na początku grudnia zakończono etap pisania pracy wraz z wprowadzonymi wszystkimi niezbędnymi zmianami edycyjnymi.

Powyższy rozdział pokazuje, iż sumienność i zaangażowanie w pracę nad projektem zaowocowało zakończeniem prac zgodnie z zakładanym planem. Pozwoliło to również na zaprezentowanie całości pracy przed opiekunem i recenzentem przed ostatecznym terminem złożenia dokumentów w dziekanacie wydziału.

12. ZEBRANE DOŚWIADCZENIA

Każdy projekt inżynierski realizowany jest w grupach, spowodowane jest to faktem, iż informatycy w późniejszej pracy zawodowej również będą pracować i tworzyć oprogramowanie w zespołach. Niniejsza praca została wykonana przez trzech autorów Błażeja Galińskiego, Łukasza Romanowskiego oraz Michała Niegrzybowskiego, którzy wspólnie studiują na profilu inżynierskim Katedry Architektury Systemów Komputerowych. W tym rozdziale przedstawiono doświadczenia wymienionych osób związane z pracą w zespole, użytymi narzędziami, jak i wykorzystanymi technologiami.

12.1. *Praca w zespole*

Już na początku warto wspomnieć o tym, jak duże korzyści przynosi praca w zespole. Podczas realizacji skomplikowanego tematu, każda z osób zdobywała wiedzę z innego zakresu. Początkowo podział został ustalony według technologii, co okazało się dobrym pomysłem, gdyż w dalszej części realizowanego projektu, podczas pisania kodu oraz szukania błędów owocowało to wymianą doświadczeń i uwag na temat testowanych technologii.

Ważna wymiana umiejętności dotyczyła również pracy z wykorzystywanymi narzędziami i technologiami. Wystarczyło, że jedna z osób w zespole sprawdziła, jak korzystać ze skomplikowanych aplikacji do pomiarów poszczególnych części uruchamianych programów, by przekazać tę wiedzę i nauczyć w krótszym czasie innych członków zespołu, którzy równolegle pracowali nad resztą realizowanego projektu. Dzięki rozłożeniu prac na poszczególnych autorów realizacja przebiegała dużo szybciej, niż gdyby ten sam projekt realizować miałyby jedna osoba.

Członkowie zespołu mogli zdobyć cenne doświadczenie nie tylko z uwagi na wymianę zdobytej wiedzy, ale również ze względu na rozwój cech związanych z komunikacją międzyludzką. Porozumiewanie się z wykorzystaniem narzędzi, wymienionych w rozdziale 10. Opis prac implementacyjnych, odbywało się zgodnie z oczekiwaniami. Nie występowały zakłócenia w zrozumieniu się, dlatego, że żadna z osób biorących udział w projekcie nie unikała rozmowy i dyskusji na temat wykonywanej pracy. W tym akapicie warto również zauważyć, że praca w zespole uczy dostosowywania się do innych osób oraz znalezienia kompromisu wpływającego na tempo realizacji, a także na znalezienie wolnych terminów dla wspólnych spotkań.

Ciekawym spostrzeżeniem jest również fakt, iż komunikacja w zespole była wystarczająco zadowalająca oraz zdobyte doświadczenie dostatecznie duże, aby autorzy pracy zaplanowali również wspólną realizację projektu magisterskiego. Ma on łączyć moduły, które zaimplementowane zostaną przez pojedynczych członków zespołu wykonującego niniejszą pracę inżynierską.

Podsumowując, praca w grupie dla każdego z autorów projektu przyniosła wymierne efekty. Członkowie drużyny dopełniali się pod wieloma względami, dzięki czemu realizacja przebiegała sprawnie i bez większych problemów. Tak zgrana komunikacja doprowadziła do zakończenia wspólnie wykonanego projektu inżynierskiego oraz napisania wyczerpującej pracy na jego podstawie.

12.2. Metodyka i system pracy

Cenne doświadczenie zdobyte podczas realizacji projektu opierało się również na stosowanej metodyce i systemie pracy. Techniką w zakresie, której zdobyto cenne umiejętności był Scrum, czyli metodyka zwinna. Głównymi jej założeniami są częste spotkania oraz regularna komunikacja z klientem, którym w przypadku pisanej pracy był jej opiekun dr hab. Paweł Czarnul. Dokładnie procedury rozwoju oprogramowania w tej technice zostały opisane w podrozdziale - Plan komunikacji w zespole i jego realizacja - znajdującego się w rozdziale 11., gdzie skupiono się na zebranych doświadczeniach związanych z pracą w metodyce Scrum.

Warto zauważyć, że ta technika wytwarzania oprogramowania jest obecnie stosowana w większości firm, również autorzy projektu mieli z nią styczność w miejscu pracy, stąd każdy wniósł inne doświadczenie. Zebrana praktyka pozwoliła również na dobór narzędzia służącego do monitorowania obranego systemu pracy. Praca z RallyDev, gdyż tak nazywa się ten serwis, pozwoliła na zapoznanie się z całym cyklem wytwarzania oprogramowania, począwszy od fazy zbierania wiedzy, poprzez implementację, rewizje i testy, aż do pisania dokumentacji na podstawie stworzonego produktu. Poznano również sposoby monitorowania postępów w poszczególnych iteracjach, tak aby móc odpowiednio poradzić sobie z zadaniami, które nie zdążyły zostać zrealizowane.

System realizowanej pracy oparty był również o repozytoria i oprogramowania kontroli wersji. Do pracy nad kodem podczas realizacji projektu wykorzystywano środowisko GIT, którego obsługa początkowo sprawia dużo problemów, szczególnie uciążliwa jest operacja scalania dwóch różniących się wersji plików. Czynność ta wykonywana w terminalu sprawiała na tyle dużo trudności, że zdecydowano, iż zatwierdzanie i przesyłanie zmian w plikach, będzie prowadzone w zintegrowanym narzędziu, które zostało opisane we wcześniejszej grupie, VisualCode. Warto jednak zauważyć, że gdy nie ma konieczności łączenia plików, praca z gitem w terminalu sprawia o wiele mniej problemów. Jednak w przypadku chęci zatwierdzenia zmian i przesłania plików na serwer podczas pracy w terminalu na zdalnej maszynie, nie ma innej możliwości, jak praca z podstawowymi komendami GIT.

W tym miejscu warto również opisać doświadczenia związane z pracą z serwisem BitBucket, który umożliwia tworzenie prywatnych repozytoriów. Bardzo łatwo integruje się go ze środowiskiem kontroli wersji, jakim jest GIT. BitBucket umożliwia śledzenie zmian w projekcie z poziomu przeglądarki, a także zmian w kodzie, choć należy zaznaczyć, że nie łatwo wprowadza się takie modyfikacje. Podczas pracy, taki sposób wprowadzania zmian wykorzystywano jedynie podczas niewielkich poprawek. Bardzo przydatną funkcją jest również dzielenie się informacjami za pomocą głównej tablicy projektu, na której umieszczono wszystkie niezbędne informacje na temat zintegrowania środowiska i rozpoczęcia realizacji projektu inżynierskiego. Nie korzystano z możliwości rewizji w serwisie, te prowadzono zazwyczaj lokalnie, a uwagi przekazywano poprzez komunikatory oraz podczas rozmowy osobistej.

Do pracy nad pisanem dokumentu oraz do zapisywania notatek, tak, aby były od razu dostępne dla innych autorów, korzystano z narzędzia OneDrive firmy Microsoft. Każdy członek grupy początkowo sceptycznie nastawiony do takiego dzielenia się plikami, szybko przekonał się,

że jest to niezwykle proste i przydatne narzędzie. Przekazywanie dokumentów tekstowych umożliwiało ich natychmiastowe sprawdzenie przez innych autorów pracy inżynierskiej, tak aby zachować spójność pisanych prac. Należy jednak zauważyć, że choć lokalnie zainstalowane narzędzie OneDrive z funkcją automatycznej synchronizacji udostępnionych dokumentów działało bez zarzutów, tak edycja dokumentów z poziomu przeglądarki nie działała już na tyle dobrze. Podczas tworzenia pracy zdarzało się, iż otwarcie dokumentu, w taki sposób powodowało usunięcie części akapitów, co było trudne do zauważenia przez osobę edytującą. Z tego powodu w celu uniknięcia problemów, tworzono również kopie lokalne dokumentów, które nie ulegały automatycznej synchronizacji.

Z pełnym przekonaniem można stwierdzić, iż zdobyta wiedza związana z realizowaną metodyką przyniesie korzyść autorom projektu, gdyż obecnie ten system pracy jest powszechnie stosowany, a wykorzystywane narzędzie jest jednym z najlepszych w swojej dziedzinie.

12.3. Narzędzia

Podczas realizacji projektu korzystano z wielu narzędzi, które można podzielić na kilka grup. W tym rozdziale opisano doświadczenia i uwagi odnośnie pracy z nimi.

Pierwszą grupą jaką można wyróżnić, są narzędzia pomocne przy pisaniu kodu, takie jak CLion, Visual Studio Code lub VIM. Pierwszy z nich jako narzędzie komercyjne udostępnia szereg ułatwień związanych z pisaniem kodu, stąd raczej pozytywne doświadczenia związane z pracą w tym narzędziu. Głównie dzięki możliwości łatwego poruszania się po plikach utworzonego projektu, realizacja pracy przebiegała szybko, a powstały kod był czytelny i późniejsza jego refaktoryzacja była dużo prostsza.

Innym narzędziem, przy pomocy którego pisano programy, jest Visual Studio Code. Zaletami tego środowiska są prostota oraz łatwe zintegrowanie z oprogramowaniem GIT, jednak podczas tej operacji można napotkać często niezrozumiałe błędy, których rozwiązania trudno jest znaleźć w Internecie. Jednak po poprawnym skonfigurowaniu proces commit zmienionych plików repozytorium, czyli lokalnego zatwierdzania zmian, przebiega dużo sprawniej niż przy użyciu konsoli.

Programy realizowane w projekcie uruchamiane były na maszynach apl12, do których łączono się z pomocą konsoli na komputerze personalnym. Chcąc pisać kod bez konieczności przesyłania go, używano często edytora konsolowego VIM. Mimo, że znany jest on z trudnej obsługi, kilka prób z tym narzędziem pozwoliło oswoić się ze skomplikowanymi skrótami. Należy przyznać, że po wstępnej nauce praca z programem okazała się wyjątkowo przyjemna. Fakt, iż korzysta on z terminalu, powoduje, że działa szybko i można z niego korzystać podczas zdalnych połączeń.

Kolejną grupę narzędzi reprezentuje program Intel VTune Amplifier XE 2016, wykorzystywany do monitorowania działań napisanych programów. W kontekście pracy z tym narzędziem należy rozpatrzyć dwie sprawy. Po pierwsze, etap wykonywania rzutu z pracy programu został wykonany przy pomocy starszej wersji programu, która jako jedyna była dostępna na maszynie, w której znajdowały się obie karty Intel Xeon Phi. Praca ta została

wykonana przy użyciu graficznego trybu pracy programu, co ze względu na brak fizycznego dostępu do maszyny, a co za tym idzie, konieczność używania połączenia przy wykorzystaniu SSH, było zadaniem bardzo czasochłonnym, ze względu na małą wydajność takiego połączenia. Sposób ten został wykorzystany głównie ze względu na popularność tego rozwiązania, a co za tym idzie, łatwiej dostępną dokumentację. Z kolei sama analiza zrzutu została wykonana na własnych komputerach, ze względu na wspomnianą wcześniej wydajność i responsywność aplikacji w trybie pracy lokalnej, co było kluczowe ze względu na ilość prezentowanych przez nią parametrów, w każdym udostępnionym widoku analizy. Aplikacja ta umożliwiła, w jasny sposób, na wgląd w szereg metryk, co umożliwiło rozwiązanie wielu problemów napotkanych w trakcie implementacji algorytmów. Pozwoliło to również na dogłębne zapoznanie się z funkcjami, które realizowane były w ramach zastosowanych technologii. Umożliwiło to również, w jasny sposób, na wskazanie różnic pomiędzy wybranymi technologiami oraz na wyjaśnienie specyficznych zachowań, dla danej technologii, przez co możliwe było wyciągnięcie wielu przydatnych wniosków. Jednakże trzeba zaznaczyć, że początkowe doświadczenia z obsługą aplikacji były trudne, gdyż program ten skierowany jest do osób wyłącznie zaznajomionych z tematem systemów wysokiej wydajności, ponieważ jest to jedno z ważniejszych narzędzi ich pracy, które wykorzystywane jest do wyszukiwania powodów zmniejszonej wydajności programów

Ostatnią grupą, którą można wyróżnić były narzędzia wykorzystane podczas prezentowania wyników pomiarów. W tym zestawieniu znajduje się jedynie środowisko programistyczne jakim jest Node.js. Pozwoliło ono na szybką wizualizację czasów wykonanych testów na trójwymiarowych wykresach. Sposób implementacji warstwy prezentacji został opisany w rozdziale 10. Opis prac implementacyjnych. W przypadku opisu doświadczeń związanych z tym narzędziem, warto wspomnieć jedynie, że praca ze środowiskiem nie sprawiała problemów, a jej efekt był zadowalający, gdyż na generowanych wykresach łatwo można dowieść skalowalność testowanych technologii.

Instalacja środowiska oraz jego użycie początkowo może sprawiać kilka problemów, jednak znalezienie ich rozwiązania w Internecie jest bardzo proste. Również nie ma problemów ze znalezieniem instrukcji implementacji, co pokazuje, że tą technologią interesuje się wiele osób i na pewno szybko nie przestanie być wykorzystywana na rynku komercyjnym.

Podczas realizacji pracy wykorzystano wiele narzędzi, które w różny sposób spełniły oczekiwania, jednak każde z nich wykonywało prawidłowo swoją funkcję. Dzięki pracy z nimi zdobyto wiele umiejętności, które z racji, iż są to narzędzia powszechnie używane na rynku, są niezwykle ważne z uwagi na wykorzystanie ich w pracy zawodowej.

12.4. Testowane technologie

W tym rozdziale opisano doświadczenia związane z pisanie programów w każdej z testowanych technologii. Wzięto pod uwagę rozbudowanie dokumentacji, łatwość zastosowania oraz ilość rozwiązań napotkanych błędów, które można znaleźć w literaturze i Internecie.

12.4.1. MPI

Praca z technologią Message Passing Interface dostarczyła różnych doświadczeń. Na wstępie należy zauważyć, że przed rozpoczęciem realizacji projektu inżynierskiego autorzy zdobyli już podstawowe umiejętności w pracy z tą technologią na przedmiocie Systemy Obliczeniowe Wysokiej Wydajności, prowadzonego na semestrze 1. studiów magisterskich. Przedmiot ten, jak już wcześniej opisano, został realizowany w ramach Indywidualnego Programu Studiów.

Opis doświadczeń warto rozpocząć od charakterystyki dokumentacji zastosowanej technologii. Korzystano z www.mpich.org, gdzie znajduje się wiele przewodników dotyczących programowania z użycie opisywanej technologii, a także wyjaśnienia możliwych do użycia funkcji. Jest ona prosta w posługiwaniu się, a opisy poszczególnych metod są krótkie i rzeczowe, łatwo również je znaleźć. Być może przydatne byłyby również przykłady użycia, jednak te pominięto przy opisach działania funkcji oraz jej parametrów, jednak mimo ich braku, pozytywnie oceniono tę część charakterystyki doświadczeń związanych z pracą z MPI.

Należy również opisać łatwość implementacji. Przy zastosowaniu podstawowych funkcji, komunikacji synchronicznej oraz zrozumieniu podstawowego modelu, tworzenie programów z wykorzystaniem technologii MPI nie sprawia problemów. Trudniej jest przy asynchronicznym przesyłaniu paczek o różnej wielkości złożonych ze skomplikowanych typów. Taki sposób pisania programów sprawiał wiele problemów związanych z prawidłowym korzystaniem z pamięci alokowanej dynamicznie. Należało dokładnie wiedzieć skąd, dokąd i jakie dane są przekazywane, aby zmniejszyć prawdopodobieństwo powstania błędów. Zdarzało się, że szybciej i prościej było napisać program od nowa, niż szukać błędu w już napisanym. Kontrola nad pracą każdego procesu daje dużą swobodę programiście, jednak jak wynika z doświadczenia, często może sprawiać problem, szczególnie przy użyciu w skomplikowanych programach. Jednak należy zauważyć, że wszystkie napotkane błędy rozwiązano, co pokazuje, iż mimo, czasem ciężkiego znajdowania błędów, korzystając z tej technologii można rozwiązać większość stawianych celów.

Ostatnią porównywaną cechą jest łatwość w znajdowaniu rozwiązań napotkanych błędów i problemów w Internecie oraz literaturze. Podczas pisania pracy w opisywanej technologii napotkano dużą ilość trudności, jednak jest wiele przewodników i porad w sieci, dzięki którym można w łatwy sposób sobie z nimi poradzić. Wynika to z faktu, iż MPI jest popularnym i często stosowanym standardem, a w kontekście opisywanej pracy, w literaturze można znaleźć wiele rozwiązań związanych z zastosowaniem Message Passing Interface bezpośrednio na koprocesorze Xeon Phi.

Praca z technologią MPI, mimo napotkanych błędów, była niezwykle przydatnym doświadczeniem. Rozwiązywanie napotkanych błędów oraz dostosowanie problemu pod powszechnie stosowany model master-slave wiele nauczyło autorów pracy, a wiedzę tę można rozwijać w rozwiązywaniu bardziej skomplikowanych zagadnień.

12.4.2. OpenMP

Jest to obecnie jedna z najszybciej i najlepiej rozwijanych technologii. W kontekście pisanej pracy warto zauważyć, iż jest ona bardzo często używana w programowaniu równoległym z wykorzystaniem akceleratorów Intel Xeon Phi. Uzasadnia to chęć zdobycia doświadczenia związanego z wykorzystaniem tej OpenMP w praktyce.

Dokumentacja opisywanej technologii znajduje się na stronie www.openmp.org. Można znaleźć tam wiele opisów jak korzystać z funkcji OpenMP oraz odpowiedzi na pytania jak radzić sobie z problemami. Jednak podczas nauki OpenMP korzystano głównie z literatury i instrukcji znalezionych w Internecie, niekoniecznie na oficjalnej stronie testowanej technologii.

Warto jednak napisać o samej specyfice biblioteki, gdyż użycie jej jest z pewnością najłatwiejsze spośród opisywanych technologii. Nie ma potrzeby dbania o to w jaki sposób przebiega zrównoleglanie kodu, wystarczy użyć prostych dyrektyw, aby wszystko zostało zaplanowane i poprawnie zakończone bez wiedzy osoby implementującej. Wpływa to na fakt, iż programy realizowane w technologii OpenMP, były zakańczane zawsze wcześniej od innych. Należy tutaj również wspomnieć o doświadczeniu związanym z wykorzystaniem techniki offload, która umożliwia przesyłanie danych do obliczeń z hosta na koprocessor Intel Xeon Phi. Również praktyka z tym związana nie sprawiała problemów, gdyż można znaleźć wiele szczegółowych opisów jak korzystać z wymienionych dyrektyw.

Wiele pozycji literatury opisuje działanie OpenMP, nawet przy zastosowaniu z koprocessorami Xeon Phi. Ponad to, w przypadku napotkanych problemów, np. z rozwiązaniem asynchronicznej pracy w algorytmie symulującym komunikację klienta z serwerem skorzystano z forum firmy Intel [8], gdzie zadano pytanie o rozwiązanie dla napotkanego utrudnienia. Odpowiedź uzyskano szybko, jednak zaimplementowanie planowanego modelu wiązało się z wykorzystaniem techniki Stream, której z uwagi na ograniczenia opisane w rozdziale 10. Opis prac implementacyjnych nie było możliwości użyć. Mimo to, doświadczenie pokazuje, iż w przypadku problemów związanych z opisywaną technologią jest wiele osób chętnych pomóc.

Powyższy opis jasno wskazuje, iż technologia OpenMP dzięki nowoczesnym rozwiązaniom firmy Intel będzie cały czas mocno wspierana, a co za tym idzie wykorzystywana przez programistów. Nie jest to nic dziwnego, zważając na fakt łatwości implementacji programów przy wykorzystaniu tej technologii i liczby dostępnej literatury na jej temat.

12.4.3. OpenCL

OpenCL jest to technologia, która rozwija się najszybciej, a w jej ulepszaniu biorą udział w chwili obecnej największe firmy na świecie jak na przykład Intel, AMD czy nVidia. Każdy z autorów pracy miał doświadczenie z OpenCL z przedmiotu przetwarzanie równoległe CUDA, który, jak już wcześniej zostało wspomniane, realizowany był w czasie Indywidualnego Programu Nauczania. Na zajęciach twórcy realizowali projekty właśnie w tej technologii. Autorzy zdecydowali się skorzystać z nakładki C++ do OpenCL tworzoną przez twórcę API OpenCL, czyli firmę Khronos. Z uwagi na to, że pozwala on pisać kod hosta zgodnie ze standardem C++14

jak i również dzięki temu, że posiada bardzo dobrą dokumentację, która znajduje się na stronie firmy Khronos.

Model programowania w OpenCL jest najbliższy językom wysokopoziomowym, z którymi twórcy pracują na co dzień, więc był to dla nich najbardziej naturalny język ze wszystkich. Oprócz plusów OpenCL posiada też swoje minusy, wszystkie implementacje w tej technologii były o wiele bardziej rozległe od analogicznych w MPI i OpenMP z uwagi na to, że całe ustawienie środowiska musi być zrealizowane w kodzie hosta. Chodzi między innymi o ustawienie kontekstu, stworzenie kolejek, wybór urządzeń na jakich będzie uruchamiany program. Minusami w pisaniu programów OpenCL była również przestarzała technologia, która powodowała ograniczenia opisane w rozdziale 10. Opis prac implementacyjnych.

Oprócz problemów związanych ze starymi bibliotekami znajdującymi się na urządzeniu, występowały również problemy natury czysto implementacyjnej. Aczkolwiek z uwagi na bogate zbiory literatury o OpenCL, w tym również jeden polski tytuł [4] oraz dużą ilość tematów o OpenCL w Internecie, w razie problemów była łatwa możliwość znalezienia ich rozwiązania.

Praca z OpenCL mimo pewnych niedogodności, była na pewno bardzo ciekawym przeżyciem dla każdego z autorów, tym bardziej, że każdy z nich wiąże swoją przyszłość naukową z tą szybko rozwijającą się technologią.

13. PODSUMOWANIE

W odniesieniu do tematu pracy inżynierskiej można stwierdzić, że jej cel, czyli porównanie technologii równoległych na koprocesorze Intel Xeon Phi, został osiągnięty. Warto podkreślić, iż zastosowano model offload, w którym proces główny oddelegowuje wszystkie obliczenia na akcelerator. W ramach działań zostały zaimplementowane trzy algorytmy w każdej z testowanych technologii Message Passing Interface, Open Multi-Processing, Open Computing Language oraz w wersji szeregowej, która stanowiła podstawę do dalszych działań. Korzystając ze stworzonych implementacji dokonano pomiarów, zazwyczaj w zależności od wielkości problemu oraz wykorzystanych jednostek obliczeniowych. Na ich podstawie dokonano analizy technologii, biorąc pod uwagę testowany obszar problemu. Wśród nich wyróżniono problem dużej ilości obliczeń, pracy z mnogą liczbą danych, oraz częstej komunikacji. Wymienione kwestie odzwierciedlają kolejno zaimplementowane algorytmy: zobrazowanie zbioru Mandelbrota, tworzenie obrazu złożonego z prostokątów na podstawie obrazu wejściowego, z wykorzystaniem algorytmu genetycznego oraz aplikacji typu klient-serwer, gdzie przesyłane żądania były przetwarzane na koprocesorze. Pozwoliło to na osiągnięcie celu pracy poprzez wyszczególnienie cech charakterystycznych technologii w stworzonych implementacjach.

Wyróżniono takie aspekty jak skalowalność, czas komunikacji, narzut czasowy związany z wykorzystaniem metod danej technologii. Dzięki tym parametrom możliwe było określenie przydatności technologii w zależności od problemu. Podczas badań korzystano z zarówno ze zmierzonych rezultatów, jak i analiz, przeprowadzonych dzięki programowi Intel Vtune Amplifier, który pozwalał na stworzenie zapisu przebiegu uruchomionego programu.

Z przeprowadzonych badań wynika, że dla problemu dużej ilości obliczeń, wyniki są zależne od jego wielkości. Dla generowania małych obrazów, o boku do 8 tysięcy pikseli, najlepszą technologią okazał się MPI, z uwagi na najniższy narzut czasowy związany z inicjalizacją środowiska. Dla większych rozmiarów problemu, najlepszy okazał się Open Computing Language. Spowodowane było to faktem, iż dzięki implementacji w technologii Threading Building Blocks, program ten, charakteryzuje bardziej równomierne rozłożenie obciążenia pomiędzy dostępne rdzenie.

Dla kolejnego problemu, jakim była aplikacja typu klient-serwer odzwierciedlająca problem dużej liczby komunikacji pomiędzy hostem, a koprocesorem, najlepszą był MPI. Wynika to z nastawienia tej technologii na komunikację pomiędzy osobnymi procesami. Jednak przy porównaniu nie była brana porównywana technologia Open Multi-Processing, z uwagi na brak możliwości skorzystania z dyrektyw typu Stream, co uniemożliwiło poprawne zaimplementowanie założonego modelu komunikacji.

W ostatnim testowanym programie, przedstawiającym problem obliczeń na dużej liczbie danych, najlepszy okazał się Open Computing Language. Również tak, jak w przypadku algorytmu Mandelbrota, było to spowodowane implementacją w technologii Threading Building Blocks. Należy jednak zauważyć, że dla małego rozmiaru problemu, czyli dla populacji o rozmiarze do 250 osobników oraz liczbie wątków równej 16, szybsza była technologia Open

Multi-Processing. Wynika to braku narzuty czasowego związanego z dystrybucją zadań przez dodatkowe technologie.

Z przeprowadzanych analiz, wynika, iż technologia Open Computing Language jest najbardziej uniwersalna, gdyż dla wszystkich testowanych obszarów osiągała dobre wyniki, przy czym dla dwóch z testowanych problemów była najlepsza. Spowodowane jest głównie jej implementacją opierającą się na Intel Threading Building Blocks, która polega na dystrybucji zadań na stałą liczbę wątków na koprocesorze.

Warto zauważyć, że powyższe rezultaty mogą stanowić podstawę do dalszych działań. W przyszłości należałoby skupić się na optymalizacjach związanych z wektoryzacją, na które podczas realizacji projektu zabrakło czasu. Również warto byłoby wprowadzić nakładanie się obliczeń i komunikacji do implementacji w technologiach Open Multi-Processing oraz Open Computing Language, aby, pod tym względem, były one wspólne z Message Passing Interface. Kolejnym aspektem, który można byłoby rozważyć, jest implementacja z wykorzystaniem Intel Threading Building Blocks programów napisanych w Message Passing Interface oraz Open Multi-Processing, tak, aby można było porównać je pod tym względem z Open Computing Language. Najważniejszą jednak rzeczą, którą potencjalnie może przynieść największe korzyści w kontekście porównania, jest zastosowanie nowszych bibliotek testowanych technologii, gdyż podczas realizacji projektu użyto starszych, niezaktualizowanych wersji, które obecnie są zainstalowane na maszynie apl12, z której korzystano podczas pracy.

W pracy opisano doświadczenia związane z wykorzystaniem każdej z technologii, co również było jej celem. Z praktyki związanej z wykorzystaniem opisywanych bibliotek, wynika, iż najłatwiej stworzyć zrównoleglony kod w Open Multi-Processing, gdyż wystarczy wykorzystać pojedynczą dyrektywę, wpływa to również na łatwość zrozumienia modelu przez programistę. Należy zauważyć, że technologia Message Passing Interface, wykorzystująca do pracy procesy, umożliwia korzystanie z pamięci rozproszonej. Daje ona możliwość programiście zarządzania obciążeniem poszczególnych węzłów, jednak wprowadza związane z tym trudności, synchronizacji procesów, komunikacji oraz przesyłania danych z uwagi na wspomnianą pamięć. Natomiast technologia Open Computing Language, poprzez swoją heterogeniczność wprowadza trudności z koniecznością zarządzania platformami, kontekstem i kolejkami, czyli konstrukcjami charakterystycznymi dla tej technologii.

Na końcu należy podkreślić, iż po zastosowaniu wybranych z powyższych optymalizacji, realizowana praca inżynierska mogłaby zostać wykorzystana jako publikacja naukowa. Pozwala na to zrealizowany cel, gdyż już teraz osiągnięte rezultaty są zgodne z teoretycznymi założeniami każdej przetestowanej technologii.

WYKAZ LITERATURY

1. Rahman R.: Intel Xeon Phi Coprocessor Architecture And Tools: The Guide for Application Developers. Apress Media LLC, New York 2013., ISBN 978-1-4302-5926-8.
2. Supalov A., Semin A., Klemm M., Dahnken C.: Optimizing HPC applications with Intel Cluster Tools, Apress Media LLC, New York 2014., ISBN 978-1-4302-6496-5.
3. Chapman B., Jost G., van der Pas R.: Using OpenMP: Portable shared memory parallel programming, The MIT Press, Cambridge 2008., ISBN 978-0-262-53302-7.
4. Sawerwain M.: OpenCL – akceleracja GPU w praktyce, PWN S.A., Warszawa 2014, ISBN 978-83-01-18012-6.
5. Vladimirov A., Asai R., Karpusenko V.: Parallel Programming and Optimization With Intel Xeon Phi Coprocessors, 2nd Edition, COLFAX 2015, ISBN 978-0-9885234-0-1.
6. Rościszewski P.: Dynamic Data Management among multiple databases for optimization of parallel computations in heterogeneous HPC systems, Meghanathan N., Nagamalai D. (Eds.), ICCSEA2014, Vol. 4 of Computer Science & Information Technology, AIRCC, 2014.
7. StackOverflow, <http://stackoverflow.com/questions/33462355/kernel-doesnt-wait-for-events>, (data dostępu 28.11.2015 r.).
8. Intel Forum, <https://software.intel.com/en-us/comment/1845793#comment-1845793>, (data dostępu 28.11.2015 r.).
9. OpenMP, <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>, (data dostępu 28.11.2015 r.).
10. Khronos, <https://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.2.pdf>, (data dostępu 28.11.2015 r.).
11. Intel, <https://software.intel.com/en-us/node/522691>, (data dostępu 28.11.2015 r.).
12. Intel, <https://software.intel.com/en-us/articles/openmp-thread-affinity-control>, (data dostępu 28.11.2015 r.).
13. MPI Forum, <http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>, (data dostępu 28.11.2015 r.).
14. Khronos, <https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>, (data dostępu 28.11.2015 r.).
15. Banger R., Bhattacharyya K.: OpenCL Programming by Example, PACKT, 2013, ISBN 9781849692342.
16. nVidia, http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, (data dostępu 28.11.2015 r.).
17. dr hab. inż. Paweł Czarnul, <http://fox.eti.pg.gda.pl/~pczarnul/cuda/slides10.pdf>, (data dostępu 28.11.2015 r.).
18. Pacheco P.: An Introduction to Parallel Programming, Morgan Kaufmann, 2011, ISBN 9780123742605.
19. MPICH, <http://www.mpich.org/>, (data dostępu 28.11.2015 r.).
20. Andreinc, <http://andreinc.net/2010/12/26/bottom-up-merge-sort-non-recursive/>, (data dostępu 28.11.2015 r.).

WYKAZ RYSUNKÓW

2.1. Obszary problemów algorytmicznych w przetwarzaniu równoległym	13
3.1. Wizualizacja dwukierunkowej topologii	17
3.2. Przedstawienie ringów na koprocessorze Xeon Phi	17
3.3. Rozwinięcie skrótów: RD – odczyt, WR – zapis, WB – zapis zwrotny do GDDR (Ram), RFO – odczyt tylko dla właściciela	18
3.4. Stany linii pamięci w pamięci podręcznej poziomu L2 z udziałem Tag Directory	19
3.5. Core i Unicore	19
3.6. Ewolucja architektury w stronę Manycore. Rozwinięcie skrótów C – pamięć podręczna, MC – kontroler pamięci, P _x – rdzeń procesora	20
4.1. Podział wątków dzięki Thread Affinity przy użyciu typu compact	25
4.2. Podział wątków dzięki Thread Affinity przy użyciu typu scatter	25
4.3. Praca z wykorzystaniem trybu offload	26
5.1. Przedstawienie zaimplementowanej komunikacji w aplikacji symulującej działanie klient-serwer	32
6.1. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii Message Passing Interface dla algorytmu z dużą liczbą prostych obliczeń	75
6.2. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenMP dla algorytmu z dużą liczbą prostych obliczeń	77
6.3. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenCL dla algorytmu z dużą liczbą prostych obliczeń	78
6.4. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii Message Passing Interface dla algorytmu z dużą liczbą komunikacji	80
6.5. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenMP dla algorytmu z dużą liczbą komunikacji	81
6.6. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenCL dla algorytmu z dużą liczbą komunikacji	83
6.7. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii MPI dla algorytmu z dużą liczbą obliczeń na dużej ilości danych	84
6.8. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenMP dla algorytmu z dużą liczbą obliczeń na dużej ilości danych	85
6.9. Wykres zależności czasu, rozmiaru oraz jednostek liczących technologii OpenCL dla algorytmu z dużą liczbą obliczeń na dużej ilości danych	86
7.1. Zrzut ekranu przedstawiający wynik uruchomienia testów dla programu implementującego algorytm Mandelbrota	92
7.2. Zrzut ekranu przedstawiający wynik uruchomienia testów dla algorytmu MergeSort	92
7.3. Zrzut ekranu przedstawiający wynik uruchomienia testów dla programu implementującego algorytm genetyczny	92

WYKAZ TABEL

1.1. Pogrupowanie pamięci w OpenCL ze względu na ich cechy pogrupowaniu	27
---	----

Dodatek A: Czasy wykonania programu realizującego algorytm genetyczny (algorytm)

Liczba procesów	Technologie													
	Rozmiar populacji	MPI					OpenCL					OpenMP		
		Czas algorytmu (ns)	Czas generacji (ns)	Czas oceny (ns)	Czas mutacji (ns)	Czas algorytmu (ns)	Czas generacji (ns)	Czas oceny (ns)	Czas mutacji (ns)	Czas algorytmu (ns)	Czas generacji (ns)	Czas oceny (ns)	Czas mutacji (ns)	
16	250	6,01541E+11	1,19822E+11	1,19807E+11	15942000	6,87638E+11	1,38353E+11	1,3833E+11	1,3833E+11	6,0285E+11	1,2028E+11	1,1928E+11	980044000	
16	500	1,19895E+12	2,40802E+11	2,40765E+11	37137000	1,36274E+12	2,8054E+11	2,8048E+11	2,8048E+11	1,21913E+12	2,42677E+11	2,41146E+11	1522179000	
16	1000	2,39889E+12	4,7867E+11	4,79589E+11	70210000	1,1021E+12	1,61603E+11	1,61489E+11	1,61489E+11	2,38239E+12	4,74022E+11	4,71187E+11	2866645000	
16	2000	4,72365E+12	9,45307E+11	9,45168E+11	140473000	6,18488E+12	1,3105E+12	1,31014E+12	1,31014E+12	4,71943E+12	9,43292E+11	9,36204E+11	7101046000	
32	250	3,12073E+11	62636835000	62629605000	7248000	6,99208E+11	1,3839E+11	1,38356E+11	1,38356E+11	3,18265E+11	63219906000	62251930000	985543000	
32	500	1,13384E+11	1,22973E+11	1,22937E+11	36116000	1,25854E+12	2,60629E+11	2,60583E+11	2,60583E+11	6,21373E+11	1,24249E+11	1,22666E+11	1583849000	
32	1000	2,10956E+12	2,41417E+11	2,41348E+11	30532000	9,6168E+11	1,35249E+11	1,35134E+11	1,35134E+11	1,23701E+12	2,48728E+11	2,46554E+11	3075768000	
32	2000	2,37464E+12	4,75717E+11	4,75566E+11	151599000	5,78226E+12	1,32922E+12	1,22897E+12	1,22897E+12	2,47216E+12	5,01487E+11	4,84862E+11	6050030000	
64	250	2,27032E+11	45288500000	45272726000	15800000	2,29292E+11	432057E+1000	43187498000	43187498000	2,55321E+11	49,3508E+11	49,3508E+11	7476330000	
64	500	4,43066E+11	88783393000	88749921000	33051000	5,94106E+11	1,19033E+11	1,18995E+11	1,18995E+11	5,39181E+11	1,08637E+11	1,07151E+11	1485844000	
64	1000	8,34147E+11	1,66445E+11	1,66371E+11	73979000	4,25748E+11	48720045000	48619471000	48619471000	1,1037E+12	2,22106E+11	2,19053E+11	3015103000	
64	2000	1,58971E+12	3,15794E+11	3,15654E+11	139791000	2,74137E+12	5,85446E+11	5,85276E+11	5,85276E+11	2,21544E+12	4,438E+11	4,37905E+11	5880617000	
128	250	2,2217E+11	44346122000	44333251000	12213000	96825994000	16383825000	16371452000	16371452000	1,91162E+11	37280380000	36591621000	703904000	
128	500	3,54915E+11	69908229000	69880004000	28745000	2,11026E+11	43468733000	43452500000	43452500000	3,94237E+11	78114171000	76675517000	1460327000	
128	1000	6,45776E+11	1,29189E+11	1,29157E+11	31893000	3,10982E+11	52623149000	52561584000	52561584000	8,16206E+11	1,63E+11	1,60033E+11	2962250000	
128	2000	1,24315E+12	2,49009E+11	2,48865E+11	143069000	1,13813E+12	2,22878E+11	2,22743E+11	2,22743E+11	1,86667E+12	3,33715E+11	3,27612E+11	6110835000	
256	250	2,124285E+11	37328249000	37316948000	11284000					1,84786E+11	96762785000	36111929000	673977000	
256	500	4,88874E+11	93751469000	93725446000	25968000					3,33381E+11	64258486000	62992425000	1276553000	
256	1000	8,62687E+11	1,71103E+11	1,71044E+11	56988000					6,76549E+11	1,36095E+11	1,33604E+11	2491267000	
256	2000	1,39902E+12	2,78874E+11	2,78735E+11	138755000					1,39892E+12	2,78771E+11	2,73716E+11	5096830000	

Dodatek B: Czasy wykonania programu realizującego algorytm genetyczny (środowisko)

Liczba procesów	Rozmiar populacji	Technologie									
		MPI				OpenCL				OpenMP	
		Czas bez inicjalizacji środowiska (ns)	Czas z inicjalizacją środowiska (ns)	Czas systemowy (ms)	Czas bez inicjalizacji środowiska (ns)	Czas z inicjalizacją środowiska (ns)	Czas systemowy (ms)	Czas bez inicjalizacji środowiska (ns)	Czas z inicjalizacją środowiska (ns)	Czas bez inicjalizacji środowiska (ns)	Czas z inicjalizacją środowiska (ns)
16	250	6,01541E+11	6,018E+11	601885	6,90618E+11	6,90618E+11	690670	6,02854E+11	6,02854E+11	605413	605413
16	500	1,19895E+12	1,19921E+12	1199307	1,36607E+12	1,36607E+12	1366114	1,21914E+12	1,21914E+12	1221806	1221806
16	1000	2,3989E+12	2,39915E+12	2399314	1,10531E+12	1,10531E+12	1105363	2,38239E+12	2,38239E+12	2385128	2385128
16	2000	4,72365E+12	4,72391E+12	4724061	6,18806E+12	6,18806E+12	6188119	4,71944E+12	4,71944E+12	4721947	4721947
32	250	3,12075E+11	3,12406E+11	312539	7,02254E+11	7,02254E+11	702325	3,18269E+11	3,18269E+11	320791	320791
32	500	6,13384E+11	6,13708E+11	613839	1,26151E+12	1,26151E+12	1261562	6,21377E+11	6,21377E+11	623777	623777
32	1000	1,20956E+12	1,20988E+12	1210019	9,65062E+11	9,65062E+11	965111	1,23701E+12	1,23701E+12	1239445	1239445
32	2000	2,37465E+12	2,37498E+12	2375130	5,78556E+12	5,78556E+12	5785606	2,47217E+12	2,47217E+12	2474544	2474544
64	250	2,27032E+11	2,27545E+11	227830	2,32661E+11	2,32661E+11	233327	2,55325E+11	2,55325E+11	257984	257984
64	500	4,43066E+11	4,4357E+11	443820	5,97472E+11	5,97472E+11	597516	5,39185E+11	5,39185E+11	541869	541869
64	1000	8,34148E+11	8,34652E+11	834921	4,28798E+11	4,28798E+11	428838	1,10371E+12	1,10371E+12	1106417	1106417
64	2000	1,58971E+12	1,5902E+12	1590470	2,74444E+12	2,74444E+12	2744487	2,21544E+12	2,21544E+12	2217870	2217870
128	250	2,22171E+11	2,23263E+11	223981	99708383000	99708383000	99749	1,91165E+11	1,91165E+11	194015	194015
128	500	3,54915E+11	3,55937E+11	356686	2,13923E+11	2,13923E+11	213974	3,9424E+11	3,9424E+11	396735	396735
128	1000	6,45777E+11	6,46808E+11	647525	3,14285E+11	3,14285E+11	314330	8,16209E+11	8,16209E+11	818682	818682
128	2000	1,24315E+12	1,24419E+12	1244836	1,13915E+12	1,13915E+12	1139199	1,66667E+12	1,66667E+12	1669280	1669280
256	250	2,14285E+11	2,18379E+11	220244				1,8479E+11	1,8479E+11	187790	187790
256	500	4,88874E+11	4,92826E+11	494794				3,33385E+11	3,33385E+11	336096	336096
256	1000	8,62687E+11	8,66878E+11	868407				6,76552E+11	6,76552E+11	679266	679266
256	2000	1,39902E+12	1,40262E+12	1404759				1,39893E+12	1,39893E+12	1401980	1401980

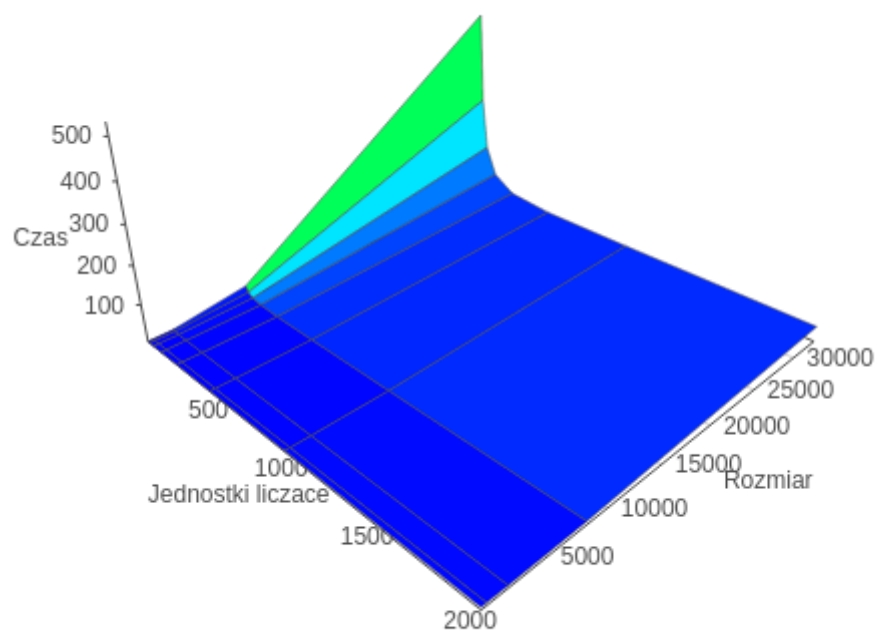
Dodatek C: Czasy wykonania programu realizującego algorytm Mandelbrot

Liczba jednostek liczących	Rozmiar problemu	Technologie									
		MPI					OpenCL				
		Czas bez inicjalizacji środowiska (ns)	Czas z inicjalizacją środowiska (ns)	Czas systemowy (ms)	Czas bez inicjalizacji środowiska (ns)	Czas z inicjalizacją środowiska (ns)	Czas systemowy (ms)	Czas bez inicjalizacji środowiska (ns)	Czas z inicjalizacją środowiska (ns)	Czas systemowy (ms)	Czas bez inicjalizacji środowiska (ns)
16	128	7944000	270516000	351	21584000	1983376000	5076	557061000	557061000	3247	571155000
16	512	86369000	346938000	418	161493000	2149579000	5187	642978000	642978000	3304	692633000
16	2048	1331724000	1588946000	1661	3650586000	5607083000	9018	1948557000	1948557000	4283	2614696000
16	8192	21216009000	21477620000	21551	59070039000	60719087000	63080	22922216000	22922216000	25501	33447425000
16	32768	3,39321E+11	3,39579E+11	339654	9,98679E+11	1,00588E+12	1003691	3,58488E+11	3,58488E+11	361945	5,28274E+11
32	128	6326000	334620000	432	21507000	1847693000	4866	571964000	571964000	2902	568934000
32	512	46369000	370277000	467	158107000	1690216000	4731	634230000	634230000	2962	641617000
32	2048	685395000	1013097000	1110	3848950000	5744897000	8787	1298781000	1298781000	3645	1715697000
32	8192	10865354000	11193895000	11292	58786024000	60485165000	63554	12264784000	12264784000	14662	18853884000
32	32768	1,73655E+11	1,73983E+11	174085	9,97743E+11	9,99632E+11	1002640	1,92111E+11	1,92111E+11	195206	2,919E+11
64	128	37034000	528537000	710	16980000	1843280000	4772	607076000	607076000	3162	620965000
64	512	59602000	543387000	719	80362000	1823722000	4917	634501000	634501000	2992	655447000
64	2048	454307000	945947000	1103	1469673000	3278793000	6317	987949000	987949000	3355	1209469000
64	8192	6791988000	7269078000	7441	24780398000	26629864000	29862	6773710000	6773710000	9287	10303652000
64	32768	1,07929E+11	1,08419E+11	108601	4,08866E+11	4,08442E+11	411523	99210306000	99210306000	102681	1,55344E+11
128	128	110897000	1154013000	1606	16858000	1811286000	4783	661024000	661024000	3163	655233000
128	512	109908000	1158859000	1565	61170000	1604077000	4635	670822000	670822000	3466	674939000
128	2048	308750000	1353575000	1767	508220000	2297762000	5688	8696880000	8696880000	3668	975403000
128	8192	3512253000	4546895000	4995	12717519000	14659447000	17626	4162744000	4162744000	6834	5814510000
128	32768	54411604000	55436956000	55948	2,07209E+11	2,08742E+11	212083	58288091000	58288091000	61449	82718099000
256	128	423784000	4119445000	5700	16705000	1811934000	4738	810260000	810260000	3291	801713000
256	512	393618000	4000220000	5338	45522000	2081964000	5065	799578000	799578000	3452	807686000
256	2048	479267000	4190641000	5695	285536000	2074501000	5185	922960000	922960000	3591	959472000
256	8192	2643958000	6431202000	7933	6744573000	8611347000	12023	3115550000	3115550000	5715	3625665000
256	32768	37181708000	40991964000	42564	1,09237E+11	1,1102E+11	114073	39577132000	39577132000	42838	46489154000
512	128				16692000	2150749000	5678	1147933000	1147933000	4021	1140677000
512	512				37389000	1697599000	4729	1176198000	1176198000	3835	1179022000
512	2048				167517000	1910320000	4974	1252508000	1252508000	3911	1281659000
512	8192				3493368000	5315519000	8397	3170061000	3170061000	5886	3377458000
512	32768				57026160000	58949516000	62086	34004898000	34004898000	37404	35142472000
1024	128				16219000	1907505000	5028	2197787000	2197787000	5602	2215976000
1024	512				36522000	1886645000	5169	2176902000	2176902000	5433	2275011000
1024	2048				107044000	2066096000	5417	2325755000	2325755000	5608	2374468000
1024	8192				1857323000	405279000	7044	4140279000	4140279000	7339	4261402000
1024	32768				29763761000	31305546000	34459	34101920000	34101920000	38143	34362344000
2048	128							5393684000	5393684000	9116	5486563000
2048	512							5607037000	5607037000	9715	5363722000
2048	2048							5522413000	5522413000	9438	5520837000
2048	8192							7627572000	7627572000	11664	7331153000
2048	32768							38203251000	38203251000	42819	38166304000

Dodatek D: Czasy wykonania programu typu klient-serwer

Liczba jednostek liczących	Rozmiar problemu	Technologie																						
		MPI							OpenCL									OpenMP						
		Czas ządania (ns)	Średni czas ządania (ns)	Czas bez inicjalizacji środowiska (ns)	Całkowity czas (ns)	Czas systemowy (ms)	Czas ządania (ns)	Średni czas ządania (ns)	Czas bez inicjalizacji środowiska (ns)	Całkowity czas (ns)	Czas systemowy (ms)	Czas ządania (ns)	Średni czas ządania (ns)	Czas bez inicjalizacji środowiska (ns)	Całkowity czas (ns)			Czas systemowy (ms)						
16	64	94816000	246245319	1908980000	2171170000	2303	999000	1438926	3684913000	7061466000	10179	570000	678502	957499000	957499000	3510								
16	256	3743000	239009782	1912021000	2173022000	2322	1019000	1500784	4556723000	6519544000	9658	1002000	1159257	1348425000	1348425000	3856								
16	1024	4212000	24981725	2509426000	2769978000	3000	2675000	3972981	3215296000	5176486000	8290	3214000	3237281	3037061000	3037061000	5437								
16	4096	13576000	14113369	8670251000	8940623000	9119	3635000	4621685	5259762000	7002432000	10480	10237000	10284353	9023846000	9023846000	11446								
16	16384	51462000	53217868	33058919000	33341823000	33416	12535000	13736371	11316298000	13142050000	16254	31465000	32662828	29214966000	29214966000	31744								
32	64	532409000	535045293	2080196000	2408030000	2559	1070000	3364083	4993302000	6939093000	10229	683000	950351	836910000	836910000	3402								
32	256	435936000	680843574	2498574000	2827175000	2989	1138000	1957765	4160051000	5982382000	9037	1004000	1241904	1015348000	1015348000	3627								
32	1024	12604000	257658761	2445903000	2774333000	2871	2886000	6382006	4412027000	6362374000	9751	3506000	4005156	2269640000	2269640000	4727								
32	4096	14120000	14753332	4964768000	5299022000	5398	3858000	5629738	5011416000	6843154000	9912	8532000	8662162	5297173000	5297173000	7684								
32	16384	52923000	53968260	19861790000	20235451000	20362	12538000	13760963	11487982000	13458165000	16895	28565000	28980322	17885113000	17885113000	20271								
64	64	12066000	194961408	2124435000	2611019000	2794	1154000	1533176	3991923000	5694282000	8756	759000	1466496	758608000	758608000	3521								
64	256	662187000	874583890	2556765000	3054646000	3228	1258000	2617486	4379820000	5837643000	9202	1518000	2367668	1001360000	1001360000	3704								
64	1024	861034000	1055239770	2798891000	3305420000	3509	2082000	4892023	4755454000	6716530000	9768	4004000	4680522	1751742000	1751742000	4192								
64	4096	20242000	20718332	3287045000	3805402000	3965	3799000	8276134	6258424000	7902854000	11303	12292000	12969585	4627198000	4627198000	7347								
64	16384	65649000	65553661	11405224000	11942376000	12235	12488000	13438009	12460316000	14540564000	17683	44612000	45820267	16033750000	16033750000	18445								
128	64	796422000	1063113500	3512193000	4600199000	5062	1157000	1958069	4730162000	6432871000	9515	1293000	3330215	795322000	795322000	3436								
128	256	43603000	582510583	3250301000	4309787000	4747	1482000	2527958	5410274000	7253713000	10484	1809000	3779329	927475000	927475000	3591								
128	1024	1901046000	1475906050	3551158000	4646374000	5084	1949000	5541988	5808090000	7801378000	10845	5249000	7334883	1594710000	1594710000	4125								
128	4096	36059000	37153550	36889113000	4768632000	5200	3726000	5541311	6861676000	8534865000	11558	15092000	17064974	3912501000	3912501000	6375								
128	16384	87119000	98071555	8741847000	9850325000	10293	12578000	13978103	10943911000	12682836000	15862	57609000	61210455	13690255000	13690255000	16256								
256	64	1790769000	1490059156	8959017000	12790404000	14274	1201000	1943898	6361292000	8037083000	11207	1828000	9304875	914302000	914302000	3458								
256	256	1966610000	1840088657	9052875000	13087863000	14439	1207000	2181211	6813313000	8871205000	12460	3276000	10744000	1051627000	1051627000	3609								
256	1024	2850805000	2544672632	9327709000	13197392000	14607	2357000	4659311	6188317000	8156705000	11526	9846000	16916875	1732963000	1732963000	4273								
256	4096	133608000	142653531	9971354000	13990654000	15324	3630000	5304424	7886171000	9715127000	12841	34964000	43147825	4317247000	4317247000	7061								
256	16384	3365552000	3152506571	13290466000	17349346000	18659	12722000	13880137	12611606000	14115852000	17309	95773000	117906300	13597387000	13597387000	16165								

Dodatek E: Wykres przedstawiający działanie programu Mandelbrot w technologii OpenMP przy wartości zmiennej MIC_KMP_AFFINITY równej compact



Dodatek F: Wykres przedstawiający działanie programu Mandelbrot w technologii OpenMP przy wartości zmiennej MIC_KMP_AFFINITY równej scatter

