Państwowa Wyższa Szkoła Informatyki i Przedsiębiorczości w Łomży

Instytut Informatyki i Automatyki

PRACA DYPLOMOWA INŻYNIERSKA

SYMULACJA FUNKCJONALNEGO SYSTEMU KWANTOWEGO NA RÓWNOLEGŁYCH KOMPUTERACH KLASYCZNYCH IV GENERACJI

Autor: Łukasz Świerczewski
Nr Albumu: 2480

Promotor: dr Dariusz Surowik

Łomża 2013

Praca Inżynierska

złożona jako częściowe spełnienie
wymagań do uzyskania tytułu zawodowego

Inżynier Informatyk

specjalizacja:

Systemy Oprogramowania

Łukasz Świerczewski
urodzony 26 maja 1989 roku w Łomży, Polska

Składam serdeczne podziękowania Promotorowi pracy, dr Dariuszowi Surowikowi za cenne uwagi i wskazówki udzielane podczas pisania niniejszej pracy. Pragnę również podziękować wszystkim pracownikom Instytutu za okazaną życzliwośc i pomoc.

Praca została przygotowana z wykorzystaniem zasobów superkomputerowych udostępnionych przez Wydział Matematyki, Fizyki i Informatyki Uniwersytetu Marii Curie-Skłodowskiej w Lublinie.

W badaniach zastosowano sprzęt Instytutu Informatyki i Automatyki Państwowej Wyższej Szkoły Informatyki i Przedsiębiorczości w Łomży.

Praca została wykonana z wykorzystaniem Infrastruktury PL-Grid.

Do obliczeń wykorzystano Intel Manycore Testing Lab oraz zasoby udostępnione przez IBM.

Spis treści

1. Wstęp

Korzeni informatyki można doszukiwać się już u samych początków cywilizacji, gdy człowiek próbował odwzorowywać wartości liczbowe do wykonania pierwszego szkicu prostego kalkulatora mechanicznego przez Leonarda da Vinci w 1500 roku. Początek współcześnie znanej informatyki datuje się na lata 40. XX wieku, gdy II Wojna Światowa wymusiła rozwój całkowicie nowych maszyn wspierających proces łamania szyfrów. Nowe technologie zostały szybko zaadaptowane przez wojsko, gdzie wykorzystywano je m. in. do obliczania tablic balistycznych i projektowania taktycznej broni atomowej. Dzisiejszy kierunek rozwoju fizycznego modelu maszyny obliczeniowej zawdzięczamy głównie opracowaniu w latach 1947-1948 roku przez Williama Bradforda Shockleya, Johna Bardeena oraz Waltera Housera Brattaina tranzystora ostrzowego. Twórcy wynalazku w 1956 roku otrzymali nagrodę Nobla w dziedzinie fizyki.

Obecnie jesteśmy już przyzwyczajeni do rozumienia komputera jako szeregu komponentów elektronicznych połączonych ze sobą w określony sposób. Wizja ta nie jest jedyną możliwą, a najprawdopodobniej nie jest także tą optymalną. Wizja kresu technologii klasycznych zmusza naukowców i inżynierów do poszukiwania rozwiązań alternatywnych. Jednym z nich jest komputer kwantowy, który w trakcie obliczeń wykorzystuje obiekty kwantowe - najczęściej fotony lub elektrony. Mechanika kwantowa nawet z niewielką ilością współcześnie znanych algorytmów umożliwia teoretycznie realizację zadań, których nie można wykonać w akceptowalnym dla człowieka czasie na żadnych klasycznych komputerach bazujących na układach scalonych. Do tego typu procesów należą m. in. możliwość przeszukiwania zbioru o rozmiarze N w poszukiwaniu określonego elementu w czasie $\sqrt{N}$ lub wymiana kluczy kryptograficznych pozornie z prędkością wyższą od prędkości światła. Cała kryptografia XX i początku XXI wieku wykorzystuje znane problemy z dziedziny matematyki. W celu złamania danego szyfru wystarczy *'jedynie'* zastosowanie algorytmu będącego implementacją rozwiązania zagadnienia matematycznego, które stanowiło bazę dla zakodowania informacji. Mechanika kwantowa dzięki całkowicie odmiennemu zdefiniowaniu zjawiska pomiaru stanu jednostki informacji ujawnia nowe oblicze możliwości przekazywania informacji w sposób zabezpieczony. Aby w nieautoryzowany sposób odczytać dane szyfrowane za pomocą komputera kwantowego musiałaby istnieć możliwość złamania praw fizyki kwantowej.

Niniejsza praca prezentuje podstawowe aspekty wykorzystania obiektów kwantowych do obliczeń oraz symulację ich na współczesnych komputerach. Do dnia dzisiejszego nie udało się zbudować w pełni funkcjonalnego komputera kwantowego. Mimo to posiadamy szereg algorytmów kwantowych, których działanie możemy testować na dostępnym sprzęcie.

1.1. Teza pracy

Tezę pracy definiujemy następująco:

> *Symulacje ewolucji w czasie układu kwantowego można efektywnie wykonywać na klasycznych komputerach równoległych czwartej generacji i opartych na nich systemach rozproszonych.*

gdzie przez *efektywność* należy rozumieć pełne wykorzystanie procesorów wpływające na uzyskane przyśpieszenie, a nie *efektywność* względem maszyny w pełni działającej zgodnie z zasadami mechaniki kwantowej, gdyż takiej sprawności nie może uzyskać żadne współczesne rozwiązanie elektroniczne [84].

1.2. Cel główny

Praca obejmuje implementację systemu umożliwiającego efektywną symulację działania komputera kwantowego z wykorzystaniem technologii programowania równoległego OpenMP, MPI, CUDA oraz OpenCL.

W ramach pracy uruchomiono platformę działającą w oparciu o Berkeley Open Infrastructure for Network Computing, dzięki której możliwe było wykonanie obliczeń w środowisku rozproszonym za pomocą setek komputerów połączonych jedynie siecią Internet.

Podczas testów wykorzystano karty graficzne AMD Radeon, nVidia GeForce, Tesla oraz procesory Cell *(zastosowane w IBM Blade QS22 i konsoli PlayStation 3)*, jednostki przetwarzania mainframe IBM System/390, a także klasyczne procesory Intel. Proste symulacje przeprowadzono na urządzeniach z układami o architekturze ARM.

2. Informatyka klasyczna

Informatyka swój rozwój zawdzięcza badaniom z wielu dziedzin nauki *(m.in. elektronika: modele obwodów, sygnałów i algorytmika)*. Większość rozwiązań opiera się na przetwarzaniu sygnałów cyfrowych. Jednak jak wiadomo pomysłów na rozwój komputerów było bardzo wiele [272], [177], [118] i część z nich w znacząco odbiega od pierwotnego modelu skrótowo poruszonego w tym rozdziale.

2.1. Abstrakcyjny model komputera klasycznego

Dział informatyki o nazwie *'Algorytmika'* w dużej mierze skupia się na złożoności obliczeniowej różnych opisów rozwiązania danego problemu. Zadanie te tylko wbrew pozorom wydaje się proste, ponieważ już na samym początku musimy odpowiedzieć na dość trudne pytanie: Dla jakiego komputera będziemy ten problem rozpatrywać? We współczesnym świecie istnieją miliardy komputerów, które różnią się od siebie. Możemy nawet przyjąć, że interesują nas tylko maszyny o architekturze SPARC, jednak i w tej puli odnajdziemy wiele procesorów w zasadniczy sposób różniących się m. in. dostępnymi listami rozkazów. W przypadku całej architektury SPARC musimy uwzględnić także organizację pamięci, strukturę jednostek przetwarzania i dostępne funkcje I/O. Rzeczy tych nie można łatwo zdefiniować i dla różnych rodzin komputerów byśmy musieli posiadać dokładne opisy ich działania, które tworzą ich model obliczeń.

W 1936 roku Alan Turing zaproponował swój model Maszyny Turinga, który jest bardzo prosty w zrozumieniu i można na jego działaniu oprzeć większość komputerów. Maszyna Turinga jest zbudowana z następujących elementów:

- słownika,
- nieskończonej taśmy, na której znajdują się komórki z symbolami zrozumiałymi dla maszyny,
- głowicy umożliwiającej przesuwanie taśmy oraz zapis i odczyt symboli,
- mechanizmu umożliwiającego sterowanie działaniem maszyny na podstawie aktualnego stanu oraz symbolu. Może on wykonać następujące operacje:
  - modyfikacja wartości komórki do której ma aktualnie dostęp głowica,
  - przesunięcie głowicy w lewo lub prawo,
  - zmiana stanu elementu sterującego.

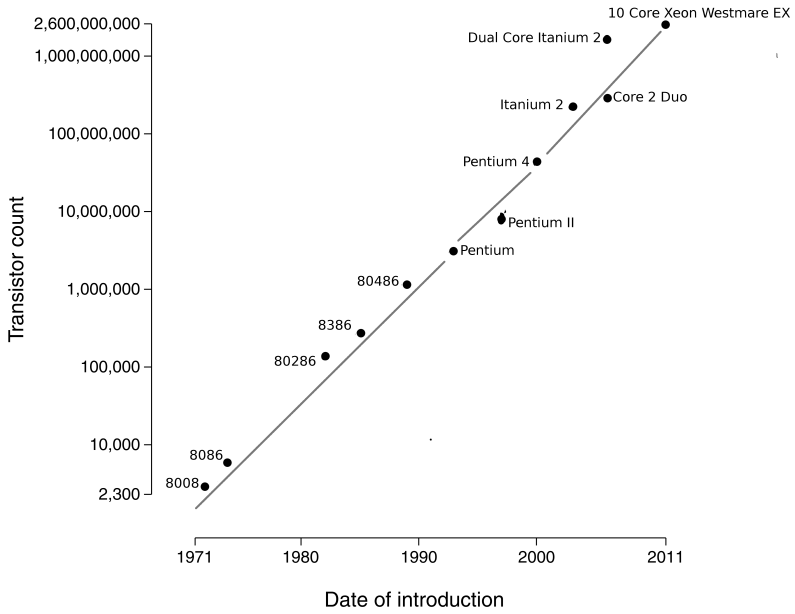
Koncepcja ta jest zaskakująco prosta i z tego powodu jest powszechnie stosowana. Należy jednak zauważyć, że Maszyna Turinga jest konstrukcją tylko teoretyczną, która nie ma swojego dokładnego odpowiednika sprzętowego z jednego powodu - operuje ona na nieskończonej taśmie. Dzięki temu mechanizm sterujący może zawsze wykonać przesunięcie nośnika danych w lewo lub prawo i nigdy nie dojdzie do niebezpiecznego wyjścia poza dostępny obszar.

U podstaw teorii obliczalności leży także pojęcie zupełności Turinga. Definiuje ona dość istotną cechę języków programowania nazywaną tzw. *'zupełnością w sensie Turinga'*, która oznacza, że język programowania lub maszyna obliczeniowa jest w stanie zrealizować dowolny algorytm, który potrafi wyrazić człowiek za pomocą opisu matematycznego. Pomimo, że pomijana jest kwestia złożoności zastosowanego kodu i efektywności realizacji obliczeń to sama gwarancja możliwości przeniesienia każdego poprawnego algorytmu zapisanego na kartce papieru do pamięci komputera, w której zostanie on zrealizowany daje olbrzymie możliwości. Cokolwiek więc by nie wymyślił matematyk to programista dysponujący odpowiednią wiedzą zawsze może to zrealizować na maszynie liczącej *(oczywiście w ramach zdrowego rozsądku)*.

W rzeczywistości problemy pojawiają się nie w samej implementacji kodu, a na etapie opracowywania algorytmu. Nie wszystko w prosty sposób można opisać za pomocą operacji matematycznych. Istnieją problemy, które do dzisiaj nie mają jasnego rozstrzygnięcia i budzą wiele emocji oraz kontrowersji *(np. problem $P = NP$ [96] [85])*.

2.2. Ograniczenia technologii konwencjonalnych - kres elektroniki

W 1965 roku Gordon Moore, jeden z założycieli firmy Intel zauważył podwajanie się liczby tranzystorów w mikroprocesorach co około 18 miesięcy [193]. Upakowywanie coraz większych ilości elementów w mikroprocesorach jest możliwe dzięki doskonaleniu procesów odwzorowywania zaprojektowanych masek układu scalonego na powierzchni płytki krzemowej - tranzystory mają coraz mniejsze rozmiary. Aktualna technologia ogranicza się do *45* lub *32* nm. Dla porównania w latach *90.* chipy były wykonywane w technice *500* nm. Trend taki jednak nie może trwać w nieskończoność. W pewnym momencie nie zdołamy już budować szybszych układów elektronicznych ze względu na granice fizyczne procesów litograficznych *(tranzystory nie mogą być mniejsze od atomów)*. Wykres przedstawiający wybrane procesory Intela w odniesieniu do krzywej Moore'a przedstawiono na Rys. 2.1.



Rys. 2.1. Krzywa Moore'a - wzrost ilości tranzystorów w procesorach *(dla układów marki Intel). Źródło: [165] w oparciu o Wikipedia.org.*

Oczywiście szybkość mikroprocesora nie wynika jedynie z ilości tranzystorów, ale także z jego architektury. Dzięki jej poprawie można zwiększyć efektywność jednostek wykonawczych. Istnieje reguła Pollack'a [24] mówiąca, że wzrost wydajności spowodowany poprawą architektury to pierwiastek kwadratowy z liczby tranzystorów lub obszaru rdzenia. Realny wzrost szybkości przetwarzania wynikający z dwukrotnego zwiększenia ilości tranzystorów może wynosić ok. *40%*, jednak kolejne *40%* to zysk uzyskany dzięki poprawie architektury. W praktyce jednak proces opracowywania nowej architektury jest bardzo drogi i nie z każdą generacją procesorów firmy wprowadzają znaczące zmiany [234].

Producenci dochodząc do granicy częstotliwości taktowania procesorów postanowili wyposażać układy w wiele rdzeni. Dla zwykłego użytkownika większa ilość tranzystorów w tego typu procesorze może niewiele zmienić. Istnieją przypadki, gdy wzrost ich ilości o *40%* daje przyrost wydajności wynoszący tylko od *10%* do *20%* [236].

Istnieją także inne czynniki, które znacznie ograniczają elektronikę takie jak skończona prędkość propagacji impulsu elektrycznego w półprzewodnikach lub prawo Amdahla [91], [206] poruszone dokładniej podczas omawiania problemów przetwarzania równoległego.

Jednak nawet w przypadku, gdy uwzględnimy plany Intela dotyczące ciągłego, dynamicznego rozwoju rynku procesorów do aż 2029 roku [89], potwierdzenie przez zespół naukowców z Uniwersytetu w Pittsburgu możliwości budowy tranzystorów o rozmiarze *1,5* nm z materiałów na bazie tlenków [254] oraz opracowanie przez specjalistów z IBM i Georgia Tech tranzystora pracującego z prędkością *500* GHz *(symu-*

lacje wykazały perspektywy pracy przy częstotliwości 1 THz) [205] to kres możliwości miniaturyzacji i tak jest tylko kwestią czasu. Zasadne więc wydaje się poszukiwanie alternatyw dla dzisiejszej elektroniki.

2.3. Idea niekonwencjonalnych maszyn obliczeniowych

Głównie wynalezieniu w 1950 roku tranzystora złączowego przez Johna Bardeena, Waltera Housera Brattaina oraz Williama Bradforda Shockleya *(za swe odkrycie otrzymali w 1956 roku nagrodę Nobla z fizyki)* zawdzięczamy fizyczną koncepcję dzisiejszych komputerów, które znajdują się zarówno w telefonach jak i zawsze stoją na naszych biurkach. Maszyna obliczeniowa jednak nie musi opierać swojego działania na tranzystorach ani też w większym stopniu na elektronice. Prowadzone są badania nad rozwiązaniami niekonwencjonalnymi takimi jak np. komputery optyczne *(lub inaczej fotoniczne)*, które wykorzystują zamiast prądu elektrycznego przepływ fotonów [177], [118]. Podstawowym problemem podczas budowy komputera tego rodzaju jest optyczny odpowiednik tranzystora, dzięki któremu w pewnym stopniu możliwe będzie kontrolowanie właściwości optycznych danego ośrodka i działanie jak bramka logiczna. Istnieją metamateriały, z których można byłoby takie elementy zbudować [204]. Sukcesy przynoszą także badania nad komputerami biologicznymi. W 2011 roku zespołowi z Imperial College London udało się utworzyć bramki logiczne z bakterii jelitowych i kwasu dezoksyrybonukleinowego [272].

Całkowicie innym podejściem są komputery kwantowe, których tematyka zostanie dokładniej opisana w kolejnych działach.

3. Mechanika kwantowa

Mechanika kwantowa jest dziedziną fizyki, która wprowadziła do świata nauki olbrzymie ilości kontrowersji. Wstęp teoretyczny w tym rozdziale jest ledwie symboliczny. Istnieje wiele koncepcji mechaniki kwantowej ja jednak będe opierać się na kwantowej teorii nazywanej *'Wieloświatową Interpretacją Mechaniki Kwantowej' (ang. The Many-Worlds Interpretation of Quantum Mechanics)*. Została ona zaproponowana przez Hugha Everetta III w jego rozprawie doktorskiej z 1957 roku [80].

Problemy z interpretacją Natury rozpoczęły się jednak już trochę wcześniej. Około 1805 roku angielski fizyk Thomas Young odnalazł bardzo poważny argument przeciwko korpuskularnej koncepcji światła, którą promował Isaac Newton. Wykonał on prosty eksperyment polegający na przepuszczeniu światła spójnego przez siatkę dyfrakcyjną *(szklana płytka z dużą ilością rys, które pełnią rolę przysłon)*. W wyniku doświadczenia Young otrzymał na ekranie nie zbiór plamek lecz prążki. Jeżeli w uproszczeniu przeanalizujemy przeszkodę dla spójnego światła złożoną z dwóch szczelin to wykorzystując teorię, której wielkim zwolennikiem był Newton, wiązka światła powinna zachowywać się jak uporządkowany zbiór cząsteczek, a co za tym idzie powinniśmy zobaczyć dwie jasne kropki, będące efektem naświetlenia. Światło jednak zachowywało się jak fala, która dochodząc do przeszkody z dwiema szczelinami, zgodnie z zasadą Huygensa automatycznie utworzyła dwie nowe fale cząstkowe interferujące ze sobą. Jasny prążek oznacza więc zgodność faz *(maksimum - wzmocnienie fal)*, a ciemny przeciwfazę *(minimum - wygaszenie fal)*. Jeżeli jednak światło zachowuje się jak fala to potrzebuje ona ośrodka, w którym może się rozchodzić. Światło jednak rozchodzi się nawet w próżni. Zagadkę próbowano wyjaśnić wprowadzając pojęcie *eteru* jako hipotetycznego ośrodka, w którym mogłyby się rozchodzić fale elektromagnetyczne oraz światło. Koncepcję tą rozwinął James Clerk Maxwell w artykule *'Eter'* opublikowanym na łamach Encyklopedii Britannica [175]. Mimo wszystko w 1887 roku doświadczenie Michelsona-Morleya zaprzeczyło możliwości istnienia takiej substancji [185].

Dobrze jednak wiemy, że światło istnieje także w postaci cząstek, które są nazywane fotonami - mówimy o naturze *'korpuskularno-falowej'* światła. Jak więc wytłumaczyć ten eksperyment, gdy będziemy rozpatrywać światło jako cząstki?[1] Standardowa interpretacja mówi, że obiekty kwantowe *(np. foton lub elektron)* w zależności od okoliczności mogą zachowywać się jak fale lub jak cząsteczki. Istnieje

[1] W mechanice kwantowej można obliczyć prawdopodobieństwo znalezienia cząstki w różnych miejscach przestrzeni. Przeprowadzono jednak eksperymenty interferencyjne na jednym i dwóch fotonach *(m.in. L. Mandel z University of Rochester [170])*, które ukazują, że interferencja nadal kwestionuje nasze tłumaczenia i najprawdopodobniej jeszcze wszystko nie zostało wyjaśnione.

jednak inna koncepcja, którą opisał Hugh Everett. Według niej np. w momencie, gdy cząstka może poruszać się n drogami jednocześnie powstaje n wszechświatów, a w każdym z nich obiekt kwantowy porusza się inną drogą.

Przed zaproponowaniem *Many-worlds interpretation* rzeczywistość była postrzegana jedynie jako jeden możliwy ciąg wydarzeń. W MWI można ją porównać do drzewa rozpinającego, gdzie w jego rozgałęzieniach każdy możliwy wynik doświadczenia kwantowego ma swoje odzwierciedlenie w innym uniwersum.

Eksperyment z dwiema szczelinami i fotonami można więc na bazie MWI sprowadzić do rozważania, że dany foton w jednym uniwersum przeleci przez jedną szczelinę, a w innym uniwersum przez drugą *(zjawisko równoległości kwantowej)*. Nie wykonując pomiaru możemy jedynie operować na prawdopodobieństwach. Gdy jednak dokonamy pomiaru stan zostanie zredukowany. W doświadczeniu gdy patrzymy na ekran widzimy rezultat, który jest wynikiem interferencji z wielu wszechświatów. Jeżeli jednak foton może poruszać się wieloma drogami jednocześnie to może istnieć sposób na wykorzystanie tego do obliczeń. Urządzenie wykorzystujące często nie do końca wyjaśnione własności mechaniki kwantowej do przetwarzania danych to komputer kwantowy. Marzenie wielu naukowców.

Konsekwencje możliwości istnienia światów równoległych są cały czas intensywnie badane przez teoretyków z całego świata [75], [255], [62]. Prowadzone są nawet rozważania nad możliwościami kwantowych algorytmów genetycznych *(QGA - ang. Quantum Genetic Algorithm)*, które w odróżnieniu do klasycznych algorytmów genetycznych *(CGA - ang. Classical Genetic Algorithm)* nie ograniczają się jedynie do jednej rzeczywistości [11].

3.1. Matematyczny model opisu mikroświata oraz fizyczna reprezentacja obiektów kwantowych

Informatyka kwantowa jako dziedzina łącząca informatykę i mechanikę kwantową definiuje własne pojęcia, które w dużej mierze są uzależnione od interpretacji obiektów fizycznych. Zarówno rejestry kwantowe jak i same kubity w zasadniczy sposób różnią się od tych stosowanych we współczesnych komputerach. Zrozumienie podstawowych struktur jest wymagane do prowadzenia symulacji i analizy algorytmów.

3.2. Kubit

Najmniejszą i niepodzielną porcją informacji w informatyce kwantowej jest kubit. Jest on dwupoziomowym układem kwantowym. Różnica między kubitem, a klasycznym bitem polega na tym, że kubit może znajdować się w dowolnej super-

pozycji dwóch stanów kwantowych. Z matematycznego punktu widzenia kubit jest wektorem dwuwymiarowej przestrzeni Hilberta[2] $\mathbb{H}_2$. Możemy obrać w przestrzeni $\mathbb{H}_2$ bazę, w której wyznaczymy wektory $|0\rangle$ oraz $|1\rangle$. W literaturze bazę tą nazywa się zazwyczaj *bazą standardową*. Baza ta ze względu na swoją prostotę jest często stosowana.

Rozważmy kubit:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{3.1}$$

gdzie liczby zespolone α i β spełniają następujący warunek:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{3.2}$$

Współczynniki tej kombinacji liniowej są nazywane amplitudami stanu. Możemy również wykorzystać notację Diraca Bra-ket [63] według której:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \Rightarrow |\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{3.3}$$

Jeżeli wykorzystamy fakt, że po dokonaniu pomiaru na kubicie $|\psi\rangle$ zostanie on zredukowany do stanu $|0\rangle$ z prawdopodobieństwem $|\alpha|^2$, natomiast z prawdopodobieństwem $|\beta|^2$ przyjmie wartość $|1\rangle$ możemy także zapisać:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.4}$$

Jedną z najistotniejszych operacji - operację pomiaru kwantowego można więc zdefiniować w następujący sposób:

Definicja:

> Pomiarem kubitu jest obserwacja, która redukuje układ do jednego ze stanów bazowych $|0\rangle$ lub $|1\rangle$. Po dokonaniu pomiaru[3] kubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ znajdzie się w stanie $|0\rangle$ z prawdopodobieństwem $|\alpha|^2$ lub w stanie $|1\rangle$, któremu odpowiada prawdopodobieństwo $|\beta|^2$.

Możemy także w przestrzeni $\mathbb{H}_2$ zdefiniować operator liniowy przekształcający wektor tej przestrzeni na liczbę zespoloną. Operator ten będzie miał postać iloczynu skalarnego z innym wektorem $|\phi\rangle$ i zostanie oznaczony jako $\langle\phi|$. Dla każdego $|\psi\rangle \in \mathbb{H}_2$ możemy zdefiniować operację:

$$\langle\phi|\psi\rangle. \tag{3.5}$$

[2]Przestrzeń Hilberta – w analizie funkcjonalnej rzeczywista lub zespolona przestrzeń unitarna *(tj. przestrzeń liniowa nad ciałem liczb rzeczywistych lub zespolonych z abstrakcyjnym iloczynem skalarnym)*, zupełna ze względu na indukowaną *(poprzez normę)* z iloczynu skalarnego tej przestrzeni metrykę.

[3]Pomiarów można dokonywać także w innych bazach - np. bazie Hadamarda.

Można zauważyć, że:

$$\langle 0|\psi\rangle = \alpha, \tag{3.6}$$
$$\langle 1|\psi\rangle = \beta,$$

gdy wektor $|\psi\rangle$ jest zdefiniowany tak jak przedstawiono to we wzorze 3.1.

Najczęściej jako fizyczny model kubitu przyjmuje się cząstki o spinie[4] $\frac{1}{2}$ - głównie elektrony lub także fotony *(w ich przypadku jednak interesująca jest polaryzacja).*

3.3. Rejestry kwantowe

Aby wykonywać obliczenia kwantowe potrzebujemy jednak więcej niż jednego kubitu. Układem złożonym z wielu kubitów jest rejestr kwantowy. Analogicznie do bitu kwantowego rejestrem kwantowym nazwiemy wektor w przestrzeni Hilberta $\mathbb{H}_{2^n}$ o rozmiarze n i jednostkowej długości.

Rejestr złożony z dwóch kubitów jest iloczynem tensorowym wektorów bazowych układów, z których się składa. Może mieć więc następującą bazę:

$$|00\rangle = |0\rangle \otimes |0\rangle, |01\rangle = |0\rangle \otimes |1\rangle, |10\rangle = |1\rangle \otimes |0\rangle, |11\rangle = |1\rangle \otimes |1\rangle \tag{3.7}$$

Zapis w notacji Diraca zaprezentowany dla pojedynczego kubitu w 3.4 można także zaprezentować dla rejestru kwantowego. Zawsze będzie on przedstawiony jako kolumna o rozmiarze 2^n elementów. Kolumny te otrzymamy licząc iloczyn tensorowy macierzy powiązanych z kubitami składowymi.

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \tag{3.8}$$

Dla pozostałych stanów uzyskamy:

$$|01\rangle = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, |10\rangle = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, |11\rangle = \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \tag{3.9}$$

Każdy rejestr możemy także przedstawić w postaci kombinacji liniowej wektorów bazy.

$$|\psi\rangle = \alpha_0|00\rangle + \alpha_1|01\rangle + \alpha_2|10\rangle + \alpha_3|11\rangle \tag{3.10}$$

[4]Spin – moment własny pędu cząstki w układzie, w którym nie wykonuje ruchu postępowego. Własny oznacza tu taki, który nie wynika z ruchu danej cząstki względem innych cząstek, lecz tylko z samej natury tej cząstki. Każdy rodzaj cząstek elementarnych ma odpowiedni dla siebie spin.

Stan rejestru kwantowego o długości n można wyrazić jako następującą kombinację liniową:

$$\sum_{i=1}^{2^{n-1}} \alpha_i |i\rangle \tag{3.11}$$

gdzie wszystkie współczynniki α_i muszą spełniać warunek:

$$\sum_{i=1}^{2^{n-1}} |\alpha_i|^2 = 1 \tag{3.12}$$

W przypadku pomiaru wykonywanego na rejestrze kwantowym dochodzi do przejścia całego układu w jeden z możliwych stanów bazowych. Konkretny stan wybierany jest z prawdopodobieństwem $|\alpha_i|^2$.

Podsumowując: System złożony z trzech kubitów umożliwia nam zapisanie nie jednej z ośmiu wartości, a wszystkich ośmiu jednocześnie, jednak z pewnym prawdopodobieństwem ich wystąpienia. Dla przykładu:

$$0 : |000\rangle; 1 : |001\rangle; 2 : |010\rangle; 3 : |011\rangle; 4 : |100\rangle; 5 : |101\rangle; 6 : |110\rangle; 7 : |111\rangle; \tag{3.13}$$

Przeanalizujmy w tym momencie ciekawszy przypadek dla pewnego rejestru $|\psi\rangle$.

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{3.14}$$

Gdy wykonamy pomiar na pierwszym kubicie możemy otrzymać z takim samym prawdopodobieństwem stan $|0\rangle$ lub $|1\rangle$. Operacja ta jednocześnie ustala stan drugiego kubitu. W momencie, gdy w wyniku pierwszego pomiaru otrzymaliśmy $|1\rangle$ to drugi kubit musiał mieć także wartość $|1\rangle$, co da rejestr $|11\rangle$. Do podobnego zjawiska dojdzie, gdy podczas pierwszej obserwacji otrzymamy $|0\rangle$ - wtedy cały układ przyjmie stan $|00\rangle$. Tą własność nazywamy *stanami splątanymi*. Fakt ten jeszcze do niedawna doprowadzał fizyków do zakłopotania. Problem jako pierwszy zaobserwowało trzech z nich: Albert Einstein, Borys Podolski i Nathan Rosen, którzy w 1935 roku we wspólnej publikacji *'Can Quantum Mechanical Description of Physical Reality Be Considered Complete?'* przedstawili mechanikę kwantową jako naukę niekompletną i prowadzącą do absurdów [2]. Uczeni w swojej pracy zaproponowali przeprowadzenie eksperymentu myślowego, według którego można wprowadzić dwie cząstki w stan splątany, a następnie oddalić je od siebie na olbrzymie odległości setek lat świetlnych, lub nawet na dwa różne krańce wszechświata *(zakładając, że jest on skończony)*. W momencie, gdy dokonamy pomiaru na jednym z nich dochodzi do automatycznego ustalenia stanu kubitu splątanego z nim lecz znajdującego się w olbrzymiej odległości. Może wydawać się, że te prawa mechaniki kwantowej są w całkowitej sprzecznością z teorią względności według, której nic nie może poruszać się z prędkością większą od prędkości światła. W 1964 roku John Stewart Bell wykazał w pracy [20] błędność podejścia twórców EPR i sformułował znane

Twierdzenie Bella, według którego jakakolwiek teoria zmiennych ukrytych zgodna z teorią względności, nie jest w stanie opisać wszystkich zjawisk mechaniki kwantowej. Badania eksperymentalne całkowicie potwierdzają przewidywania mechaniki kwantowej. Występowanie stanów splątanych udało się zaobserwować m. in. na odległości 144 km między wyspami La Palma i Teneryfa przez Anton'a Zeilinger'a i jego współpracowników. Od tego momentu znacznie więcej zespołów naukowych potwierdziło fenomeny fizyki kwantowej [283], [263], [284]. Wiadomo, że aby możliwe było zajście *'piorunującego zjawiska'* jak nazywał Albert Einstein stan splątany nie trzeba wcale przekraczać prędkości światła. Współczesna nauka zauważyła, że świat znacznie prościej można opisać, gdy nie ograniczamy się do trzech wymiarów, które potrafimy zaobserwować swoimi bardzo niedoskonałymi ludzkimi zmysłami.

Najprawdopodobniej pierwszy w historii wykład, który wyszedł poza zwykłe ramy geometrii w matematyce wygłosił w 1854 roku Riemman. Rozpoczął On od ukazania twierdzenia Pitagorasa

$$a^2 + b^2 = c^2 \tag{3.15}$$

Prawo to można bezproblemowo uogólnić na trzy wymiary. Wtedy suma kwadratów długości trzech krawędzi sześcianu, które wychodzą z tego samego wierzchołka będzie równa kwadratowi długości przekątnej. Zasada ta będzie działała także dla N wymiarów. Możemy więc, opisać relację długości krawędzi hipersześcianu do długości jego przekątnej jako:

$$a^2 + b^2 + c^2 + d^2 + ... = z^2 \tag{3.16}$$

gdzie litery a, b... wyrażają długości krawędzi wychodzących od jednego wierzchołka, a z długość przyprostokątnej. Żaden człowiek nie potrafi sobie wyobrazić ani tym bardziej zobaczyć hipersześcianu, potrafimy jednak badać je wykorzystując matematykę. Podobnie trendy przedostały się do fizyki *(zastosowanie dodatkowych wymiarów w opisie praw przyrody proponował sam Riemann [211])*, w której nowe działy tworzy teoria superstrun [137] i jej rozszerzenie zwane M-Teorią [69]. Zależnie od wersji postulują one istnienie wszechświata mającego od 4 [77] do 13 [17] wymiarów.

Zjawisko stanów splątanych i szybkość przenoszenia informacji można próbować wyjaśnić tym, że odległość pomiędzy dwoma punktami w trzech wymiarach wcale nie musi być równa odległości pomiędzy tymi punktami jeżeli będziemy rozpatrywać przestrzeń złożoną z większej ilości wymiarów. Dla przykładu można przeanalizować płaszczyznę 2D - weźmy dla uproszczenia zwykłą kartkę papieru. Odległość pomiędzy najdalej oddalonymi od siebie punktami na kartce formatu A4 wynosi niewiele ponad 36 cm. W dwóch wymiarach, w których możemy rozpatrywać tę kartkę jest to najniższa z możliwych długości dróg pomiędzy tymi punktami. Jeżeli jednak zanurzymy kartkę w dodatkowym wymiarze i będziemy mogli ją zgiąć to najmniejsza odległość jaką uda nam się uzyskać wyniesie 0.

Zjawisko stanu splątanego wcale nie musi więc być irracjonalnym, które wymyka się z ram jakiejkolwiek w miarę racjonalnej teorii.

Jedno z ciekawszych pytań jakie można jednak zadać brzmi: Dlaczego nie widzimy tych dodatkowych wymiarów? Odpowiedzi na te pytanie poszukuja w swoich książkach *'Krótka historia czasu'* [106] Stephen Hawking i profesor fizyki teoretycznej Uniwersytetu Princeton Michio Kaku w *'Hiperprzestrzeń'* [138]. Jak się okazuje wymiary te moga być skręcone i tak małe *(mniejsze niż długość Plancka* $l_p \approx 1,6161 \cdot 10^{-37}$ *metra)*, że żadne skonstruowane urządzenie pomiarowe nie zdoła potwierdzić eksperymentalnie ich istnienia. Być może sama natura podczas procesu ewolucji *(znanego informatykom doskonale pod przykrywką algorytmów genetycznych)* ograniczyła nas do możliwości postrzegania rzeczywistości tylko w tak drastycznie ograniczonym stopniu. Fizycy twierdzą, że może i wielu rzeczy nie są pewni lub nie wiedzą jednak kwestią tylko odpowiedniego czasu jest rozwiązanie zagadek. Postęp nauki jest imponujący - można przyjąć, że stan wiedzy jaką dysponuje ludzkość podwaja się co 10 lat *(Micho Kaku)*. Wiedza ta jednak nie zawsze wskazuje, że dany problem można rozwiązać, a czasem dowodzi, że problem jest i nie będzie nigdy rozwiązany. Doskonale przekonali się o tym matematycy w 1931 roku, gdy austriacki logik Kurt Gödel udowodnił, że w niesprzecznej teorii matematycznej zawierającej pojęcie liczb naturalnych da się sformułować takie zdanie, którego w ramach tej teorii nie da się ani udowodnić, ani obalić [94]. Oznacza to tylko tyle, że można postawić hipotezę, z którą kompletnie niczego nie uda się zrobić. Nadal jednak nie wiadomo, które wielkie problemy matematyczne są rozwiązywalne, a które nie. Niemniej wiemy, że są takie których nigdy nie uda się definitywnie rozwiązać. Zagadnienie jednak może mieć odbicie w fizyce co na większą skale miałoby dość brzemienne skutki, ponieważ ograniczyłoby człowieka do postaci zamkniętej w klatce przyrody, z której nigdy nie zdoła się wydostać.

Każdy sam musi zdecydować jak będzie rozpatrywał problemy i której teorii będzie zwolennikiem. Wybory te w jakikolwiek sposób nie wpływają na treści zawarte na kolejnych stronach tej pracy.

3.4. Bramki kwantowe

Definicja:

> Bramką w sensie informatyki kwantowej wykonującą przekształcenie rejestru o wielkości n kubitów jest unitarny operator liniowy w przestrzeni $\mathbb{H}_{2^n}$. Bramki kwantowe są macierzami hermitowskimi o rozmiarze 2^n x 2^n.

Z powyższej definicji wynika, że bramka musi być unitarna, więc jej postać powinna być następująca

$$e^{it}\begin{bmatrix} \alpha & \beta^* \\ \beta & \alpha^* \end{bmatrix} \tag{3.17}$$

gdzie

$$|\alpha|^2 + |\beta|^2 = 1 \; oraz \; t \in R \tag{3.18}$$

Jedną z podstawowych bramek kwantowych jest bramka Hadamarda [166], oznaczana za pomocą symbolu H. Bramka ta jest reprezentowana przez macierz:

$$H_1 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.19}$$

Jak widać jest to zwykły iloczyn macierzy Hadamarda przez stałą $\frac{1}{\sqrt{2}}$.

Bramkę tą możemy rozwinąć, aby znalazła swoje zastosowanie podczas wykonywania przekształceń na rejestrach złożonych z n kubitów.

$$H_2 = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{3.20}$$

$$H_3 = \frac{1}{2^{\frac{3}{2}}}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \tag{3.21}$$

Dla dowolnego rozmiaru m można ją zdefiniować w sposób rekurencyjny

$$H_m = \frac{1}{\sqrt{2}}\begin{bmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{bmatrix} \tag{3.22}$$

Konstrukcja ta ma bardzo istotną właściwość. W momencie gdy kubit lub rejestr kwantowy jest wyzerowany i wykonamy na nim przekształcenie za pomocą bramki kwantowej Hadamarda uzyskamy równomierną superpozycję[5].

Tak więc dla kubitu działanie na wektorach stanów bazowych $|0\rangle$ i $|1\rangle$ można przedstawić następująco:

[5]Wszystkie amplitudy stanu mają jednakowe wartości.

$$H_1|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{3.23}$$

$$H_1|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{3.24}$$

Wynik zaprezentowany za pomocą wzoru 3.23 jest równomierną superpozycją, a rezultat przedstawiony w 3.24 stanem splątanym, którego zagadka została poruszona w poprzednim podrozdziale.

Dwukrotne zastosowanie bramki Hadamarda na wektorach stanów bazowych wygląda następująco:

$$\begin{aligned} H_1H_1|0\rangle = H_1\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right) = \\ = \frac{1}{\sqrt{2}}H_1|0\rangle + \frac{1}{\sqrt{2}}H_1|1\rangle = \\ = \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) + \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |0\rangle \end{aligned} \tag{3.25}$$

$$\begin{aligned} H_1H_1|1\rangle = H_1\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right) = \\ = \frac{1}{\sqrt{2}}H_1|0\rangle - \frac{1}{\sqrt{2}}H_1|1\rangle = \\ = \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) - \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |1\rangle \end{aligned} \tag{3.26}$$

dochodzi do powrotu do stanów początkowych. Operacja przekształcenia dla bramki Hadamarda jest więc odwracalna. Bramka ta jednak nie posiada swojego odpowiednika wśród bramek klasycznych.

W sposób bardzo intuicyjny możemy zdefiniować klasyczną bramkę *NOT*. Jeżeli przyjmiemy, jako prawdę $|1\rangle$, a fałsz $|0\rangle$ to może mieć ona poniższą postać.

$$NOT = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{3.27}$$

Po takim określeniu bramka NOT będzie działała na kubit następująco:

$$\begin{aligned} NOT|0\rangle = |1\rangle \\ NOT|1\rangle = |0\rangle \end{aligned} \tag{3.28}$$

Inną bardzo często wymienianą bramką jest $\sqrt{NOT}$. Nazywana jest *'pierwiastkiem kwadratowym z negacji'*.

$$\sqrt{NOT} = \frac{1}{2}\begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix} \tag{3.29}$$

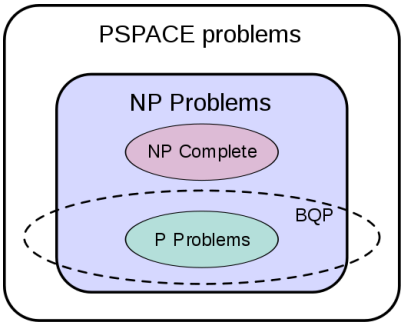
Złożenie w obwodzie kwantowym dwóch operacji $\sqrt{NOT}$ daje ten sam efekt, co zastosowanie jednej operacji NOT.

3.5. Ograniczenia komputera kwantowego

Z możliwości symulacji komputera kwantowego wynika, że nie może on zrealizować jakiegokolwiek zadania, którego nie zdoła wykonać komputer klasyczny symulujący jego działanie. Nie rozwiążemy więc za pomocą obliczeń kwantowych nowych problemów, które są *nieobliczalne* w tradycyjnej informatyce. Istnieje jednak wiele zagadnień, które pomimo, że możemy wykonywać na klasycznych komputerach czas ich realizacji wykraczałby poza życie człowieka lub nawet istnienie wszechświata. Jednymi z klasycznych przykładów są problem komiwojażera lub faktoryzacji liczb naturalnych. O ile w pierwszym przypadku informatyka próbuje sobie radzić wykorzystując algorytmy probabilistyczne [92] lub nawet genetyczne [95] to w przypadku faktoryzacji nie jesteśmy w tak komfortowej sytuacji[6].

Komputery kwantowe mogą jednak zwiększyć możliwości zastosowania obliczeń do zagadnień, dla których aktualne maszyny nie zwróciłyby wyników w racjonalnym czasie. Nie wiadomo czy za pomocą komputera kwantowego można rozwiązać problem komiwojażera w czasie wielomianowym, jednak wiadomo, że np. faktoryzacja nie stanowi dla niego problemu. Klas złożoności problemów rozpatrywanych na maszynie kwantowej jest kilka. Jednym z tych o którym warto wspomnieć jest BQP *(ang. bounded error quantum polynomial time)* [3]. Zawarte w tej klasie problemy można rozwiązać na komputerze kwantowym w czasie wielomianowym z prawdopodobieństwem błędu wynoszącym co najwyżej $\frac{1}{3}$ dla danego przebiegu. W klasycznej teorii złożoności istnieje analogiczna klasa BPP *(ang. bounded-error probabilistic polynomial)* [159] mająca za odniesienie probabilistyczną maszynę Turinga [132].

W pracy z 2000 roku [181] Michael Nielsen i Isaac Chuang proponują umiejscowienie klasy BQP względem P i NP tak jak przedstawiono na Rys. 3.1.



Rys. 3.1. Umiejscowienie klasy BQP względem innych klas złożoności problemów. *Źródło: Wikipedia.org*

[6]Problem ten został poruszony podczas omawiania algorytmu Shora.

Sprawa jest jednak nadal otwarta - znamy niewiele algorytmów kwantowych i brakuje dowodów na temat tego co faktycznie mógłby zrobić komputer kwantowy *(nie licząc tych kilku fenomenalnie przyśpieszających rozwiązań).*

Podczas opisu kubitu została wskazana bardzo ciekawa cecha tego układu - *superpozycja amplitud*. Z punktu widzenia fizyki jednak bardzo ciężko zbudować jakikolwiek prawidłowo funkcjonujący układ kwantowy ze względu na oddziaływanie obiektów kwantowych z otoczeniem. Stany będące superpozycją są bardzo nietrwałe i układ często *przeskakuje* do jednego z możliwych stanów bazowych. Zjawisko to nazywa się dekoherencją. Czas w jakim można utrzymywać układ w wymaganym stanie wacha się od nanosekund do sekund [65] *(dla technologii NMR [12] i MRI [220])*, a w przypadku rozwiązań optycznych wynosi jeszcze mniej.

Dużym problemem jest także budowa układów złożonych z dużej ilości kubitów. Jak oszacowano [70] do przeprowadzenia faktoryzacji liczby o długości 1000 bitów potrzeba od 10^4 do 10^7 kubitów *(w zależności od zastosowania korekcji błędów)* co znacznie przekracza dzisiejsze możliwości techniczne[7].

Największym osiągnięciem jakie udało się oficjalnie osiagnąć za pomocą obliczeń kwantowych jest wygenerowanie kilku liczb Ramseya na 84-kubitowym procesorze [285].

3.6. Kwantowa transformata Fouriera

Kwantowa transformata Fouriera[8] jest w obliczeniach kwantowych odpowiednikiem dyskretnej transformaty Fouriera. FFT została szczegółowo zaprezentowana w [14].

Kwantowa transformata Fouriera została zdefiniowana m. in. w [35] oraz [104].

Możemy w następujący sposób zdefiniować przekształcenie wykonujące kwan-

[7] Zbudowano przynajmniej kilka prostych systemów realizujących fizycznie obliczenia kwantowe. Pierwszym z nich był 7-kubitowy komputer kwantowy opracowany przez grupę informatyków z IBM i Uniwersytetu Stanford, który dokonał faktoryzacji liczby 15 [265]. Dwa lata później firma D-Wave Systems zaprezentowała znacznie bardziej zaawansowany procesor posiadający aż 128 kubitów. W 2011 roku korporacja amerykańskiego przemysłu obronnego Lockheed Martin zakupiła za 10 milionów dolarów tego typu sprzęt z aparaturą kriogeniczną. Dodatkowo podpisano wieloletni kontrakt obejmujący obsługę oraz opracowywanie algorytmów [195]. Można jedynie domyślać się, że nad maszyną kwantową pracuje wiele rządów i agencji na świecie - opracowanie jej w odcięciu od osób postronnych dałoby dużą przewagę technologiczną nad organizacjami, które takiego sprzętu nie posiadałyby.

[8] Algorytm Shora bazuje na znajdowaniu rzędu elementu z w grupie $\mathbb{Z}_N^*$. Do rozwiązywania tego typu problemów związanych z periodycznością danej funkcji często stosowane są metody teorii szeregów Fouriera.

tową transformatę na stanie bazowym $|x\rangle$:

$$|x\rangle \rightarrow \frac{1}{\sqrt{n}} \sum_{k=0}^{2^{n-1}} e^{\frac{2\pi i x k}{2^n}} |b\rangle \tag{3.30}$$

Macierz wykonująca taką transformację oczywiście musi mieć rozmiary 2^n x 2^n, a wartość odpowiedniego elementu można policzyć ze wzoru

$$M_{j,k} = \frac{1}{\sqrt{2^n}} e^{\frac{2\pi i j k}{2^n}} \tag{3.31}$$

gdzie j oznacza numer kolumny, a k wiersza.

Aby zbudować układ kwantowy realizujący tego typu przekształcenia potrzebujemy wcześniej zdefiniowanych bramek Hadamarda oraz bramki wprowadzającej warunkowo czynnik fazowy

$$e^{\frac{\pi i}{2^{k-j}}} \tag{3.32}$$

Czynnik ten powinien zostać wprowadzony tylko w przypadku, gdy w układzie kubity o numerach l i k mają wartość *1*.

Odwzorowanie to więc powinno wyglądać następująco:

$$\begin{aligned} |0\rangle|0\rangle &\rightarrow |0\rangle|0\rangle \\ |0\rangle|1\rangle &\rightarrow |1\rangle|0\rangle \\ |1\rangle|0\rangle &\rightarrow |1\rangle|0\rangle \\ |1\rangle|1\rangle &\rightarrow e^{\frac{\pi i}{2^{k-j}}} |1\rangle|1\rangle \end{aligned} \tag{3.33}$$

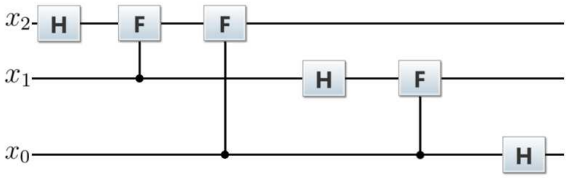
Macierz reprezentująca tą bramkę musi zatem być postaci:

$$F = \frac{1}{2} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{\pi i}{2^{k-j}}} \end{bmatrix} \tag{3.34}$$

Kwantową transformatę Fouriera można skonstruować jako złożenie określonej ilości tych dwóch bramek w układ:

$$H_{m-1}(F_{m-2,m-1}, H_{m-2}) \ ... \ (F_{0,m-1} \ ... \ F_{0,2}, F_{0,1}, H_{m-2}) \tag{3.35}$$

Schemat układu realizującego odpowiedni obwód kwantowy zaprezentowano na Rys. 3.2.



Rys. 3.2. Schemat układu realizującego kwantową transformatę Fouriera. *Źródło: Wykonanie własne.*

4. Struktury programowalne wykorzystywane w obliczeniach kwantowych

Prezentowane w tym rozdziale przykładowe kody są kompatybilne z językiem ANSI C [148] lub Fortran 95 [32].

4.1. Kubit i rejestry kwantowe

Najmniejszą jednostkę informacji kwantowej, kubit można reprezentować w języku C lub Fortran 95 za pomocą struktury. Przykłady zaprezentowano w Listingu 1. *(dla C)* i Listingu 2. *(dla Fortran 95)*.

```c
struct complex_float
{
    float real;
    float imag;
};

struct qubit
{
    complex_float alfa;
    complex_float beta;
};
```

Listing 1. Przykładowa struktura reprezentująca kubit w języku C *(dla typu bazowego float)*.

```
TYPE complex_float
    REAL            :: re
    REAL            :: img
END TYPE complex_float

TYPE complex_float
    complex_float   :: alfa
    complex_float   :: beta
```

```
9 END TYPE complex_float
```

Listing 2. Przykładowa struktura reprezentująca kubit w języku Fortran 95 *(dla typu bazowego real).*

Można wykorzystywać typ specjalny *complex*, który jest dostępny dla nowszych wersji języka C. Struktury te mogą wydawać się optymalne jednak pewną komplikacją okazuje się dostęp do określonych pól *'alfa'*, *'beta'* w przypadku większych rejestrów. Najprościej wykorzystać zwykłą tablicę liczb zespolonych *(lub nawet czasem można ograniczyć się do przeprowadzania operacji na liczbach rzeczywistych - nie będziemy mogli jednak w tym zbiorze wykonać wszystkich obliczeń)* i odwoływać się do kolejnych jej indeksów jak do kolejnych wartości współczynników kubitu lub całego rejestru kwantowego.

Na Listingach 3 i 4 przedstawiono przykładowe funkcje odpowiadające za pomiar stanu kubitu.

```
1  qubit_measurement(float qubit[])
2  {
3      double random_value;
4
5      random_value = rand() / (double) RAND_MAX;
6
7      if(random_value < pow(qubit[0]))
8      {
9          qubit[0] = 1;
10         qubit[1] = 0;
11     }
12     else
13     {
14         qubit[0] = 0;
15         qubit[1] = 1;
16     }
17 }
```

Listing 3. Implementacja funkcji pomiaru pojedynczego kubita w języku C i liczb rzeczywistych.

```
1  qubit_measurement(complex_float qubit[])
2  {
3      double random_value;
4
5      random_value = rand() / (double) RAND_MAX;
6
7      if(random_value < pow(qubit[0]))
8      {
9          qubit[0].real = 1;
10         qubit[0].imag = 0;
11         qubit[1].real = 0;
12         qubit[1].imag = 0;
13     }
14     else
15     {
```

```
16          qubit[0].real = 0;
17          qubit[0].imag = 0;
18          qubit[1].real = 1;
19          qubit[1].imag = 0;
20      }
21  }
```

Listing 4. Implementacja funkcji pomiaru pojedynczego kubita w języku C i typu complex_float.

Funkcje te generują ułamek losowy oraz sprawdzają jaką wartość względem niego kubit powinien przyjąć. Na wejściu kubit nie ma ustalonego stanu - jest opisany przez amplitudy prawdopodobieństwa zawarte w przekazywanej tablicy. Procedura redukuje stan kubitu do jednego z możliwych stanów uwzględniając prawdopodobieństwo wystąpienia poszczególnych z nich.

Znając reprezentację rejestru kwantowego możemy spróbować w określony sposób ustawić jego stan. Funkcja zaprezentowana w Listing 5 inicjuje układ kwantowy w ten sposób, aby wszystkie jego wartości były równie prawdopodobne. Po wywołaniu tej funkcji na wektorze n-elementowym będzie on reprezentował rejestr kwantowy $|\psi\rangle$

$$|\psi\rangle = \frac{1}{\sqrt{2^n}}(|\phi_0\rangle + |\phi_1\rangle + ... + |\phi_{2^n}\rangle) \tag{4.1}$$

gdzie ϕ_i oznacza i-tą możliwą wartość jaką może przyjąć układ.

```
1  void qregister_set_average_sp(complex_float q_register[], unsigned int qubits)
2  {
3      unsigned long long int quantum_register_size;
4      quantum_register_size = (unsigned long long int) pow(2, qubits);
5
6      double probability;
7      probability = pow(number, -.5);
8
9      unsigned long long int i;
10
11     for (i = 0 ; i <= number; i++)
12     {
13         q_register[i].real = prob;
14         q_register[i].imag = 0;
15     }
16 }
```

Listing 5. Implementacja funkcji ustawienia rejestru kwantowego z wszystkimi równie prawdopodobnymi stanami w języku C i typu complex_float.

Taki sam efekt można uzyskać zerując rejestr kwantowy, a następnie wykonując na nim przekształcenie za pomocą bramki Hadamarda.

Do kompletu elementarnych funkcji brakuje jeszcze możliwości pomiaru stanu rejestru kwantowego. Możliwa implementacja tej operacji została przedstawiona na Listingu 6.

```
qregister_measurement_sp(complex_float qubit[], int qubits)
{
    unsigned long long int done = 0;
    unsigned long long int index_value = -1;

    double random_value, a, b;

    unsigned long long int quantum_register_size;
    quantum_register_size = (unsigned long long int) pow(2, qubits);

    random_value = rand() / (double) RAND_MAX;

    a = b = 0;

    unsigned long long int i;
    unsigned long long int j;

    for (i = 0 ; i < quantum_register_size; i++)
    {
        if ( !done )
        {
            b += pow(q_register[i].real, 2) + pow(q_register[i].imag, 2);
            if (b > random_value && random_value > a)
            {
                for (unsigned long long int j = 0; j < quantum_register_size; j↩
                    ++)
                {
                    q_register[j].real = 0;
                    q_register[j].imag = 0;
                }

                q_register[i].real = 1;
                q_register[i].imag = 0;

                index_value = i;
                done = 1;
            }
            a += pow(q_register[i].real, 2) + pow(q_register[i].imag, 2);
        }
    }
    return index_value;
}
```

Listing 6. Funkcja pomiaru rejestru kwantowego w języku C i typu complex_float.

Powyższa funkcja dokonuje pomiaru rejestru i ustawia jego wartość. Dodatkowo zwracany jest numer indeksu stanu bazowego, na którym ustawiono układ. W przypadku, gdy funkcja zwróci wartość *-1* pomiar nie został wykonany prawidłowo i stan wektora nie uległ zmianie. W tym przypadku najprawdopodobniej nie jest spełniony konieczny warunek normalizacji *(dane wejściowe są błędne)*.

4.2. Symulacja przejścia rejestru przez bramki kwantowe

Operację przekształcenia układu kwantowego zdefiniowanego przez jedną z bramek kwantowych można przeprowadzić przy pomocy mnożenia wektora przez ma-

cierz. W wyniku tej operacji otrzymamy wektor wyjściowy, który będzie rejestrem kwantowym po odpowiednim przekształceniu. Jedną z możliwych realizacji tej operacji przedstawiono w Listingu 7.

```
void matrix_mul_vector_sp(complex_double matrix_A[], complex_double vector_input[], complex_double vector_output[], unsigned long long int size)
{
    unsigned long long i;
    unsigned long long j;

    for(i = 0; i < size; i++)
    {
        vector_output[i].real = 0;
        vector_output[i].imag = 0;
    }

    complex_double temporary_value;

    for (i = 0; i < size; i++)
    {
        for (j = 0; j < size; j++)
        {
            temporary_value = complex_mul(matrix_A[i*size+j], vector_input[j]);

            vector_output[i] = complex_add(vector_output[i], temporary_value);
        }
    }
}
```

Listing 7. Funkcja realizująca mnożenie wektora przez macierz dla typu complex_double w języku C.

W funkcji zastosowano zdefiniowane procedury *complex_mul* oraz *complex_add*, które wykonują operacje na liczbach zespolonych.

Po wprowadzeniu niezbędnych pojęć możemy wykonać pierwszą, elementarną symulację. Jej przykład został przedstawiony na Listingu 8.

```
//...
complex_double *q_register;
complex_double *not_matrix;

unsigned long long int result;

unsigned int qubits;
scanf("%lld", &qubits);

unsigned long long int register_size;
register_size = (unsigned long long int) pow(2, qubits);

q_register = (complex_double*) malloc(sizeof(complex_double) * register_size );
not_matrix = (complex_double*) malloc(sizeof(complex_double) * register_size * register_size);

qregister_set_average_sp(q_register, qubits);
matrix_not_init_sp(not_matrix, qubits);
```

```
18
19 matrix_mul_vector_sp(q_register, not_matrix, q_register, qubits);
20
21 result = qregister_measurement_sp(q_register, qubits);
22
23 printf("Result: %lld\n", result);
24
25 free(q_register);
26 free(not_matrix);
27 //...
```

Listing 8. Implementacja pierwszego obwodu kwantowego symulującego przejście układu przez kwantową bramkę *NOT*

Po podaniu przez użytkownika liczby kubitów, z której ma być złożony układ kwantowy, następuje alokacja odpowiedniej ilości miejsca w pamięci komputera oraz realizacja przejścia rejestru przez bramkę, czyli mnożenie wektora przez macierz. Ostatecznie zostaje przeprowadzony pomiar i na ekranie użytkownik zobaczy jego wynik.

Symulacje tego typu na dużych rejestrach są jednak problematyczne. W raporcie [184] przedstawiono wyniki czasowe transformacji Hadamarda i pomiaru rejestru złożonego z 33 kubitów. Na maszynie JUGENE IBM BlueGene/P [227] i 128 procesorach procedura ta trwała prawie 150 sekund.

4.3. Kwantowa transformata Fouriera w algorytmie Shora

Na temat implementacji kwantowej transformaty Fouriera napisano wiele prac [276] [276]. W ninejszej pracy w części dotyczącej algorytmu Shora i zastosowanej w niej kwantowej transformaty Fouriera głównie będę opierał się na publikacji Matthewa Haywarda z 2005 roku [107]. W artykule Hayward kwantową transformatę Fouriera sprowadza do operacji:

$$\frac{1}{\sqrt{2^n}} \sum_{a=0}^{2^{n-1}} \frac{1}{\sqrt{2^n}} \sum_{c=0}^{2^{n-1}} |c,k\rangle \cdot e^{\frac{2\pi ac}{2^n}} \tag{4.2}$$

gdzie k jest wartością z przedziału od 0 do $2^n - 1$ i spełniona jest równość

$$x^a \equiv k \ (mod \ n) \tag{4.3}$$

Ulepszona implementacja procedury została dołączona w załączniku A *(w wersji zaimplementowanej dla C oraz Fortran 95)*. Większość zmian polegała na maksymalnej redukcji ilości operacji zmiennoprzecinkowych w pętlach. Dodatkowo także w języku C zmieniono wywołanie dwóch funkcji *sin* i *cos* na pojedynczą *sincos*. Większość układów sprzętowych posiada koprocesor potrafiący w jednym cyklu zegara zrealizować obie funkcje dla tych samych argumentów. W języku Fortran 95 w do-

stępnych bibliotekach nie miałem dostępu do analogicznej funkcji *SINCOS* dostępnej dla C / C++.

Różnice w szybkościach implementacji całości procedur w języku *C* przedstawiono w Tabeli 4.1.

Tab. 4.1. Czasy wykonywania sekwencyjnego QFT *(cały program w C)* wykorzystującego algorytm Shora do faktoryzacji liczby wejściowej 511, jednego rdzenia procesora Intel Core 2 Quad Q8200 oraz kompilatora g++ w wersji 4.4.5.

Hayward QFT				
Min	Max	Avg	Sum	Optymalizacja
384.03 s	14016.49 s	1641.13 s	410282.01 s	Nie
282.08 s	10412.36 s	1157.53 s	289381.58 s	Tak
Swierczewski QFT				
Min	Max	Avg	Sum	Optymalizacja
111.89 s	14098.82 s	754.28 s	188570.00 s	Nie
80.51 s	6770.61 s	545.79 s	136446.56 s	Tak

Dodatkowo także kwantową transformatę Fouriera przepisano do języka *Fortran 95*. Podlegała ona kompilacji za pomocą kompilatora *gfortran*, a następnie linkowaniu z programem głównym za pomocą *g++*. Wyniki dla takiego rozwiązania zaprezentowano w Tabeli 4.2.

Tab. 4.2. Czasy wykonywania QFT *(wersja zaimplementowana w Fortran 95)* wykorzystującego algorytm Shora do faktoryzacji liczby wejściowej 511, jednego rdzenia procesora Intel Core 2 Quad Q8200 oraz kompilatora g++ w wersji 4.4.5 oraz gfortran 4.4.5.

Swierczewski QFT				
Min	Max	Avg	Sum	Optymalizacja
161.35 s	20388.39 s	1481.62 s	370405.10 s	Nie
74.44 s	9405.09 s	463.64 s	115909.90 s	Tak

W obu przypadkach testy wykonano z włączoną optymalizacją dla kodu *(optymalizacja drugiego stopnia i wsparcie dla architektury nocona)* lub też z wyłączoną jakąkolwiek optymalizacją.

Czas realizacji algorytmu jest w dużym stopniu uzależniony od wartości losowych, które są generowane na początku algorytmu. Z tego też powodu do porównań algorytmu nie ograniczono się do pojedynczego wywołania procedury *(wtedy czasy uzależnione byłyby od losowości)*. Testy przeprowadzano za każdym razem 250 razy, co w takiej dużej puli prób może dać dobry pogląd na różnice między rozwiązaniami. W tabelach wyróżniono najniższy, średni oraz najwyższy uzyskany czas pracy proce-

sora nad algorytmem. Dodatkowo także zliczono czas całkowity dla wszystkich 250 symulacji.

Analizując bardziej szczegółowo pierwotną wersję Haywarda można zauważyć, że losowość występuje z powodu zastosowania jednego bloku instrukcji warunkowej *if*. Schemat kodu zaprezentowano w Listingu 9.

```
1  // ...
2  for (a = 0 ; a < q ; a++)
3  {
4      if ((pow(q_register[a].real, 2) + pow(q_register[a].imag, 2)) > epsilon)
5      {
6          for (c = 0 ; c < q ; c++)
7          {
8
9              // Instructions
10
11         }
12     }
13 }
14 // ...
```

Listing 9. Szkielet kwantowej transformaty Fouriera w języku C.

Problematyczna instrukcja *if* znajduje się w 4 linijce uproszczonego kodu. Dzięki niej nie we wszystkich przypadkach będziemy wykonywać operacje w pętli wewnętrznej[9]. Wartości znajdujące się w tablicy *q_register*, która reprezentuje rejestr kwantowy są uzależnione od wartości losowej oraz wzorów 4.2 i 4.3. W najgorszym przypadku wykonamy więc q^2 iteracji[10]. W pętli wewnętrznej wykonujemy obliczenia tylko na liczbach zespolonych w *q_register*, które jednak mogą w istotny sposób wpłynąć na wyniki, ponieważ ich wartość jest znaczna *(większa niż zdefiniowany epsilon)*. Jeżeli usuniemy instrukcję warunkową z kodu to czas wykonywania będzie stały. Zestawienie czasów realizacji przedstawiono w Tabeli 4.3.

Tab. 4.3. Czasy wykonywania QFT *(z pominięciem instrukcji warunkowej)* wykorzystującego algorytm Shora do faktoryzacji liczby wejściowej 511, jednego rdzenia procesora Intel Core 2 Quad Q8200 oraz kompilatora g++ w wersji 4.4.5 i ewentualnie gfortran 4.4.5.

Swierczewski QFT		
Time	Optymalizacja	Język
25345.43 s	Nie	C
15452.21 s	Tak	C
28573.64 s	Nie	Fortran 95
17491.51 s	Tak	Fortran 95

[9]Tylko wtedy, gdy długość wektora będzie wyższa niż zdefiniowany przez nas epsilon.
[10]Co jest wartością bardzo dużą, ponieważ $q = 2^n$, gdzie n jest ilością kubitów w rejestrze.

Wszystkie zaprezentowane w tym rozdziale realizacje opierają się na liczbach zmiennoprzecinkowych podwójnej precyzji.

Gdy porównujemy czas realizacji pierwotnej wersji Haywarda z kodem po modyfikacjach możemy zauważyć, że w przypadku braku optymalizacji kompilatora, przyspieszenie uzyskane dzięki modyfikacjom wynosi w przybliżeniu *2.17*. Po włączeniu optymalizacji lekko spada do poziomu ok. *2.12 (porównanie dla czasów średnich)*. Jest to różnica dość znaczna - jeżeli uwzględnimy optymalizację czas wykonywania 250 symulacji dla liczby 511 zmniejsza się z ponad 80 do 39 godzin.

Analizując Tabele 4.1 i 4.2 można zauważyć, że pomimo braku w zastosowanych bibliotekach języka Fortran 95 funkcji *SINCOS* wersja QFT napisana w tym języku wykonuje się szybciej od pełnej w C.

Oznacza to tylko tyle, że kompilator sam optymalizuje ten fragment kodu - mało tego, optymalizuje on inne operacje lepiej od *g++*.

Wyniki zaprezentowane w tym dziale są jedynie cząstką ze wszystkich przeprowadzonych doświadczeń. Część właściwa została opisana w rozdziale 7.

5. Model algorytmu kwantowego

Projektowanie algorytmów kwantowych nie jest tak proste i intuicyjne jak w przypadku klasycznych algorytmów. Ich model jest obarczony pewnymi ograniczeniami, które mogą tworzyć trudną do przekroczenia barierę dla programisty.

5.1. Sieci bramek kwantowych

Znając struktury oraz operacje elementarne w informatyce kwantowej musimy wiedzieć jaką postać powinien mieć algorytm kwantowy. Współcześnie najczęściej stosowane są sieci bramek kwantowych, które nie odbiegają daleko od swoich klasycznych odpowiedników. W informatyce klasycznej wyniki z jednej bramki mogą być przekazane do wielu innych bramek. W informatyce kwantowej tego typu operacja jest niedozwolona - nie można kopiować stanu kubitu na inny kubit.

Operator działający na rejestr kwantowy musi być liniowy[11] i unitarny[12]. Możemy przyjąć istnienie takiego operatora X, że dla dowolnego wektora $|\phi\rangle$

$$X|\phi, 0\rangle = |\phi, \phi\rangle \tag{5.1}$$

Oznacza to jednak, że

$$X\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes 0\rangle = X\frac{1}{\sqrt{2}}(|0,0\rangle + |1,1\rangle) = \tag{5.2}$$
$$= \frac{1}{\sqrt{2}}(|0,0\rangle + |1,1\rangle)$$

możemy z przekształceń uzyskać także

$$X\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes 0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \tag{5.3}$$
$$= \frac{1}{2}(|0,0\rangle + |0,1\rangle + |1,0\rangle + |1,1\rangle)$$

Zatem taki operator X nie może istnieć ponieważ

$$\frac{1}{2}(|0,0\rangle + |0,1\rangle + |1,0\rangle + |1,1\rangle) \neq \frac{1}{\sqrt{2}}(|0,0\rangle + |1,1\rangle) \tag{5.4}$$

[11] W algebrze liniowej funkcja między przestrzeniami liniowymi *(nad ustalonym ciałem)* zachowująca ich strukturę; z punktu widzenia algebry jest to zatem homomorfizm przestrzeni liniowych nad ustalonym ciałem.

[12] W analizie funkcjonalnej, operator normalny którego złożenie z jego operatorem sprzężonym jest identycznością.

Problem ten zauważono w [271]. Stawia on główne ograniczenie podczas projektowania algorytmów kwantowych.

Układ kwantowy realizujący obliczenia musi być skończonym zbiorem bramek kwantowych, który operuje na określonej ilości kubitów. Cała procedura powinna zostać zdefiniowana dla każdej długości rejestrów wejściowych. Podczas szacowania złożoności obliczeniowej algorytmu brana jest pod uwagę wykorzystana ilość bramek kwantowych. Opis tego typu został zaproponowany w [47], a następnie rozwijany [169].

Możliwe jest jednak tworzenie operatorów kopiujących stan układu znajdującego się w jednym ze stanów bazowych *(np. w bazie standardowej)*.

Zdefiniujmy macierz XOR jako:

$$XOR = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{5.5}$$

Macierz ta, jeżeli przyjmiemy $x, y \in \{0, 1\}$, wykona procedurę

$$XOR|x, y\rangle = |x, y \otimes x\rangle \tag{5.6}$$

co w specyficznym przypadku, gdy y = 0, oznacza, że

$$XOR|x, 0\rangle = |x, x\rangle \tag{5.7}$$

Przekształcenie to można jednak wykonać tylko na układach już zredukowanych do jakiegoś stanu *(np. po pomiarze)*.

5.2. Problem szukania elementu w zbiorze – algorytm Grovera

Kwantowy algorytm poszukiwania w zbiorze danego elementu został zaproponowany przez amerykańskiego informatyka Lova Kumara Grovera w pracy z 1996 roku [102]. Grover zaproponował swój algorytm głównie w odniesieniu do możliwości komputera kwantowego podczas przeszukiwania rekordów w olbrzymich bazach danych. Zaproponowana przez niego procedura charakteryzuje się kosztem $O(\sqrt{N})$, gdzie jako N rozumiemy rozmiar przeszukiwanego zbioru. Jak dobrze wiemy w informatyce klasycznej, gdy bierzemy pod uwagę nieuporządkowany w jakikolwiek sposób zbiór danych optymalny algorytm musi wykonać $O(N)$ operacji elementarnych porównywania. Jesteśmy zmuszeni do sprawdzania wszystkich elementów od pierwszego aż do ostatniego. Komputery kwantowe mają więc olbrzymią przewagę

nad maszynami klasycznymi, co może być w przyszłości widoczne podczas przeszukiwania tabel olbrzymich baz danych.

Grover początkowo rozpatrywał jedynie przypadek zbioru, w którym tylko jeden element spełnia określony warunek. W tym samym roku opublikowano wyniki zespołu badawczego, które prezentują analizę zachowania algorytmu w momencie, gdy wiele elementów spełnia kryterium wyszukiwania [168]. Oszacowano także złożoność obliczeniową tej procedury.

Wiązano olbrzymie nadzieję z uzyskaniem jeszcze wyższego przyśpieszenia niż kwadratowe, jednak do dnia dzisiejszego nie znaleziono optymalniejszego rozwiązania problemu.

5.2.1. Definicja problemu

Zdefiniujmy przeszukiwany zbiór danych jako $\mathbb{S}$. Niech $\mathbb{S}$ będzie zbiorem N-elementowym. Istnieje także funkcja $f : \mathbb{S} \to \{0, 1\}$, która w naszym przypadku[13] musi spełniać warunek:

$$\exists!_{q \in \mathbb{S}} : f(q) = 1 \tag{5.8}$$

z czego wynika, że przyjmuje ona wartość *1* dokładnie raz.

Naszym zadaniem jest więc odnalezienie elementu q w zbiorze $\mathbb{S}$, dla którego zachodzi równość $f(q) = 1$.

W informatyce klasycznej możemy przeszukiwać zbiór o dowolnej wielkości. W jej kwantowym odpowiedniku jednak dla uproszczenia przyjmijmy, że przeszukujemy zbiór danych o wielkości maksymalnej jaką może pomieścić dany rejestr kwantowy, a więc

$$N = 2^n \tag{5.9}$$

gdzie n będzie liczbą naturalną definiującą ilość kubitów, z których jest zbudowany rejestr. W przypadku gdy N nie będzie spełniało równości 5.9, możemy bezproblemowo uzupełnić nasz zbiór $\mathbb{S}$ o dodatkowe nieznaczące elementy dla których $f(q) = 0$.

Zdefiniujmy także poszukiwany stan jako $|\omega_0\rangle$.

5.2.2. Algorytm

Algorytm składa się z szeregu iteracji, które zmieniają stan rejestru kwantowego i zwiększają prawdopodobieństwo, że podczas pomiaru rejestr ten zostanie zredukowany do poszukiwanego stanu $|\omega_0\rangle$. Pozostałe amplitudy stanu są zmniejszane w ten sposób, że warunek normalizacji wektora jest zawsze spełniony.

[13] Także w przypadku który rozpatrywał Grover w swojej pierwszej pracy z 1996 roku.

W momencie uzyskania najwyższego prawdopodobieństwa sukcesu zostaje przeprowadzony pomiar. W przypadku, gdy algorytm nie zwróci prawidłowej wartości, proces należy wykonać ponownie.

Początkowo układ kwantowy należy ustawić w stan superpozycji z równymi amplitudami. Operację tą możemy zapisać jako:

$$|\phi_0\rangle = \frac{1}{\sqrt{2}} \sum_{i=0}^{N-1} |i\rangle \tag{5.10}$$

Procedurę tę realizuje implementacja zaprezentowana w Listingu 5.

Następnie iteracyjnie jest wykonywana operacja:

$$|\phi_{n+1}\rangle = AB|\phi_n\rangle \tag{5.11}$$

gdzie

$$A = \mathbb{I} - 2|\omega_0\rangle\langle\omega_0| \tag{5.12}$$

$$B = 2|\phi_0\rangle\langle\phi_0| - \mathbb{I} \tag{5.13}$$

Przez $\mathbb{I}$ oznaczono macierz jednostkową.

Algorytm Grovera często jest nazywany *metodą wzmacniania Grovera* ze względu na swój charakter ciągłego wzmacniania amplitudy, która odpowiada rozwiązaniu.

5.2.3. Analiza procedury

Operator A zmienia fazę amplitudy układu $|\phi_i\rangle$ odpowiadającej poszukiwanemu stanowi $|\omega_0\rangle$. Pozostałe amplitudy nie ulegają zmianie.

$$\begin{aligned} A|\omega\rangle &= (\mathbb{I} - 2|\omega_0\rangle\langle\omega_0|)|\omega\rangle = \\ &= |\omega\rangle - 2|\omega_0\rangle\langle\omega_0|\langle\omega_0|\omega\rangle = \\ &= |\omega\rangle - 2|\omega_0\rangle \begin{cases} 1 \text{ dla } \omega = \omega_0 \\ 0 \text{ dla } \omega \neq \omega_0 \end{cases} = \\ &= \begin{cases} -|\omega\rangle \text{ dla } \omega = \omega_0 \\ |\omega\rangle \text{ dla } \omega \neq \omega_0 \end{cases} \end{aligned} \tag{5.14}$$

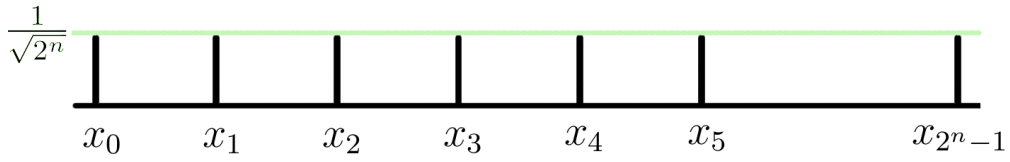
Operator B wykonuje obrót amplitud względem wartości średniej[14]. Można jego rozwinąć jako

$$\begin{aligned} B|\omega\rangle &= (2|\phi_0\rangle\langle\phi_0| - \mathbb{I})|\omega\rangle = \\ &= 2|\phi_0\rangle\langle\phi_0|\omega\rangle - |\omega\rangle = \end{aligned} \tag{5.15}$$

[14]Możliwy stan przed przekształceniem B zaprezentowano na Rys. 5.2, a stan po przekształceniu przedstawiono na Rys. 5.3.

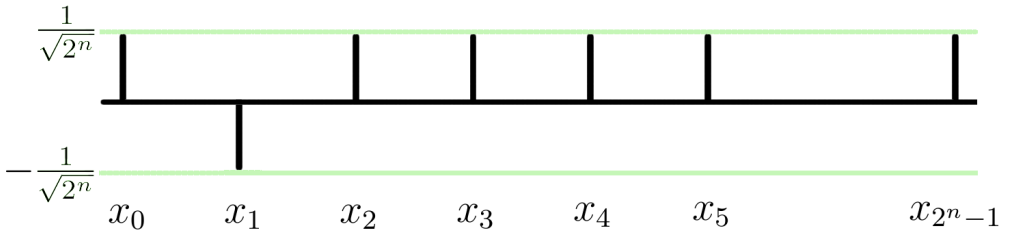
$$= \frac{1}{\sqrt{2}}|\phi_0\rangle - |\omega\rangle$$

Możemy spróbować w sposób wizualny przedstawić modyfikację układu kwantowego po kolejnych operacjach. Początkowo amplitudy w rejestrze muszą być ustawione w konfiguracji początkowej będącą równomierną superpozycją. Przedstawia to Rys. 5.1.



Rys. 5.1. Przedstawienie graficzne stanu rejestru po jego wstępnej inicjacji. *Źródło: Wykonanie własne.*

Kolejnym krokiem jest wykonanie przekształcenia A zmieniającego fazę poszukiwanej jednej z amplitud. Stan układu po przeprowadzeniu tej procedury został zaprezentowany na Rys. 5.2.



Rys. 5.2. Stan układu kwantowego po wykonaniu procedury A. *Źródło: Wykonanie własne.*

Po operacji A pozostaje nam do wykonania jedynie procedura B. W łatwy sposób możemy wyznaczyć dokładne wartości amplitud po jej wykonaniu. Przyjmijmy jako U wartość średnią układu. Można ją obliczyć w następujący sposób wykorzystując wzór na średnią arytmetyczną

$$U = \frac{1}{2^n}\sum_{i=0}^{2^{n-1}} x_i \tag{5.16}$$

Średnia amplitud po wykonaniu operacji A wynosi więc

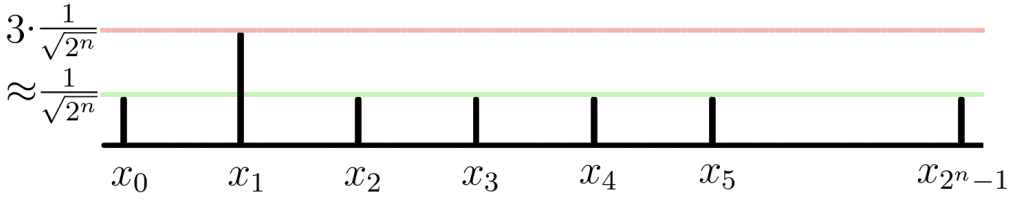
$$U = \frac{1}{2^n}\left((2^n-1)\frac{1}{\sqrt{2^n}} - \frac{1}{\sqrt{2^n}}\right) = \tag{5.17}$$
$$= \frac{1}{\sqrt{2^n}}\left(1 - \frac{2}{2^n}\right) = \frac{1}{\sqrt{2^n}} - \frac{2}{\sqrt{2^n}\cdot 2^n}$$

Możemy zaobserwować, że po wykonaniu tego przekształcenia średnia amplitud nie ulegnie znacznej zmianie. W tym momencie możemy obliczyć wartości amplitud naszego układu po wykonaniu przekształcenia B.

$$\frac{1}{\sqrt{2^n}} \xrightarrow{B} 2\left(\frac{1}{\sqrt{2^n}} - \frac{2}{\sqrt{2^n}\cdot 2^n}\right) - \frac{1}{\sqrt{2^n}} = \tag{5.18}$$
$$= \frac{2}{\sqrt{2^n}} - \frac{4}{\sqrt{2^n}\cdot 2^n} - \frac{1}{\sqrt{2^n}} =$$
$$= \frac{1}{\sqrt{2^n}} - \frac{4}{\sqrt{2^n}\cdot 2^n} \approx \frac{1}{\sqrt{2^n}}$$

$$-\frac{1}{\sqrt{2^n}} \xrightarrow{B} 2\left(\frac{1}{\sqrt{2^n}} - \frac{2}{\sqrt{2^n}\cdot 2^n}\right) + \frac{1}{\sqrt{2^n}} = \tag{5.19}$$
$$= \frac{2}{\sqrt{2^n}} - \frac{4}{\sqrt{2^n}\cdot 2^n} + \frac{1}{\sqrt{2^n}} =$$
$$= \frac{3}{\sqrt{2^n}} - \frac{4}{\sqrt{2^n}\cdot 2^n} \approx 3\cdot\frac{1}{\sqrt{2^n}}$$

Wynik ten został zaprezentowany na Rys. 5.3.



Rys. 5.3. Stan rejestru po przeprowadzeniu operacji B. *Źródło: Wykonanie własne.*

Należy zwrócić szczególną uwagę na fakt, że najlepszy klasyczny algorytm wykorzystujący wyszukiwanie probabilistyczne [282] [121] cechuje się prawie 4,5-krotnie gorszą skutecznością niż algorytm Grovera, który jak zaprezentowano po pojedynczej iteracji *(złożonej z dwóch procedur AB)* daje szanse sukcesu z prawdopodobieństwem bliskim $\frac{9}{2^n}$.

Można zadać pytanie: jak długo należy powtarzać iteracje, aby uzyskać największe prawdopodobieństwo sukcesu. Krzysztof Giaro i Marcin Kamiński [157] wskazują, że można określić równania rekurencyjne opisujące kolejne wartości kolejnych amplitud. Przyjmijmy, że amplitudę α_k będzie określała stan właściwy $|\omega_0\rangle$ w k-tej iteracji. Jako β_k zdefiniujmy natomiast wszystkie pozostałe stany, których będzie $N-1$ *(całkowity rozmiar rejestru pomniejszony o α)*.

Równania te wyglądają następująco:

$$\alpha_{k+1} = 2\frac{-\alpha_k + (N-1)\beta_k}{N} + \alpha_k = \left(1 - \frac{2}{N}\right)\alpha_k + \left(2 - \frac{2}{N}\right)\beta_k \tag{5.20}$$

$$\beta_{k+1} = 2\frac{-\alpha_k + (N-1)\beta_k}{N} + \beta_k = -\frac{2}{N}\alpha_k + \left(1 + \frac{2}{N}\right)\beta_k \tag{5.21}$$

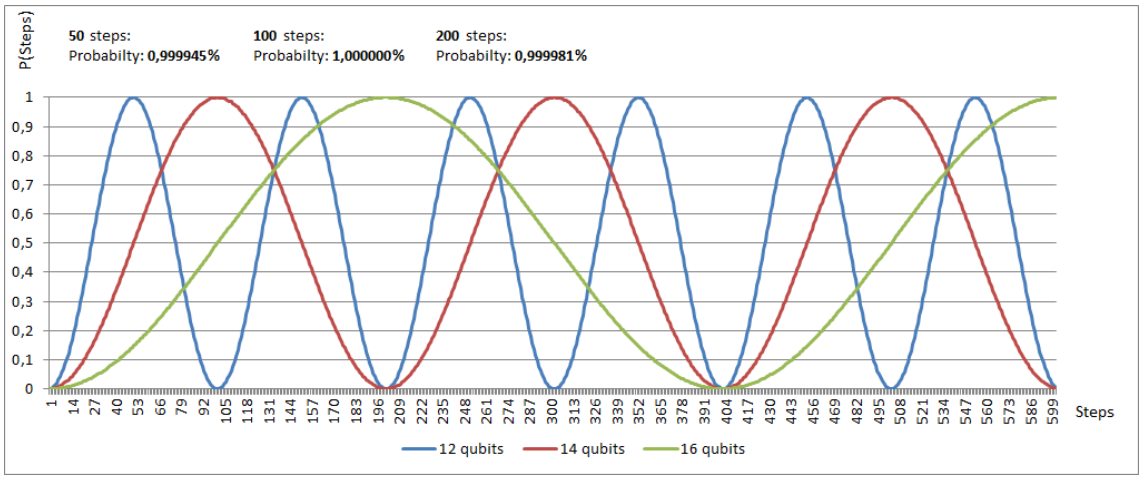
Wykorzystując równania 5.20 i 5.21 można ustalić prawdopodobieństwo z jakim po wykonaniu k iteracji i przeprowadzeniu odpowiedniego pomiaru system zostanie zredukowany do $|\omega_0\rangle$. Prawdopodobieństwo to wyznacza się według wzoru:

$$\sin^2\left((2k+1)arcsin\left(\sqrt{\frac{1}{N}}\right)\right) \tag{5.22}$$

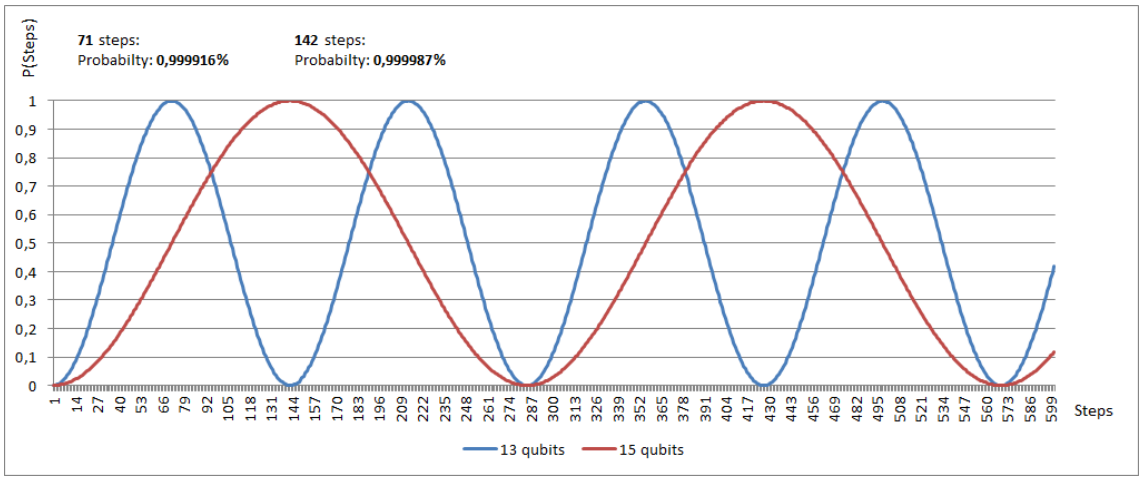
Przebieg określa funkcja trygonometryczna sin^2. Osiąga ona najbliższe maksimum przy wartości $\frac{\pi}{2}$ dlatego taką wartość najoptymalniej przyjąć jako jej argument.

$$(2k+1)arcsin\left(\sqrt{\frac{1}{N}}\right) = \frac{\pi}{2} \Rightarrow k = \frac{\frac{\pi}{4}}{arcsin\left(\sqrt{\frac{1}{N}}\right)} - \frac{1}{2} \tag{5.23}$$

Można więc zauważyć, że dla N $= 2^{13} = 8192$, algorytm musi wykonać 71 iteracji, a dla N $= 2^{15} = 32768$ dwa razy więcej - 141. Zbiór danych wzrósł więc czterokrotnie, ale złożoność obliczeniowa tylko dwukrotnie. Odpowiada to wstępnie zadeklarowanej złożoności rzędu $O(\sqrt{N})$. Odpowiednie przebiegi obrazujące ten proces można zobaczyć na Rys. 5.4 oraz Rys. 5.5.

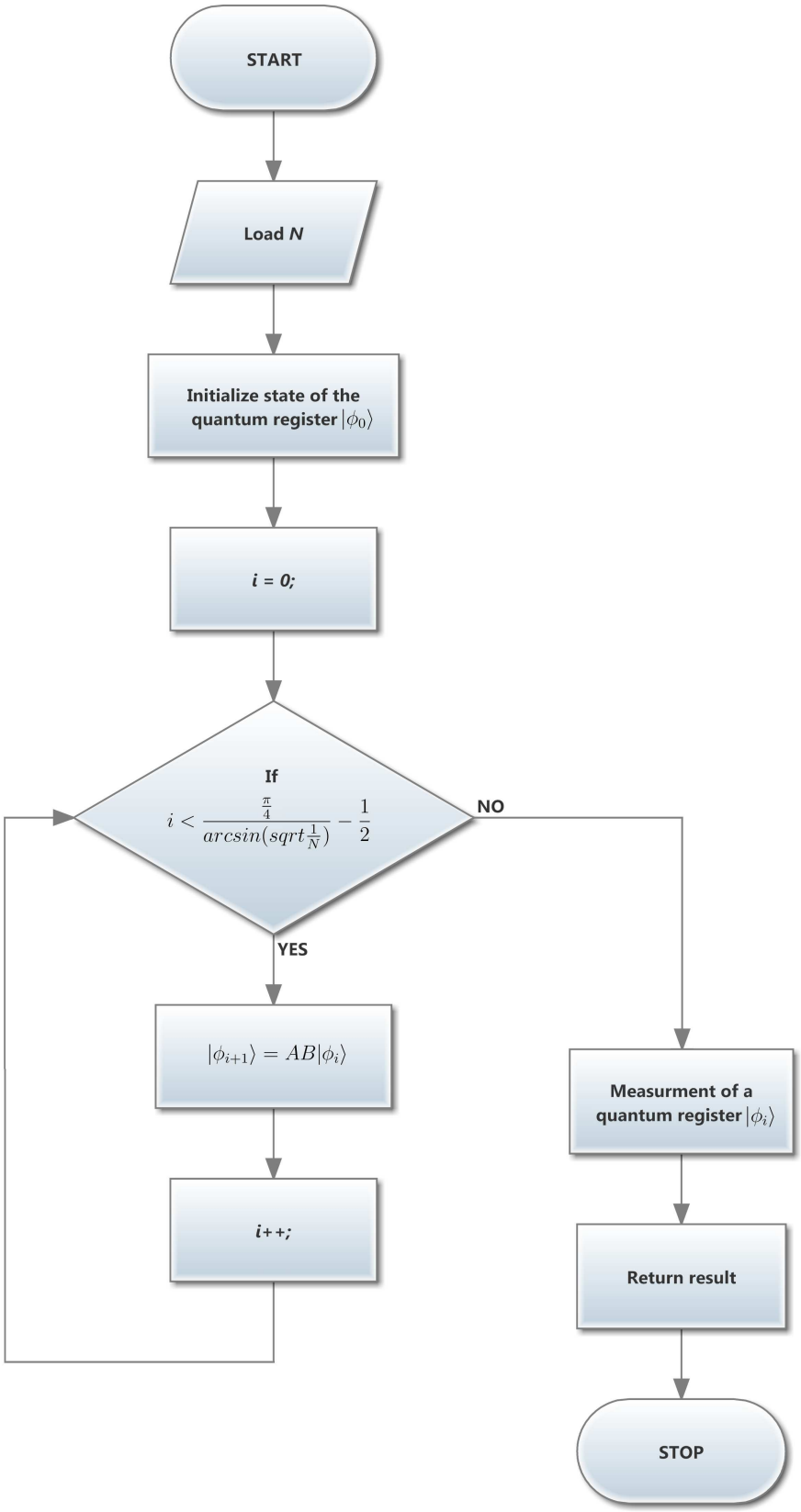


Rys. 5.4. Przebieg funkcji określającej prawdopodobieństwo prawidłowego pomiaru algorytmu Gorvera dla układów o rozmiarze 12, 14 i 16 kubitów. *Źródło: Wykonanie własne.*



Rys. 5.5. Przebieg funkcji określającej prawdopodobieństwo prawidłowego pomiaru algorytmu Gorvera dla układów o rozmiarze 13 i 15 kubitów. *Źródło: Wykonanie własne.*

Powyższe przebiegi obrazują to, że w przypadku algorytmów kwantowych nie musi być prawdą zasada według, której im dłużej prowadzimy obliczenia tym lepsze wyniki końcowe uzyskujemy. Obliczenia kwantowe są często przez autorów publikacji porównywane do pieczenia ciasta, które należy wyjąć z pieca ani nie za wcześnie, ani nie zbyt późno tak, aby było ono odpowiednio dopieczone. Schemat blokowy algorytmu Grovera będzie wyglądał więc tak jak na Rys. 5.6.



Rys. 5.6. Schemat blokowy algorytmu Grovera. *Źródło: Wykonanie własne.*

5.3. Problem rozkładu liczb naturalnych na czynniki pierwsze – algorytm Shora

Problemów z liczbami pierwszymi w matematyce jest wiele. Niektóre z nich mają swoje zastosowanie w innych dziedzinach, które są w dużo większym stopniu związane z inżynierią. Dobrym przykładem jest zagadnienie rozkładu liczb naturalnych na czynniki pierwsze *(proces faktoryzacji)*. Mając dwie liczby p i q można bardzo prosto obliczyć ich iloczyn $n = p \cdot q$. Na bazie najnowszego testu AKS[15] opublikowanego w 2002 roku w publikacji [171] taką procedurę wyznaczania n można przeprowadzić ze złożonością $O(log^{12+\epsilon}(p) + log^{12+\epsilon}(q))$. Rząd wielkości jest określany przez logarytm, więc procedura ta nie powinna sprawiać problemów. Jednak zagadnienie całkowicie odwrotne jest związane z komplikacjami. Panuje opinia, że nie istnieje algorytm klasyczny umożliwiający efektywnie rozwiązać problem faktoryzacji. Na tym problemie opiera się większość współczesnych kryptosystemów - w tym powszechnie dzisiaj wykorzystywany i uważany za bezpieczny RSA. 18 marca 1991 rozpoczął się konkurs RSA Factoring Challenge. Był on otwartymi zawodami zorganizowanymi przez RSA Security. Celem zawodów było pobudzenie do badań nad algorytmami służącymi do faktoryzacji liczb. Za złamanie szyfrów o określonych długościach *(złamanie szyfru RSA czyli rozłożenie określonej liczby n na czynniki pierwsze)* wyznaczono duże nagrody[16]. Pomimo rozwoju techniki i wielu prób zarówno pojedynczych użytkowników, jak i setek maszyn internautów wykorzystujących obliczenia rozproszone dużo szyfrów do dnia dzisiejszego nie udało się złamać. Wszelkie tego typu algorytmy cechują się złożonością wykładniczą. Najszybsza obecnie znana procedura to GNFS *(ang. general number field sieve)* [212]. Stosując algorytm GNFS zespół z Uniwersytetu w Bonn 2 listopada 2005 dokonał faktoryzacji liczby RSA-640 (posiada ona 193 cyfry w rozwinięciu dziesiętnym). Zadanie to trwało 5 miesięcy. Złożoność nawet tego algorytmu opisuje wzór [278]

$$e^{\left(\left(\sqrt[3]{\frac{64}{9}}+o(1)\right)(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}\right)} = L_n\left[\frac{1}{3}, \sqrt[3]{\frac{64}{9}}\right] \tag{5.24}$$

W 1994 roku Peter Shor przedstawił kwantowy algorytm faktoryzacji charakteryzujący się złożonością wielomianową [238]. Okazało się więc, że gdyby ktoś posiadał komputer kwantowy mógłby złamać wiele stosowanych obecnie szyfrów - także wcześniej wspominany RSA. Algorytm ten wymaga dokładnie $O(n^2 log^2 n)$ operacji kwantowych.

[15] Test pierwszości.

[16] Najwyższa z nich wynosiła 200 000 dolarów.

5.3.1. Algorytm

Ze względu na ograniczone miejsce w pracy nie będę bardzo szczegółowo omawiać tego algorytmu *(więcej informacji mozna odnaleźć m. in. w [21])*. W tym przypadku najpierw wprost zaprezentuję procedurę.

1. Wprowadź N - liczbę podlegającą procesowi faktoryzacji.
2. Sprawdź czy N jest potęgą liczby naturalnej. Jeżeli tak, zakończ pracę *(liczba będąca podstawą potęgi jest także dzielnikiem)*. W przeciwnym wypadku przejdź do następnego kroku.
3. Wybierz w sposób losowy element x *(*$x > 0$*)* z pierścienia $\mathbb{Z}_N$.
4. Oblicz $p = NWD(x,\ N)$. Jeżeli $p \neq 1$ to p jest nietrywialnym dzielnikiem N i zakończ algorytm. W przeciwnym wypadku przejdź do następnego kroku.
5. Oblicz rząd r elementu x w pierścieniu $\mathbb{Z}_N$.
6. Jeżeli r jest liczbą nieparzystą lub $x^{\frac{r}{2}} \equiv -1 (mod\ N)$ zakończ wykonywanie algorytmu. Procedura nie powiodła się. W przeciwnym wypadku przejdź do następnego kroku.
7. Oblicz $p = NWD(x^{\frac{r}{2}} - 1,\ N)$ i zwróć p jako wynik działania algorytmu.

Kroki *1-4* oraz *6-7* cechują się złożonością wielomianową. Najbardziej skomplikowaną procedurą wykorzystaną w tych punktach jest algorytm Euklidesa[17] za pomocą, którego możemy obliczyć największy wspólny dzielnik - jednak nawet jego można zrealizować w czasie wielomianowym. W czasie wielomianowym można także sprawdzić czy liczba n jest potęgą innej liczby naturalnej.

Problemy ze złożonością obliczeniową będziemy mogli mieć tylko w piątym kroku. Przyjmijmy, że chcemy wykonać faktoryzację liczby $N = 25$. W kroku trzecim wylosujemy dowolne $x \in \mathbb{Z}_N$ jednak x musi być różne od 0. Wylosowana liczba musi być względnie pierwsza z N, ponieważ gdyby tak nie było byłaby ona rozwiązaniem naszego problemu i obliczenia zostałyby zakończone. W kolejnym punkcie (5) występuje problem odnalezienia rzędu x w $\mathbb{Z}_N$. Wartości jakie w naszym przypadku może przyjąć x można więc ograniczyć do grupy reszt względnie pierwszych z N.

$$\mathbb{Z}^*_{25} = \{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 21, 22, 23, 24\} \tag{5.25}$$

ich rzędy wynoszą odpowiednio:

[17] Jest zastosowany w punkcie czwartym i siódmym.

$$1, 20, 20, 10, 5, 4, 20, 10, 5, 20, 20, 10, 5, 20, 4, 10, 5, 20, 20, 2$$

Liczby x oraz ich rzędy, które umożliwiają przejście do ostatniego kroku algorytmu zaznaczono kolorem czerwonym. Dla przykładu weźmy więc $x = 6$. Z tego wynika, że

$$p = NWD\left(6^{\frac{5}{2}} - 1, 25\right) = 5 \tag{5.26}$$

Co jest naszym ostatecznym rezultatem, ponieważ faktycznie 5|25.

Implementacja całego algorytmu w sposób klasyczny mija się całkowicie z celem. Przykładowy kod dodano do pracy w postaci załącznika B, a jego wyniki czasowe zostały przedstawiono w Tabeli 5.1.

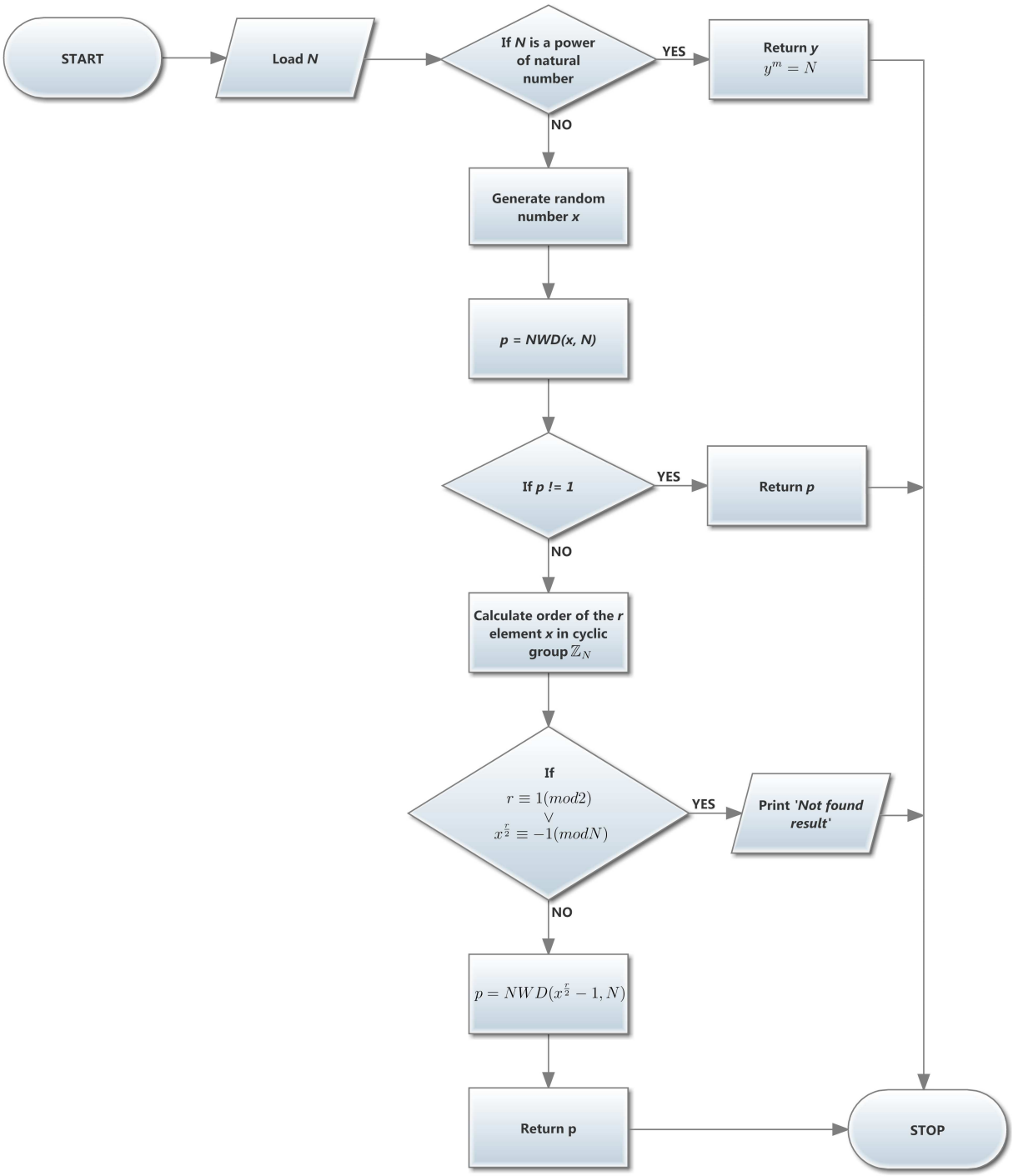
Tab. 5.1. Czas trywialnego rozkładu rzędów dla programu skompilowanego za pomocą kompilatora g++ w wersji 4.4.5 z wykorzystaniem optymalizacji drugiego stopnia oraz optymalizacji kodu pod architekturę *nocona*. Program wykonywany sekwencyjnie na jednym rdzeniu procesora Intel Core 2 Quad Q8200. *Źródło: Wykonanie własne*

N:	Czas realizacji:
25	0.002 s
250	0.003 s
2 500	0.058 s
25 000	3.459 s
250 000	252.728 s
2 500 000	25891.581 s
25 000 000	> 1887924.041 s

Wyniki czasowe realizacji dla faktoryzacji przekonują, że złożoność jest nadal wykładnicza i najprawdopodobniej dużo gorsza od innych opracowanych algorytmów. W poniższej tabeli przedstawiono czasy dla tej procedury i typu *double*. Należy jednak wziąć pod uwagę fakt, że procedura ta sprawdza wszystkie możliwe wartości x i ich rzędy. Można spróbować opracować bardziej efektywny algorytm wykonujący operacje tylko na określonych elementach. Nie jest to jednak celem tej pracy.

Krok odpowiadający za obliczanie rzędów w przypadku komputera kwantowego jest realizowany za pomocą kwantowej transformaty Fouriera.

Schemat blokowy algorytmu Shora zaprezentowano na Rys. 5.7.



Rys. 5.7. Schamt blokowy prezentujący algorytm Shora. *Źródło: Wykonanie własne.*

6. Techniki programowania równoległego

Współczesna inżynieria i nauka do rozwiązywania zaawansowanych zagadnień wymaga zastosowania bardzo *szybkich* komputerów. Wraz z rozwojem techniki możliwości komputerów pod względem mocy obliczeniowej znacznie wzrastają. UNIVAC I *(ang. Universal Automatic Computer)* [244]- maszyna zbudowana przez Johna Eckerta i Johna Mauchly z Eckert-Mauchly Computer Corporation w 1951 roku i często uznawana za pierwszy na świecie elektroniczny komputer ogólnego przeznaczenia przy wydajności rzędu 1 KFLOPS jest szesnaście bilionów $(16 \cdot 10^{12})$ razy wolniejszy od aktualnie najszybszego superkomputera Sequoia, który znajduje się w Lawrence Livermore National Laboratory i osiąga 16,32 PFLOPS[18] [141].

Do niedawna NASA wykorzystywała komputery dużej mocy do symulacji misji, w której zastosowanie znajdowały wahadłowce [23]. Z wykorzystaniem specjalistycznych pakietów do obliczeń dynamiki płynów wyznaczano rozkład ciśnień wokół statku kosmicznego. Innym sektorem w NASA, gdzie do bardzo trudnych zadań wykorzystuje się komputery są centra badawcze dotyczące jednostek napędów odrzutowych, gdzie symulowany jest m. in. przepływ płynnego paliwa tłoczonego przez pompę do jednostek napędowych [48].

W 1990 roku powstał projekt *Human Genome Project*, który miał na celu rozszyfrowanie tajemnic ludzkiego DNA. Do obliczeń w projekcie wykorzystano klaster obliczeniowy złożony z 27 węzłów po 4 procesory w każdym z nich i jednego 16-procesorowego komputera o architekturze NUMA *(ang. nonuniform memory access)*. Dane przechowywano na oddzielnym serwerze plików o pojemności 1 TB [42]. Obliczenia wykonywane sekwencyjnie trwałyby 20 000 godzin *(ponad 22 lata)* [266] jednak wstępne wyniki były znane już w 2000 roku, kiedy to podczas oficjalnego wystąpienia prezydent USA Bill Clinton oraz premier Wielkiej Brytanii Tony Blair ogłosili powstanie mapy ludzkiego genomu [38]. Po ponad roku - w maju 2001 roku opublikowano wyniki badań [266] [42].

Dużo słyszy się o badaniach dotyczących klimatu. Z pomocą metod numerycznych można symulować zmiany pogody. Oprogramowanie tego typu często jest oparte na CCSM3 *(ang. Community Climate System Model)* i opisuje takie czynniki jak stan atmosfery, pokryw lodowych, lądów i oceanów [39]. Dane dotyczące tych grup mogą być w sposób całkowicie niezależny przetwarzane przez różne grupy procesorów. Wymiana danych obejmuje jedynie określone przepływy *(np. mas)*. W swojej pracy z 2008 roku [68] dr John Drake wraz ze współpracownikami przeprowadził symulacje atmosfery przy użyciu komputerów marek Cray i IBM Cluster i wykazał możliwość jej modelowania na przestrzeni do nawet kilkudziesięciu lat, jeżeli posiadamy dostęp

[18] Wydajność komputerów mierzymy w ilości operacji zmiennoprzecinkowych, którą mogą wykonać w ciągu sekundy i definiuje się ją jako ***FL****oating point* ***O****perations* ***P****er* ***S****econd*.

przez 24 godziny do kilkuset procesorów.

Często nie zdajemy sobie sprawy z tego jak istotne dla społeczeństwa i rozwoju cywilizacji są trudne problemy, z którymi zmagają się dzień i noc komputery zaprogramowane przez programistów. Praktycznie wszystkie dostępne nowoczesne leki zostały opracowane dzięki analizie komputerowej [78] [28] [115]. Za pomocą medycyny i technik obliczeniowych walczymy z rakiem [232] i chorobami zakaźnymi *(m. in. HIV)* [114] [15]. To dzięki nim zarządzamy złożami ropy naftowej i gazu [154] oraz symulujemy zderzenia pojazdów z przeszkodami, aby uczyć się jak konstruować bezpieczniejsze samochody [192]. Dzisiaj nie ma potrzeby stosowania bardzo kosztownych tuneli aerodynamicznych do testów wykonywanych na samochodach lub samolotach - to samo można zrobić symulując dany proces [191]. Pół wieku temu można było symulować jedynie rozkład ciśnienia wokół jednego skrzydła statku powietrznego. W XXI wieku symulacja obejmująca cały płatowiec samolotu nie sprawia problemów [269] [189]. Superkomputery znajdują także zastosowanie przy Wielkim Zderzaczu Hadronów, gdzie analizują przesyłane z detektorów olbrzymie ilości danych [231].

Z punktu widzenia tematu tej pracy oczywiście były, są i będą prowadzone symulacje algorytmów kwantowych. Jedne z ciekawszych wyników można zobaczyć w [136], gdzie testami objęto symulacje wykonywane równolegle na 4096 procesorach i wykorzystujące 1 TB pamięci operacyjnej. Inną ciekawą publikacją jest tekst [134], gdzie prezentowane są symulacje na komputerze równoległym Sun Enterprise 4500.

Maszyny obliczeniowe są stosowane do przewidywania katastrof przyrodniczych takich jak tornada [281], fale tsunami [174] i trzęsienia ziemi [67]. Superkomputery są bardzo drogim sprzętem, którego koszty często przekraczają miliony dolarów. Stanowią jednak bardzo ważny filar wysoko rozwiniętego społeczeństwa co zaznaczono m. in. w raporcie zespołu doradców prezydenta USA [40].

Intensywny rozwój technik równoległych jest spowodowany ograniczeniami technologicznymi. Problem ten został poruszony w rozdziale 2.2. W wyniku rozwoju układów zwiększa się ich stopień scalenia. W związku ze wzrostem ilości tranzystorów zwiększa się sumaryczne natężenie prądów między nimi płynących przez co negatywnej zmianie ulega także ilość wydzielanej energii w postaci ciepła. Także opóźnienia sygnałów, które są transmitowane wzdłuż ścieżek układu ograniczają do pewnego progu wzrost częstotliwości taktowania. Z tych powodów wszystkie procesory produkowane w ciągu ostatnich lat posiadają kilka rdzeni z których każdy jest przystosowany do realizowania innych strumieni instrukcji na różnych zbiorach danych. Tego typu rozwiązania są stosowane w różnego typu urządzeniach od konsol do gier, przez nowoczesne telefony komórkowe do zwykłych komputerów i serwerów. Są także podstawą superkomputerów. Można więc się spodziewać, że zastosowanie przetwarzania równoległego będzie coraz większe.

Problematyka wykorzystania potężnych zasobów obliczeniowych budzi czasem dużo wątpliwości. Tylko *33%* mocy obliczeniowej 500 najszybszych komputerów jest przeznaczona do celów czysto badawczych [259]. Według oficjalnych obliczeń niewiele ponad *dwie* operacje na sto przypada dla instalacji o charakterze obronnym, a aż *49%* nie ma jasno określonego celu. W czołówce najbardziej rozwiniętych państw pod względem możliwości obliczeniowych znajdują się Stany Zjednoczone, Japonia i Chiny [259].

Kolejne podrozdziały prezentują różne kategorie komputerów równoległych. Odnajdziemy w nich skrótową informację o tym z wykorzystaniem jakich bibliotek i środowisk programistycznych można wykorzystać potencjał systemów równoległych. Informacje te są jednak tylko symboliczne i chcąc zgłębić temat należy zapoznać się z literaturą. Szczególną uwagę należy zwrócić na publikację Michaela Flynna z 1972 roku dotyczącą taksonomii architektur komputerów [186]. Inne prace opisujące organizację maszyn obliczeniowych przedstawili m. in. Schwartz [117], Feng [261] oraz Skillicorn [55] i Handler [270].

6.1. Definicja podstawowych pojęć

Model zaprezentowany w tym podrozdziale będzie tylko dużym uogólnieniem zagadnień z punktu widzenia programisty[19], który często nie wie co naprawdę dzieje się w procesorze, a jedynie widzi ostateczne wyniki i może zmierzyć czas realizacji zadania.

6.1.1. Procesy, wykonywanie współbieżne, równoległe i rozproszone, technika przeplotu

Program sekwencyjny jest najprostszym z możliwych opisem w jaki sposób przekształcić dane wejściowe, aby otrzymać określony wynik. Przyjmuje się, że taki program jest wykonywany przez jeden procesor choć w rzeczywistości może w systemie być realizowany przez kilka procesorów. W istocie jednak nawet program sekwencyjny nie jest przetwarzany jako ciąg instrukcji *jedna po drugiej*, gdyż współczesne procesory to układy *superskalarne* [142] i *potokowe* [222]. Sam kompilator generując kod dla danego układu może wykorzystać m. in. wektoryzację dzięki, której niezależne instrukcje lub ich bloki będą wykonywane równolegle przez różne systemy funkcyjne jednego rdzenia mikroprocesora [208]. Także instrukcje multimedialne *(np. SSE, AVX)* umożliwiają wykonanie wielu instrukcji w czasie jednego cyklu.

W momencie, gdy do rozwiązania jakiegoś problemu chcemy użyć więcej pro-

[19]Nie dotyczy to programisty języka asembler, gdyż ten może znacznie więcej jednak niezwykle trudno byloby pisać programy równolegle w samym asemblerze.

cesorów niż jeden jesteśmy zmuszeni do podzielenia interesującego nas zagadnienia na wiele mniejszych podproblemów. Podproblemy te powinniśmy móc rozwiązywać równolegle, a wyniki ich obliczeń po odpowiednim złożeniu dają nam końcowy rezultat.

Należy jedak wiedzieć, że nie każdy program można zrównoleglić. W szczególności nie można zrównoleglić wykonywania zadań, gdy końcowy wynik jednego z zadań jest daną wejściową zadania kolejnego[20]. Nie można więc w żaden sposób zrównoleglić zadania przedstawionego na Listingu 10.

```
1  //...
2  double tab[N];
3  tab[0] = 1.5;
4
5  int i;
6
7  for(i = 0; i < N-1; i++)
8  {
9      tab[i+1] = sqrt(tab[i]) + pow(tab[i], i);
10 }
11 //...
```

Listing 10. Sekwencyjny algorytm obliczania elementów tablicy na podstawie wartości we wcześniejszym indeksie.

Systemy operacyjne, z którymi się spotykamy posiadają wiele cech - są one m. in. wielozadaniowe. Własność ta umożliwia nam równoczesne wykonywanie więcej niż jednego procesu[21]. Cecha ta jest szczególnie interesująca, jeżeli dysponujemy systemem z jednym procesorem. Gdy jesteśmy zalogowani w tego typu systemie możemy uruchomić w nim wiele programów *(inne usługi systemowe muszą także działać w tle)*. Nie jesteśmy zmuszeni do oczekiwania na zakończenie jednego zadania, aby wykonać drugie mimo, że fizycznie posiadamy tylko jedną jednostkę obliczeniową. Czasy realizacji procesów są dzielone na mniejsze kwanty czasu, które są wykonywane przez procesor potrafiący analizować w danym momencie tylko jeden z nich. W momencie wykonania kwantu obliczeń dochodzi do tzw. przeplotu *(ang. interleave)* [123] i zmiany kontekstu *(ang. context switching)* [198]. Za szeregowanie zadań w kolejce odpowiada planista systemowy, a do przełączania dochodzi tak szybko,

[20] Nawet jeżeli rozpatrujemy program sekwencyjny i dwa podproblemy, w którym pierwszy zwraca wynik typu prawda (1) lub fałsz (0) wymagany do rozpoczęcia działania drugiego to zdania te nie muszą być faktycznie zrealizowane sekwencyjnie. W przypadku posiadania wolnych zasobów mikroprocesor może wykonać niezależnie i równolegle operacje zarówno dla 0, jak i dla 1, a po zakończeniu obliczeń wymaganego podzadania jedynie wykorzystać odpowiednie dane. Nawet w przypadku, gdy nie ma wystarczających zasobów do wykonania obliczeń dla dwóch wariantów procesor może próbować *wstrzelić się* w jedną z możliwości.

[21] Proces jest często definiowany jako jeden egzemplarz wykonywanego programu. W wielu systemach operacyjnych jest on w sposób jednoznaczny identyfikowany za pomocą numeru PID *(ang. process identifier)*.

że użytkownik odnosi wrażenie wykonywania równoległego programów[22]. Tego typu rozwiązanie można jednak nazwać jedynie pseudorównoległym.

W literaturze często w odniesieniu do tego typu realizacji można spotkać się z ujęciem jego jako *przetwarzania współbieżnego*. Tryb ten jednak nie jest efektywny ze względu na konieczność wcześniej wspomnianego przełączania kontekstu. Polega ono na zapamiętaniu wszelkich niezbędnych danych określających aktualny stan procesu *(m. in. wartości przechowywane w rejestrach)* tak, aby możliwe było wznowienie obliczeń dokładnie w momencie, gdy je przerwano. Czynność ta zajmuje dużo czasu i jest szczególnie skomplikowana dla procesorów superskalarnych [179] dlatego w praktycznie wszystkich współczesnych systemach operacyjnych jest ona wykonywana w lekko rozwiniętej odmianie uwzględniającej podział czasu *(ang. time-sharing)*. Jednostki obliczeniowe są przydzielone do procesów na pewien odcinek czasu *(zwykle o długości nie przekraczającej 1 ms)* [59]. Rola planisty systemowego jest bardzo istotna i odnosi się nie tylko do wydajności samej w sobie a także m.in. oszczędności energii w układach mobilnych [173] lub różnych rozwinięć sprzętowych takich jak Hyper-Threading Technology [151]. Zarówno problem planisty systemowego, jak i technologii HT został poruszony w tej pracy *(głównie w kontekście analizy efektywności algorytmów kwantowych na różnych układach)*. Sposoby przetwarzania można zdefiniować w następujący sposób:

> *Dwa procesy są współbieżne, gdy jeden z nich rozpoczyna się przed zakończeniem drugiego.*
> *Dwa procesy są równoległe, jeżeli jeden z nich rozpoczyna się przed zakończeniem drugiego, a także realizowane są na różnych jednostkach obliczeniowych.*

W przypadku przetwarzania współbieżnego nie ma informacji o systemie na jakim są realizowane obliczenia więc można przyjąć, że dotyczy to systemu jednoprocesorowego i zjawiska przeplotu. Tego typu realizacja została przedstawiona na Rys. 6.1.

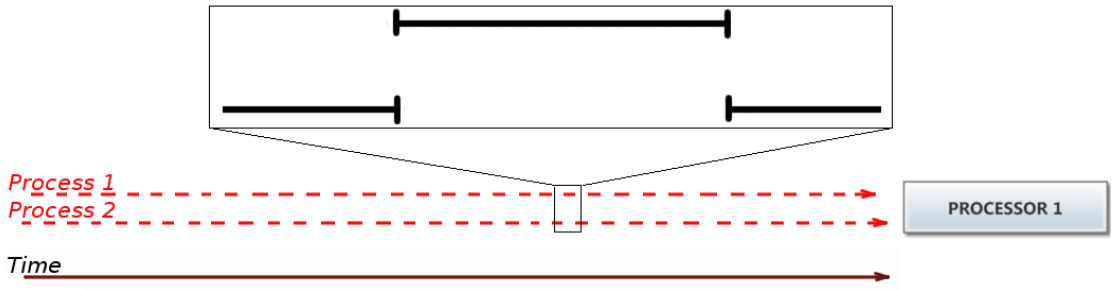
Klasą najbardziej nas interesującą jest przetwarzanie równoległe. Zostało ono zaprezentowane graficznie na Rys. 6.2.

Wykonywanie procesów może przebiegać także w sposób całkowicie rozproszony *(obliczenia rozproszone - ang. distributed computing)*.

Definicja:

> *Obliczenia rozproszone realizują procesy wykonywane całkowicie odrębnie*

[22] Wersji systemów operacyjnych jak i rodzajów planistów jest wiele. Odrębną kategorią są systemy czasu rzeczywistego. Gwarantują one określone założenia odnośnie czasu realizacji określonego zadania, od których w przypadku niektórych zastosowań może być uzależnione ludzie życie *(np. aparatura medyczna podtrzymująca funkcje życiowe)*.



Rys. 6.1. Schemat przedstawiający dwa procesy przetwarzane w sposób współbieżny. *Źródło: Wykonanie własne.*

Rys. 6.2. Schemat przedstawiający dwa procesy przetwarzane w sposób równoległy. *Źródło: Wykonanie własne.*

> *przez rozproszone jednostki obliczeniowe (często rozproszone pod względem swojej lokalizacji geograficznej) i połączone za pomocą kanałów komunikacyjnych.*

Projektowanie, implementacja oraz testowanie programów równoległych są trudniejszymi zadaniami od pracy przy ich sekwencyjnych odpowiednikach. Podczas programowania równoległego możemy spotkać się z dodatkowymi problemami, które nie występują przy programowaniu systemów sekwencyjnych. Problemy te dotyczą głównie koordynacji pracy pomiędzy procesorami realizującymi różne zadania. Zagadnieniami najbardziej klasycznymi w obszarze problemów programowania równoległego jest problem wzajemnego wykluczania [219] i ucztujących filozofów [50]. W podręczniku akademickim *'Wprowadzenie do obliczeń równoległych'* [46] dość dokładnie pisze o nich Zbigniew Czech.

6.1.2. Ocena efektywności algorytmów równoległych

Aby móc porównywać między sobą efektywność działania algorytmów równoległych należy znać kryteria ich oceny. Podczas badania algorytmów sekwencyjnych opieramy się głównie na zliczaniu czasu działania lub wykorzystaniu pamięci. Można także oceniać rozwiązanie pod względem łatwości implementacji. Programowanie równoległe pociąga jednak za sobą dodatkowy czynnik, którym jest liczba procesorów jaką posiada maszyna na której został uruchomiony algorytm.

Czyhają także inne zmienne mogące wpłynąć na ocenę efektywności takie jak

przenośność lub stopień wykorzystania komunikacji. Obydwa te kryteria są bardzo istotne. Architektur komputerów równoległych jest wiele, a być może będziemy chcieli uruchomić napisany program na kilku z nich. Z drugiej strony problemy w komunikacji między szybkimi procesorami za pomocą magistral danych potrafią w niektórych przypadkach niesamowicie spowolnić wykonywanie.

Dla komputerów równoległych najczęściej głównym czynnikiem jaki wykorzystujemy do porównywania algorytmów jest przyśpieszenie uzyskane dzięki uruchamianiu algorytmu na określonej ilości procesorów w odniesieniu do algorytmu wykonywanego sekwencyjnie na jednej jednostce obliczeniowej. Przyśpieszenie można policzyć ze wzoru:

$$Speedup(p) = \frac{t(p)}{t(1)} \tag{6.1}$$

gdzie *t(1)* określa złożoność rozwiązania sekwencyjnego, a *t(p)* odnosi się do algorytmu wykonywanego równolegle na p procesorach. Podczas badań empirycznych można jako *t(p)* rozumieć także faktyczny czas realizacji danego algorytmu na p procesorach. Uzyskamy wtedy bardziej rzeczywiste wyniki, w których zostaną uwzględnione różne dodatkowe straty *(wynikające m. in. z konieczności synchronizacji i komunikacji)*.

Przyśpieszenie ma wartość progową. Wykorzystując p procesorów nie można uzyskać przyśpieszenia wyższego niż p.

$$Speedup(p) \leq p \tag{6.2}$$

Tak zdefiniowane przyśpieszenie jest doskonałą miarą korzyści, ponieważ oddaje uzyskany zysk czasowy względem algorytmu sekwencyjnego. Niestety uzyskiwane wyniki często odbiegają od ideału.

6.2. Ograniczenia programowania równoległego

Obliczenia równoległe nie są doskonałym rozwiązaniem wszystkich problemów obliczeniowych. Bez względu na rozwój technologii i procesorów można dostrzec bariery, które ciężko pokonać.

6.2.1. Prawo Amdahla

W prawie każdym programie możemy wyróżnić bloki instrukcji, które mogą być wykonywane równolegle i te które muszą być realizowane w sposób sekwencyjny. Całkowite zrównoleglenie jest mało kiedy możliwe - musimy najczęściej chociażby wczytać dane wejściowe i jakoś zainicjować odpowiednie tablice, na których będziemy

operować. Jeżeli jednak program ma taki właśnie charakter i nie możemy pozbyć się całkowicie podejścia sekwencyjnego, to maksymalne przyśpieszenie jest zbieżne do jakiejś wartości.

Przyjmijmy, że s określa jaką część kodu musimy wykonywać sekwencyjnie. Jeżeli uznamy, że *1* będzie oznaczać całość to przez *1-s* możemy rozumieć kod, który zdołamy zrównoleglić. Uzyskane przyśpieszenie $Speedup_A(p)$[23] możemy obliczyć ze wzoru:

$$Speedup_A(p) = \frac{1}{s + (1-s)/p} \tag{6.3}$$

Wyniki dla $s \in \{0.05; 0.10; 0.25\}$ zaprezentowano w Tab. 6.1.

Tab. 6.1. Problem maksymalnego przyspieszenia uzyskiwanego według twierdzenia Amdahla. *Źródło: Wykonanie własne.*

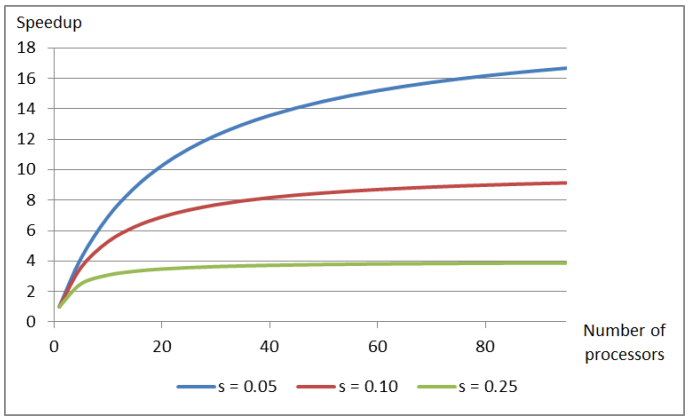
	Speedup(p)		
p	s = 0.05	s = 0.10	s = 0.25
1	1.0000	1.0000	1.0000
5	4.1666	3.5714	2.5000
10	6.8965	5.2631	3.0769
15	8.8235	6.2500	3.3333
20	10.2564	6.8965	3.4782
25	11.3636	7.3529	3.5714
30	12.2448	7.6923	3.6363
35	12.9629	7.9545	3.6842
40	13.5593	8.1632	3.7209
45	14.0625	8.3333	3.7500
50	14.4927	8.4745	3.7735
		...	
95	16.6666	9.1346	3.8775

Prezentację graficzną problemu można zobaczyć na Rys. 6.3.

Łatwo zauważyć, że dla $s = 0.25$ barierą jest przyśpieszenie wynoszące *4*. Po dodaniu do *45* procesorów dodatkowych pięciu przyśpieszenie wzrasta zaledwie o ok. *0.02*.

Prawo, które zostało tutaj ukazane nazywa się prawem Amdahla. Jest ono dokładniej omówione w pierwotnej publikacji Amdahla [87] oraz książkach Quinna [188] [187] i Goodmana [97]. Nie należy zbytnio przejmować się ograniczeniami tego podejścia do problemu. Według tej koncepcji nie warto budować komputerów po-

[23] A - od nazwiska twórcy Amdahla; p - liczba procesorów.



Rys. 6.3. Problem granicy przyspieszenia uzyskiwanego według twierdzenia Amdahla. *Źródło: Wykonanie własne.*

siadających olbrzymie ilości jednostek obliczeniowych co oczywiście jest błędnym wnioskiem.

6.2.2. Prawo Gustafsona

Do problemu ograniczenia wydajności w przypadku komputerów równoległych możemy jednak podejść w inny sposób. Przyjmijmy, że działanie programu podzielimy na dwa etapy *a (sekwencyjne)* i *b (równoległe)* podobnie jak w przypadku prawa Amdahla. Jeżeli więc będziemy rozpatrywać czas działania algorytmu sekwencyjnego względem wersji równoległej to musi on wynieść:

$$a + p \cdot b \tag{6.4}$$

Przyspieszenie dzięki przetwarzaniu równoległemu wyniesie więc:

$$Speedup_G(p)^{24} = \frac{a + p \cdot b}{a + b} \tag{6.5}$$

Po przekształceniach uzyskamy:

$$Speedup_G(p) = p - \frac{a}{a + b}(p - 1) \tag{6.6}$$

$\frac{a}{a+b}$ jest ułamkiem określającym czas wykonywania obliczeń sekwencyjnych na komputerze równoległym. Możemy dla uproszczenia przyjąć $\alpha = \frac{a}{a+b}$.

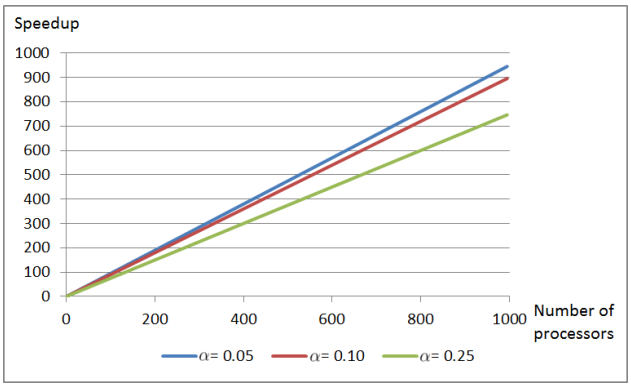
$$Speedup_G(p) = p - \alpha(p - 1) \tag{6.7}$$

Gdy α ma małą wartość funkcja przyspieszenia wraz ze wzrostem ilości procesorów p dąży do wartości idealnej. Tego typu podejście zostało zaprezentowano przez

[24] G od nazwiska Gustafson.

Gustafsona w [129] [103].

Graficzną prezentację krzywej Gustafsona zaprezentowano na Rys. 6.4, a przykładowe dane obrazujące problem zastały przedstawione w Tabeli 6.2.



Rys. 6.4. Prezentacja graficzna krzywej Gustafsona. *Źródło: Wykonanie własne.*

Tab. 6.2. Problem maksymalnego przyspieszenia uzyskiwanego według prawa Gustafsona. *Źródło: Wykonanie własne.*

	Speedup(p)		
p	$\alpha = 0.05$	$\alpha = 0.10$	$\alpha = 0.25$
1	1.00	1.00	1.00
5	4.80	4.60	4.00
10	9.55	9.10	7.75
15	14.30	13.60	11.50
20	19.05	18.10	15.25
25	23.80	22.60	19.00
30	28.55	27.10	22.75
35	33.30	31.60	26.50
40	38.05	36.10	30.25
45	42.80	40.60	34.00
50	47.55	45.10	37.75
...			
95	90.30	85.60	71.50
...			
995	945.30	895.60	746.50

6.2.3. Empiryczne wyznaczanie części sekwencyjnej algorytmu - Miara Karpa-Flatta

Obydwa przedstawione wcześniej modele są jednak wyidealizowane i nie oddają dokładnie problemu ograniczenia programowania równoległego. W celu uzyskania dokładniejszej analizy musimy ustalić $t_s(1)$ *(część wykonywana sekwencyjnie)*, $t_p(p)$ *(część wykonywana równolegle)* oraz $t_h(p)$ *(obliczenia dodatkowe wynikające z prowadzenia obliczeń równoległych)*. Jeżeli więc przyjmiemy za $t(p)$ czas wykonywania algorytmu równoległego to zachodzi równość:

$$t(p) = t_s(1) + t_h(p) + \frac{t_p(p)}{p} \tag{6.8}$$

Możemy oznaczyć jako e i obliczyć współczynnik obliczeń sekwencyjnych i obliczeń wynikających z realizacji równoległej do czasu całego realizacji w następujący sposób:

$$e = \frac{t_s(1) + t_h(p)}{t(1)} \tag{6.9}$$

W istocie zachodzi równość:

$$t_p(p) = t(1) - (t_s(1) + t_h(p)) \tag{6.10}$$

dzięki czemu wzór 6.8 możemy przekształcić

$$\begin{aligned} t(p) = t_s(1) + t_h(p) + \frac{t(1) - (t_s(1) + t_h(p)}{p} = \\ = e \cdot t(1) + \frac{(1-e) \cdot t(1)}{p} \end{aligned} \tag{6.11}$$

Dzieląc wynik przez $t(p)$ uzyskamy odwrotność przyspieszenia:

$$\frac{1}{Speedup(p)} = e + \frac{(1-e)}{p} \tag{6.12}$$

Z czego wynika, że e można policzyć ze wzoru:

$$e = \frac{\frac{1}{Speedup(p)} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{6.13}$$

Po wykonaniu pomiaru znamy wartości wszystkich zmiennych *(uzyskane przyspieszenie i ilość procesorów)* wymaganych do obliczenia e. Możemy zatem policzyć dokładnie jaka część programu nie została zrównoleglona. Wartość e jest zależna od wielu czynników takich jak architektura mikroprocesora lub zastosowana pamięć operacyjna. Opisany model został zaproponowany w pracy Karpa i Flatta [143].

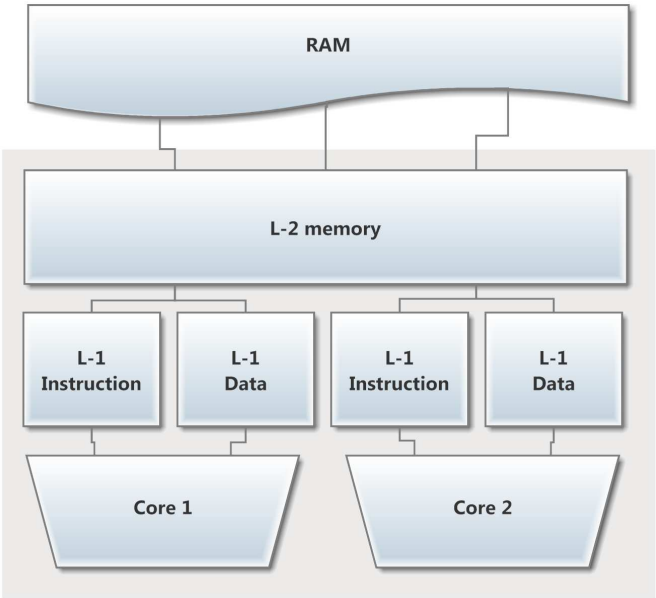
Inne prace dotyczące miary efektywności algorytmów równoległych zaproponowali także Carmony i Rice [29] oraz Van-Catledge [81].

6.3. Komputery równoległe z pamięcią wspólną

W tym rozdziale zostanie zaprezentowana tematyka komputerów równoległych z pamięcią wspólną oraz pamięci DSM.

6.3.1. Komputery wieloprocesorowe

Obecnie większość procesorów posiada kilka rdzeni. Przez rdzeń rozumiemy autonomiczną część procesora potrafiącą przetwarzać dane i przechowywać dla własnych potrzeb informacje[25]. Na Rys. 6.5. przedstawiono ideowy schemat blokowy typowego procesora dwurdzeniowego.



Rys. 6.5. Schemat ideowy klasycznego procesora dwurdzeniowego. *Źródło: Wykonanie własne.*

Procesor posiada dostęp do pamięci podręcznej kilku poziomów *(w zależności od typu może być to nawet pamięć trójstopniowa)*. Najczęściej pamięć pierwszego poziomu jest podzielona na moduły do przechowywania instrukcji i danych, do których układ ma najszybszy dostęp. Odwoływanie się do kolejnych poziomów jest wolniejsze *(jednak i tak bardzo szybkie)*. Pamięc L-1 jest najmniejsza *(głównie ze względu na wysokie koszty i małe możliwości integracji)* i znajduje się najbliżej jądra procesora.

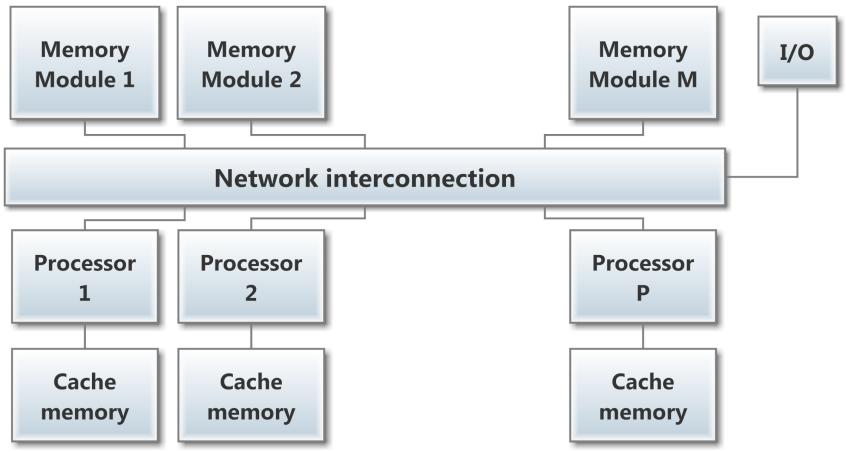
[25]Dokładna definicja rdzenia może być różna w zależności od konstrukcji urządzenia, w którym mogą występować np. rdzenie o specjalistycznym lub ogólnym przeznaczeniu.

Kolejne segmenty pamięci podręcznej są większe, wolniejsze i zazwyczaj spełniają rolę bufora pomiędzy procesorem, a charakteryzującą się większym czasem dostępu pamięcią RAM.

Rozważając komputery z pamięcią wspólną nie ograniczamy się jednak tylko do maszyn zbudowanych z jednego procesora wielordzeniowego. Do rozwiązań tej klasy należą także komputery posiadające wiele procesorów *(także wielordzeniowych)*. Głównym warunkiem przynależności do tej grupy pozostaje przestrzeń adresowa o jednolitym dostępie *(ang. uniform memory access, UMA)* [51].

Zazwyczaj tego typu komputery opierają się na architekturze MIMD[26] *(ang. muliple instruction, multiple data)*. Każdy układ przetwarzający dane jest wyposażony we własny zegar, własną jednostkę sterującą i wszystkie rejestry. Instrukcje mogą być wykonywane w sposób całkowicie asynchroniczny na różnych zbiorach danych przez różne strumienie instrukcji.

Schemat układu wieloprocesorowego MIMD z pamięcią wspólną przedstawiono na Rys 6.6.



Rys. 6.6. Struktura komputera o architekturze MIMD posiadającego wiele procesorów. *Źródło: Wykonanie własne.*

Programowanie tego typu komputerów jest najprostsze i zazwyczaj pierwszeństwo tutaj wiodą programy wielowątkowe[27], które mają w niektórych przypadkach znaczną przewagę nad procesami - krótszy czas tworzenia i naturalne współdzielenie pamięci z innymi wątkami działającymi w ramach danego procesu.

Często wykorzystywanym interfejsem do pisania programów wielowątkowych jest OpenMP. Standard ten został bardzo dokładnie opisany w książkach Chapmana [19] i Chandra [31]. Programista pisząc oprogramowanie w OpenMP wykorzystuje głównie specjalne dyrektywy *(tzw. pragmy)* zapisywane jako komentarze, które w

[26] Istnieją odstępstwa, ale tylko w bardzo specyficznych przypadkach i dla klasycznych procesorów marki Intel lub AMD to założenie jest prawdą.

[27] Wątek jest instancją wykonawczą procesu.

odpowiedni sposób interpretuje kompilator. W ten sposób można prosto zdefiniować, które części kodu mają zostać zrównoleglone.

Prosty przykład napisany z wykorzystaniem OpenMP został zaprezentowany jak na Listingu 11.

```
1  int main(int argc, char **argv)
2  {
3      const int N = 1000000000;
4      long long int i;
5      long long int table[N];
6
7      #pragma omp parallel for num_threads(2)
8      for (i = 0; i < N; i++)
9          table[i] = 3 * sqrt(i);
10
11     return 0;
12 }
```

Listing 11. Zrównoleglenie pętli za pomocą środowiska OpenMP.

W kodzie tym za pomocą pragmy *omp parallel for num_threads(2)* wykonano zrównoleglenie pętli. Do zrównoleglenia dochodzi z wykorzystaniem dwóch wątków, co jest zdefiniowane za pomocą *num_threads*. W przypadku, gdy tego typu program zostanie uruchomiony na systemie z jednym procesorem wątki będą realizowane wcześniej omówioną metodą przeplotu. Oczywiście OpenMP posiada wiele różnego rodzaju dyrektyw służących m. in. do synchronizacji i operacji redukcji danych.

Większość współczesnych procesorów działa w trybie SMP *(ang. symmetric multiprocessor)*, które charakteryzują się symetrycznym dostępem do wszystkich obszarów pamięci. Każdy procesor ma szybki dostęp do całej pamięci RAM. Gdy jednak liczba procesorów w systemie rośnie konstrukcja odpowiednio szybkich magistral danych staje się bardzo trudna. W większości rozwiązań wydajność teoretyczna będąca sumą wydajności wszystkich procesorów może wzrastać liniowo przez dokładanie kolejnych układów obliczeniowych, lecz przepustowość pamięci wcale nie określa trend liniowy.

Aby rozwiązać ten problem wprowadzono architekturę NUMA *(ang. nonuniform memory accress)* [124]. W jej przypadku pamięć dostępna dla procesora dzieli się na obszary, którym są przyporządkowane czasy dostępu do nich podczas odczytu lub zapisu danych przez określoną jednostkę. Dzięki tego typu hierarchicznej pamięci procesor może w pewnym stopniu sam decydować o wykorzystaniu w danym momencie najoptymalniejszych obszarów przestrzeni.

Rozwinięciem koncepcji NUMA jest ccNUMA *(ang. Cache Coherent Non-Uniform Memory Access)* [146], która zapewnia spójność danych nie tylko dla RAM, ale także dla pamięci podręcznej procesorów. Technika ta jest realizowana z wykorzystaniem dodatkowego sprzętu i specjalnego protokołu. ccNUMA została zaimplementowana m. in. w SGI Altix 4000 *(maksymalnie 1024 procesory)* [264], SGI Origin *(do 512*

procesorów) [122], [45] oraz HP Superdome *(maksymalnie 128 procesorów)* [88].

Jedną książek jakie napisano o OpenMP jest *'Using OpenMP: Portable Shared Memory Parallel Programming'* [19]. Ukazało się dużo innych pozycji prezentujących możliwości programowania z wykorzystaniem tego interfejsu *(np. [31])*. Model systemu z pamięcią wspólną często nazywany jest PRAM *(ang. Parallel Random Access Machine)* [246].

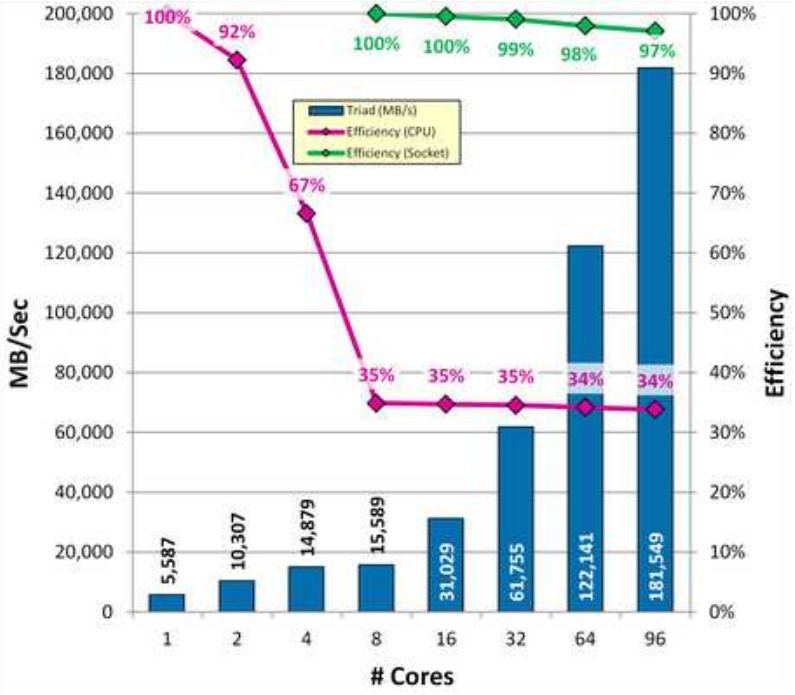
6.3.2. Architektura pamięci DSM

Konstrukcje z pamięcią wspólną są w znacznym stopniu ograniczone. Istnieją jednak technologie wprowadzające na drodze sprzętowej lub programowej pamięć, która mimo iż fizycznie jest rozproszona pod względem logicznym jest wspólna. Programista dzięki temu może programować komputer z pamięcią rozproszoną tak jak zwykły komputer z pamięcią wspólną. Tego typu technika nosi miano DSM *(ang. distributed shared memory)* [214], [197].

Koncepcja systemu wykorzystującego DSM jest prosta. Każdy procesor ma dostęp do całej pamięci z wykorzystaniem jednolitej przestrzeni adresowej. Lokalnym komórkom pamięci są przyporządkowane adresy wirtualne. W momencie, gdy jednostka obliczeniowa zgłasza zapotrzebowanie na określone dane i okazuje się, że nie posiada ich we własnych zasobach dochodzi do translacji adresu wirtualnego na fizyczny i przesyłu odpowiednich informacji za pomocą odpowiednich łącz komunikacyjnych. Programista może nawet nie być świadom tego, że dane są w odpowiednim momencie przesyłane pomiędzy procesorami - proces ten jest dla niego *przezroczysty*. Odwoływanie się do danych za pomocą wolniejszych łącz jest jednak związane ze znacznym wzrostem czasu dostępu do pamięci i spadkiem efektywności obliczeń.

Rozwiązania tego typu nie są nowością. Zaproponował je Li w [135] oraz [161]. W 1993 roku jedno z pierwszych rozwiązań sprzętowych zastosowało Kendall Square Research w komputerze KSR1 [86].

Warto również wspomnieć o rozwiązaniach ScaleMP vSMP *(virtual symmetric multiprocessing system)* [64]. Umożliwiają one wykorzystywanie jednolitej przestrzeni adresowej na różnego typu komputerach. Problem skalowalności pamięci z zastosowaniem ScaleMP vSMP i platformy opartej na węzłach zbudowanych z czterech sześciordzeniowych procesorów Nehalem EX [190] przedstawiono na Rys. 6.7.



Rys. 6.7. Aspekt przepustowości pamięci na platformie vSMP opartej na Nehalem EX. *Źródło: ScaleMP.*

Już podczas aktywnego wykorzystywania 8 rdzeni przepustowość pamięci przypadająca na jeden z nich spada do *35%*.

6.4. Komputery równoległe z pamięcią rozproszoną

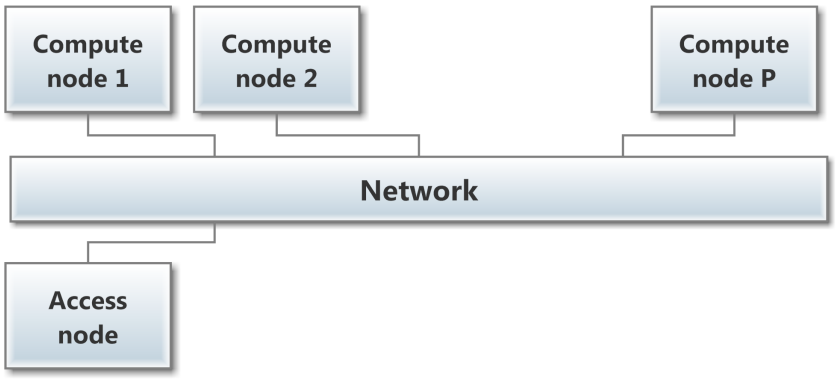
Ten rozdział jest poświęcony klastrom komputerowym i wykorzystywanym w nim topologiom sieci.

6.4.1. Klastry komputerowe

Rozwiązaniem umożliwiającym dalsze zwiększanie mocy obliczeniowej jest zwielokrotnienie ilości maszyn wykonujących obliczenia *(węzłów - ang. nodes)* i połączenie ich za pomocą możliwie szybkiej sieci komputerowej. Węzły mogą pracować w pewnym stopniu niezależnie i wykorzystywać sieć do wymiany danych. Komunikacja w ramach jednego węzła może być nawet 1000 razy szybsza od komunikacji między węzłami.

Aktualne rozwiązania tego typu mają od kilkudziesięciu do kilku milionów procesorów. Według rankingu TOP500 z czerwca 2012 roku aż 407 z 500 najszybszych na świecie superkomputerów to klastry komputerowe [259]. Swoją popularność zawdzięczają one głównie prostocie budowy olbrzymich systemów i co za tym idzie ciągłym wzrostem wydajności. Znacznym ograniczeniem jest jednak sieć za pomocą, której są

przesyłane dane między węzłami. Wszystkie maszyny z listy TOP500 to wysoko wyspecjalizowane rozwiązania. Krąg odbiorców tego typu systemów jest bardzo mały *(tylko najlepsze laboratoria naukowe, wojsko i organizacje rządowe)*, a dedykowany sprzęt na małą skalę jest w stanie produkować niewielka ilość producentów. Klaster komputerowy można jednak zbudować nie tylko ze specjalistycznego sprzętu, ale także z komputerów osobistych klasy PC. System taki po raz pierwszy uruchomiono w NASA Goddard Space Fight Center w 1994 roku i nazwano Beowulf [243], [66]. O możliwości budowy klastrów z domowych komputerów kilka lat później napisano książkę [257]. Schemat klasycznej maszyny tego typu przedstawiono na Rys. 6.8.



Rys. 6.8. Model typowego klastra komputerowego z jednym węzłem dostępowym. *Źródło: Wykonanie własne.*

Komputery będące klastrami wydajnościowymi to np. BlueGene/L [6], BlueGene/P [227] i ostatnia wersja Q [27].

Maszyny z pamięcią rozproszoną programuje się z wykorzystaniem specjalnych bibliotek, które mają zaimplementowane odpowiednie funkcje do przesyłania danych między różnymi procesorami. Aktualnie uznawanym na całym świecie standardem jest MPI *(ang. Message Passing Interface)*. W latach 80. przed powstaniem MPI istniały takie standardy jak PICL [90], Argonne's P4 [217] i PVM [249]. Ohio Superkomputer Center opracowało także bibliotekę LAM *(później nazywana jako TCGMSG)* do obliczeń chemii kwantowej [105]. Własne rozwiązanie w programowaniu komputerów rozproszonych ISIS zaproponował także Cornell University [147]. Wszystkie te rozwiązania jednak były stopniowo wypierane przez MPI, którego pierwsza wersja była gotowa w 1994 roku. Istnieje kilka odmian MPI *(także przenośnie [100])*. Najpopularniejsze wersje to MPI-1 oraz bardziej rozwinięta MPI-2 *(wprowadzona w 1998 roku)* [7]. Można spotkać także wiele specyficznych modyfikacji zoptymalizowanych pod konkretne komputery. MPI-2 zawiera m.in. rozwinięcia dotyczące operacji I/O i dynamicznego zarządzania procesami. Wiele elementów MPI-2 została przeniesiona z MPI-IO, który był tworzony na potrzeby NASA [216].

Podobnie jak w przypadku początków w zwykłym programowaniu tak i na klastrze komputerowym pierwszym programem jaki możemy napisać i uruchomić jest

'Hello world!'. Oczywiście tak prosty program jedynie wywoła funkcję *printf* na wielu węzłach naszego klastra przez co nie zobaczymy wszystkich wyników na maszynie dostępowej. Przykładowy kod znajduje się w Listingu 12.

```
1 # include <mpi.h>
2 # include <stdio.h>
3
4 int main(int argc, char **argv)
5 {
6     int my_rank;
7     int size;
8
9     MPI_Init(&argc, &argv);
10
11    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &size);
13
14    printf("Hello world\n");
15    printf("I'm processor %d\n", my_rank);
16    printf("All processors %d\n", size);
17    MPI_Finalize();
18 }
```

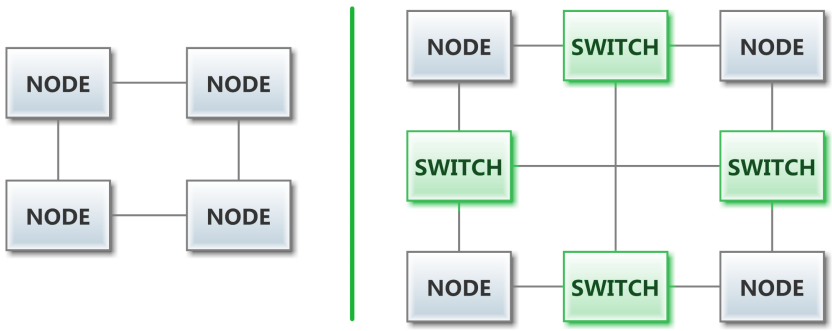
Listing 12. *'Hello World'* w wersji na klastry komputerowe.

MPI jest bardzo zaawansowanym standardem posiadającym około 500 funkcji służących m.in. do komunikacji, przesyłania danych i synchronizowania procesów. Operuje także w odróżnieniu od OpenMP nie na wątkach lecz na procesach. Pomocne z zapoznaniem się z tym standardem mogą być książki Josepha Sloana [99], Petera Pacheco [207]. Zarówno o MPI jak i OpenMP pisał też Quinn [116].

6.4.2. Topologie połączeń międzywęzłowych

Jak już wcześniej wspomniałem w przypadku komputerów z pamięcią rozproszoną kluczowym aspektem jest sieć zastosowana do komunikacji pomiędzy komputerami wchodzącymi w skład systemu. Sieci można podzielić na dwa typy: statyczne i dynamiczne. W przypadku tych pierwszych łącza są stałymi połączeniami pomiędzy konkretnymi maszynami. Sieci dynamiczne są nieco bardziej skomplikowane i posiadają przełączniki, które mogą, zależnie od potrzeb umożliwić dane połączenie lub nie. Przełączniki mogą posiadać dodatkowe funkcjonalności takie jak broadcast pakietów[28] lub routing[29]. Schematy sieci połączeń statycznych i dynamicznych zaprezntowano na Rys. 6.9.

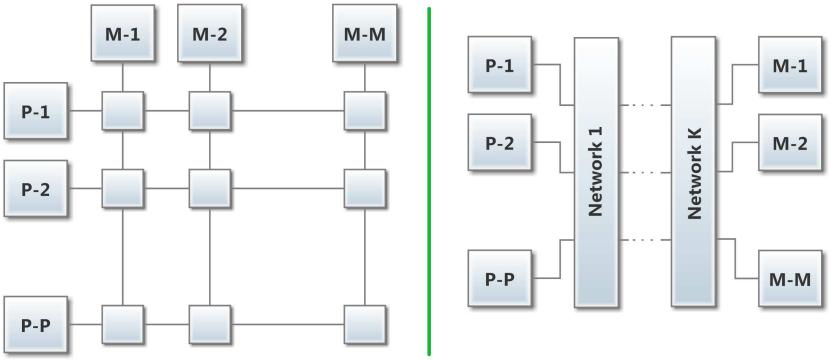
[28] Rozgłoszeniowy tryb transmisji danych.
[29] Wyznaczanie trasy i wysłanie nią pakietu danych w sieci komputerowej.



Rys. 6.9. Od lewej: sieci połączeń statycznych i dynamicznych. *Źródło: Wykonanie własne.*

Klasyczną topologią jaka może znaleźć zastosowanie podczas opracowywania komputera jest *wspólna magistrala*. Ze względu na prostotę, można ją stosować w rozwiązaniach posiadających maksymalnie kilkadziesiąt procesorów. Została zastosowana m.in. w Sun Enterprise [260].

We współczesnych superkomputerach są stosowane znacznie bardziej skomplikowane topologie. Rozwiązaniami, na które warto zwrócić uwagę są sieć wielostopniowa oraz sieć z przełącznicą krzyżową[30]. Zostały one zaprezentowane na Rys. 6.10.



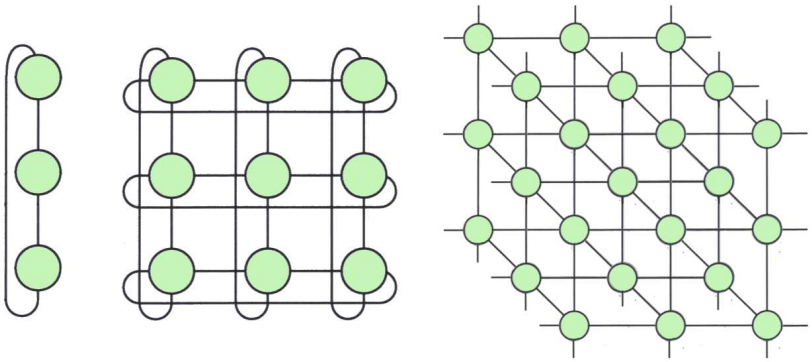
Rys. 6.10. Sieć oparta na przełącznicy krzyżowej *(po lewej)* i sieć wielostopniowa *(po prawej). Źródło: Wykonanie własne.*

Sieć oparta na przełącznicy jest siatką połączeń, w której za pomocą przełączników ustanawiamy odpowiednie połączenie między zasobami sprzętowymi. Przy sieci wielostopniowej dane są transmitowane przez kilka stopni przełączników. Przełącznice krzyżowe znalazły zastosowanie m. in. w Fujitsu VPP 500 [108], [26], Convex Exemplar [44] i NEC Earth Simulator [237]. Pierwszym komputerem wykorzystującym tą topologię był C.mmp [279].

Obecnie bardzo często wykorzystywana jest topologia oparta na wielowymiarowych torusach. W rozwiązaniu tym każdy węzeł może komunikować się ze swoimi

[30]Przykładowa sieć bez blokad. Połączenie dowolnego procesora z dowolnnym modułem pamięci nie blokuje połączeń pozostałych procesorów z innymi modułami

sąsiadami i np. na siatce jednowymiarowej węzeł skrajny także może przesyłać dane do dwóch sąsiadów, ponieważ za brakującego sąsiada przyjmuje się maszynę z przeciwnego krańca infrastruktury. Istnieje kilka wersji sieci opartych na torusach - 2D zaimplementowano w Cray T3E [71] oraz XT3 [267], 3D znalazło zastosowanie m. in. w BlueGene/L [5]. Opis możliwości sieci torus dla BlueGene/L przedstawiono w raporcie badawczym IBM [167]. Na Rys. 6.11 przedstawiono topologie Torus 1D, 2D oraz 3D.



Rys. 6.11. Topologie typu Torus 1D, 2D i 3D *(odpowiednio od lewej). Źródło: Wykonanie własne.*

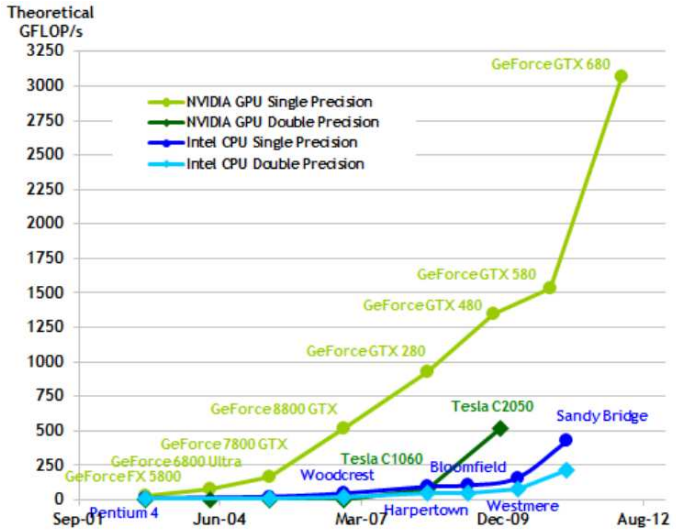
Pozostałe często wspominane topologie połączeń wykorzystywane w superkomputerach to omega [240] *(zastosowana m. in. w EM-4 [13])*, motyl [218] *(zaimplementowana w NYU Ultracomputer [8], BBN Butterfly [256])* oraz grubego drzewa [30] *(maszyna CM-5 [33])*. Mniej znanym rozwiązaniem jest sieć Clos [245] *(komputer Cray BlackWidow [58])*.

6.5. Akceleratory graficzne

Karty graficznie to obecnie bardzo silnie rozwijająca się gałęź metod przetwarzania. Na rynku rywalizuje ze sobą głównie dwóch producentów: nVidia i AMD *(niegdyś ATI)*. Według danych TOP500 w 53 najszybszych komputerach znajdują się akceleratory nVidia Fermi, a tylko w dwóch AMD Radeon [259]. Dysproporcja ta jest duża i przekłada się na cały rynek. Akceleratory graficzne wprowadziły dość dużą rewolucję w obliczeniach mimo, że są to układy o całkowicie innej architekturze niż zwykłe procesory. Można je sklasyfikować w taksonomii Flynna jako SIMD[31]. Są znacznie bardziej ograniczone w porównaniu do układów MIMD, jednak w niektórych zadaniach doskonale sobie radzą *(pierwotnie projektowane z myślą o przetwarzaniu grafiki)*. Dzięki GPU możliwa jest lepsza realizacja terapii z wykorzystaniem promieniowania X gdzie uzyskano przyśpieszenie rzędu 34x-98x [223].

[31] Faktycznie pewne rozwinięcie SIMD - SIMT *(ang. Single Instruction, Multiple Thread)* [162].

Pierwszym układem umożliwiającym wykorzystanie GPU do obliczeń był G80 [57], którego premiera odbyła się w listopadzie 2006 roku. Wraz z upływem czasu pojawiały się kolejne rozwiązania [162], [36]. Porównanie wydajności pomiędzy różnymi generacjami CPU i GPU przedstawiono na Rys. 6.12.



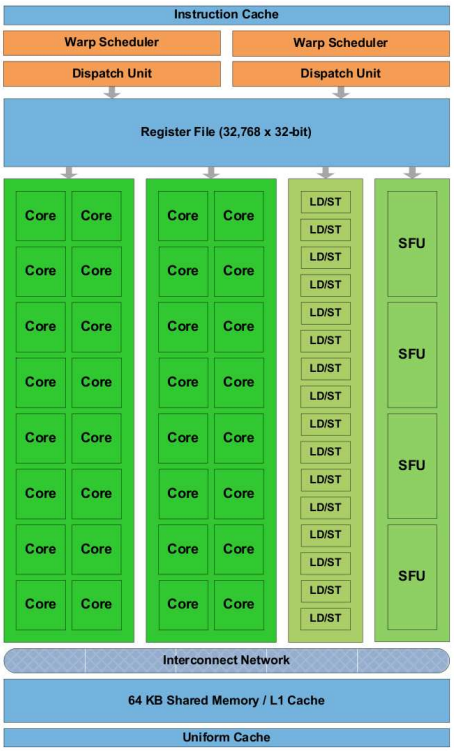
Rys. 6.12. Porównanie wydajności pomiędzy CPU, a GPU. *Źródło: nVidia [201].*

Mimo przodowania rozwiązań nVidia także AMD odnosiło sukcesy. Zespół z Chiba University oraz Shohoku University z wykorzystaniem kart graficznych AMD HD5000 generował hologramy z prędkością dwa razy wyższą niż było to możliwe na kartach marki nVidia [258].

Istnieje oprogramowanie umożliwiające automatyczne portowanie oprogramowania do kodu realizowanego przez procesory graficzne [233]. Nie daje ono zazwyczaj dobrych efektów.

6.5.1. nVidia Tesla i GeForce

Architektura układów GPU firmy nVidia zostanie opisana na bazie układów o nazwie kodowej *Fermi* [36] z częściowym odniesieniem do starszej generacji *Tesla* [162]. Zgodnie z uproszczoną kwalifikacją procesora graficznego do klasy SIMD musi posiadać on wiele procesorów strumieniowych realizujących na różnych zbiorach danych te same instrukcje. Procesory strumieniowe w układach nVidia są zgrupowane w multiprocesory, które posiadają znacznie większą autonomię *(m.in. niezależną pamięć współdzieloną)*. W rozwiązaniach klasy *Fermi* w jednym muliprocesorze odnajdziemy 32 jednostki przetwarzania co dla karty Tesla C2050 i 14 multiprocesorów w jej rdzeniu daje w sumie 448 jednostek przetwarzania danych [200]. Schemat jednego multiprocesora przedstawiono na Rys. 6.13.

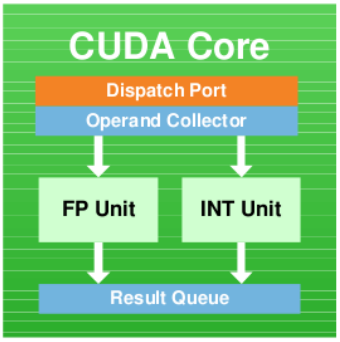


Rys. 6.13. Struktura multiprocesora strumieniowego w architekturze *'Fermi'*. *Źródło: nVidia [199].*

Najistotniejsze w obliczeniach wykonywanych na GPU jest to, że jak wcześniej wspomnieliśmy wszystkie te układy mogą realizować te same obliczenia. Klasyczne procesory CPU pod tym względem nie są niczym ograniczone i zbiór operacji wykonywanych na jednym rdzeniu jest całkowicie niezależny od tego co robi inny rdzeń. Innym ograniczeniem jest konieczność przesyłania danych do karty graficznej. GPU nie ma dostępu do RAM i wszystkie dane na jakich będzie operować muszą zostać przesłane przez procesor główny do pamięci karty graficznej, co jest kolejnym ograniczeniem. W przypadku gdy wymagana jest intensywna wymiana informacji CPU-GPU obliczenia na akceleratorze mogą nie być opłacalne.

W układach *Tesla* w pojedynczym procesorze strumieniowym znajdowała się jedna jednostka odpowiedzialna zarówno za obliczenia stałoprzecinkowe, jak i zmiennoprzecinkowe. W *Fermi* znajdują się dwa wyspecjalizowane podukłady - każdy do innego typu operacji. Różne rodzaje obliczeń nie mogą być jednak wykonywane jednocześnie. Starsze układy mnożenie liczb stałoprzecinkowych fizycznie wykonywały jedynie na 24 bitach, a operacje na większych rozmiarach były emulowane. Fermi potrafi realizować taką operację na rozmiarze 32-bit w jednym cyklu zegara. Rozwinięto także wsparcie dla normy IEEE 754-2008 i dodano obsługę instrukcji FMA [228] *(ang. Fused Multiply Add)*. Diagram przedstawiający procesor strumieniowy znajdujący się w nVidia Fermi został przedstawiony na Rys. 6.14[32].

[32]Więcej informacji o stronie sprzętowej można znaleźć w [93] lub w książce [82].



Rys. 6.14. Diagram przedstawiający budowę jednego procesora strumieniowego w architekturze *'Fermi'. Źródło: nVidia [199].*

Akceleratory graficzne firmy nVidia programuje się najczęściej w środowisku CUDA, które wspiera kilka języków programowania *(w tym C/C++, Fortran i Python)*. Na kartach graficznych główna funkcja, która jest wykonywana zawsze posiada kwantyfikator *__global__* i jest nazywana *kernelem*. Funkcja ta może wywoływać inne procedury jednak te muszą posiadać kwantyfikator *__device__* określający, że dane operacje będą wykonywane na urządzeniu *(akceleratorze)*. Kod, który ma być wykonywany na karcie graficznej jest tłumaczony do języka PTX *(ang. Parallel Thread Execution - odpowiednik asemblera dla CPU)* [149], a następnie sterownik konwertuje go do wersji binarnej, która jest realizowana na sprzęcie. Prosty kernel podnoszący wszystkie elementy danej tablicy do kwadratu przedstawiono na Listingu 13.

```
1 __global__ void gpu_kernel(float *table, int size)
2 {
3     int idx = blockIdx.x * blockDim.x + threadIdx.x;
4     if (idx<size) table[idx] = table[idx] * table[idx];
5 }
```

Listing 13. Kernel w standardzie CUDA wykonywany na karcie graficznej.

Kernel może być wywoływany z poziomu innych funkcji wykonywanych na CPU. Wywołanie przedstawiono na Listingu 14.

```
1 gpu_kernel <<< number_of_blocks, block_size >>> (table_device, size);
```

Listing 14. Wywołanie kernela CUDA.

W tym przypadku kernel zostanie zrealizowany z wykorzystaniem *number_of_blocks* bloków, z którego każdy będzie miał rozmiar *block_size*[33]. Wątki mogą być zorganizowane w jednym, dwóch lub trzech wymiarach, a każdy z nich posiada, dzięki środowisku CUDA, zdefiniowany wektor *threadIdx* określający położenie danego wątku w ramach bloku. W trzech wymiarach będą to odpowiednio *threadIdx.x*, *threadIdx.y*

[33]Przykład bazuje na organizacji 1D.

oraz *threadIdx.z*. Bloki mogą być ułożone w maksymalnie trójwymiarowe tablice nazywane *gridami (do określenia pozycji bloku w gridzie mozna wykorzystać wektor blockIdx)*, a każdy z nich jest realizowany sprzętowo przez jeden muliprocesor.

Każdy wątek jest wykonywany przez ten sam kernel. Przekazywany w parametrach kernela wskaźnik do tablicy nie może odnosić się do lokalizacji danych w pamięci operacyjnej RAM *(do tej GPU nie ma bezpośredniego dostępu)*, a do pamięci karty graficznej. Przed rozpoczęciem obliczeń należy więc przesłać odpowiednie dane do akceleratora. W celu wymiany informacji pomiędzy CPU, a GPU wykorzystywana jest najczęściej funkcja *cudaMemcpy*[34].

Na akceleratorze nie możemy wywoływać zwykłych funkcji z bibliotek wykorzystywanych na CPU. W szczególności, jak łatwo sobie wyobrazić, nie możemy wykorzystać funkcji *printf*, gdyż GPU nie mógłby *'wprost'* z wykorzystaniem środowiska obliczeniowego wyświetlić wyników na ekranie. Do wizualizacji wyników można wykorzystać biblioteki OpenCL [239] lub DirectX [277]. Jest także dużo wyspecjalizowanych bibliotek do obliczeń numerycznych na CUDA - np. cuBLAS *(ang. CUDA Basic Linear Algebra Subroutines)* [202] i CULA [131].

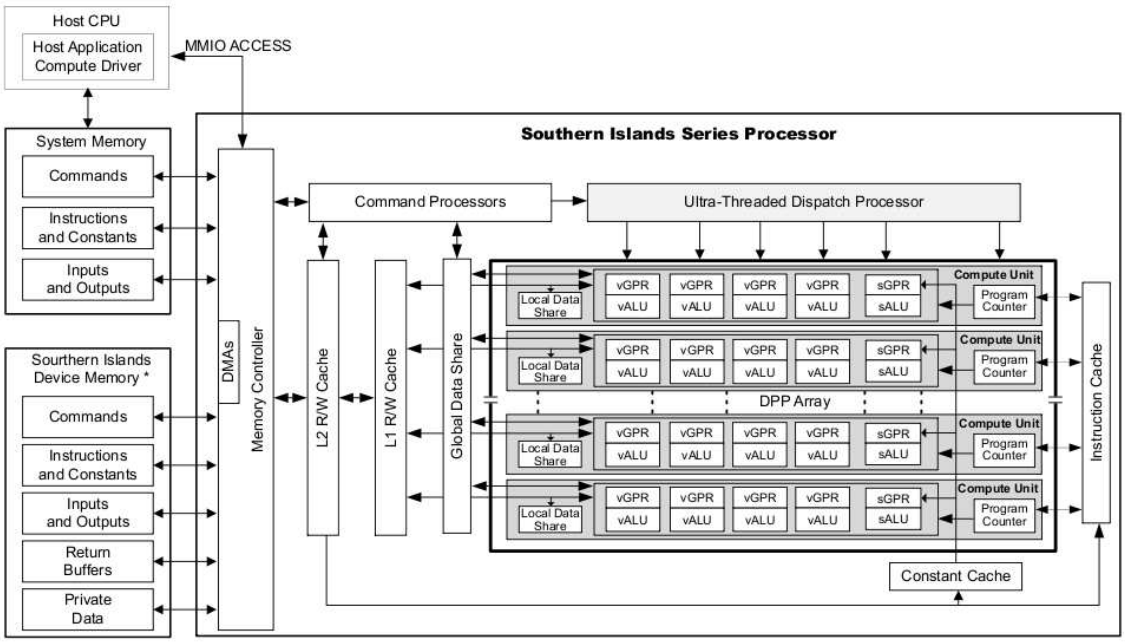
Szczegółowe dane odnośnie CUDA i akceleratorów graficznych firmy nVidia można odnaleźć w książkach Kirka [152] i Nylana [203]. Bardzo pomocny może okazać się także poradnik firmy nVidia [201]. Drobny artykuł o układzie Fermi napisał inżynier z Silicon Valley Peter Glaskowsky [93]. Sanders i Kandrot napisali także poradnik oparty na dużej liczbie przykładów [125].

6.5.2. AMD Radeon

Pierwsze karty graficzne AMD, które można było wykorzystywać do obliczeń pochodziły z serii HD2900. Za pomocą jednego Radeona HD2900 XT udało się uzyskać przyspieszenie od 6 do 60 razy w stosunku do wydajności na CPU podczas szyfrowania danych za pomocą AES/DES [126]. Procesory graficzne firmy AMD pokonały jednak bardzo dużą drogę przez kilka ostatnich lat. Najnowsza generacja HD7000 ma całkowicie nową architekturę.

Uproszczony schemat procesora *'Southern Islands'* przedstawiono na Rys. 6.15.

[34] Środowisko CUDA ma wiele specyficznych możliwości odnośnie tego typu operacji np. przesyłanie danych może odbywać się w sposób synchroniczny lub asynchroniczny



Rys. 6.15. Struktura procesora graficznego AMD Radeon HD 7000 Series *'Southern Islands'. Źródło: AMD [10].*

Jednostki przetwarzające dane zlokalizowane są w *Compute Units*[35]. W jednej *CU* można odnaleźć cztery jednostki wektorowe *(vALU)* i cztery rejestry wektorowe *(vGPR)*. Jednostki wektorowe są głównymi układami, które realizują obliczenia. Są to rozwiązania zdolne do wykonywania operacji *16-way*[36] zarówno na liczbach zmiennoprzecinkowych pojedynczej precyzji jak i całkowitych oraz *4-way* dla liczb zmiennoprzecinkowych podwójnej precyzji. W *Compute Units* dostępna jest także jedna jednostka skalarna *(sALU)* wraz z dedykowanym rejestrem *(sGPR)* oraz pamięć podręczna *Local Data Share* umożliwiająca transfer 128 bajtów na jeden cykl zegara co daje maksymalnie 120 GB/s.

Akceleratory AMD kiedyś programowano z wykorzystaniem środowiska Brook [112]. Obecnie coraz częściej wykorzystuje się możliwości biblioteki OpenCL *(Open Computing Language)*. Biblioteka ta umożliwia nie tylko programowanie kart AMD, ale także np. układów graficznych nVidia oraz procesorów Cell. Istota OpenCL polega na tym, że inżynierowie opracowujący sprzęt udostępniają developerom OpenCL dokładne specyfikacje, które są następnie nanoszone do jednego standardu. Jeden kod może być więc zgodny z różnymi urządzeniami. Koncepcja OpenCL jest podobna do CUDA i bazuje na wywołaniu kerneli, które są wykonywane na jednostkach obliczeniowych. Kernel analogiczny do zaprezentowanego przy CUDA został przedstawiony na Listingu 15.

[35] Jednostka obliczeniowa.

[36] Wielodrożność procesora. Układ może w jednym cyklu realizować wiele równoległych operacji.

```
1 __kernel void dqft_cdouble(__global int *table, __global int block_size, __global↩
      int size)
2 {
3     int idx = block_size*get_global_id(0) + get_global_id(1);
4
5     if (idx<size) table[idx] = table[idx] * table[idx];
6 }
```

Listing 15. Kernel w standardzie OpenCL.

Porównanie kerneli może być mylne, ponieważ nie wskazuje na istnienie większych różnic pomiędzy OpenCL, a CUDA. W praktyce jednak różnice są znaczne i OpenCL skoro wspiera większą gamę urządzeń jest prawdopodobnie trudniejszy w początkowym opanowaniu.

Oczywiście wykonano wiele testów porównawczych pomiędzy CUDA, a OpenCL w poszukiwaniu odpowiedzi na pytanie *'Co jest szybsze?'*. Jeden z takich testów dla adiabatycznych algorytmów kwantowych[37] *(AQUA)*[38] i symulacji metody Monte Carlo [16] kwantowego układu spinowego przedstawili Karimi, Dickson i Hamze [139].

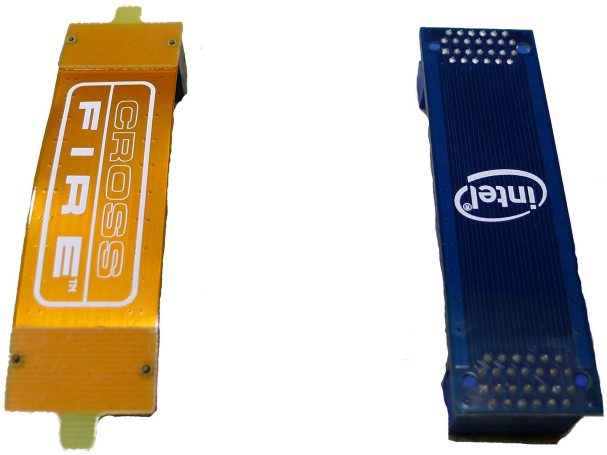
Dokładną specyfikację języka OpenCL można znaleźć w dokumentach [101], [150], a szczegóły odnośnie możliwości wykorzystania i optymalizacji w [226].

6.5.3. Karty wieloprocesorowe i tryby pracy współbieżnej

Możliwość skalowania pracy kilku akceleratorów graficznych została wprowadzona już w 1998 roku w kartach firmy 3dfx Voodoo 2. Dwie karty można było wówczas połączyć dodatkowym mostkiem, co umożliwiało pracę w trybie przeplatania linii. Pierwsza karta mogła generować linie parzyste, druga nieparzyste, które następnie były składane w klatkę wynikową.

W 2004 roku razem z kartami GeForce 6 technologia ta została przywrócona do życia. Analogiczne rozwiązanie o nazwie CrossFire wprowadziło także ATI. Producenci rozpoczęli budowę kart graficznych posiadających nie tylko jeden GPU, a nawet dwa. Pierwsze karty tego typu to GV-3D1 i GV-3D1-68GT. Były to karty oparte na GeForce 6600 i 6800. W marcu 2008 roku nVidia zaprezentowała kartę GeForce 9800 GX2 posiadającą dwa procesory G92, które już można było wykorzystać za pomocą CUDA. Wielordzeniowe karty znalazły się także w następnych generacjach jako GTX 295 i GTX 590. Współczesne mostki za pomocą, których można połączyć dwie karty graficzne zaprezentowano na Rys. 6.16.

[37] AQC *(ang. adiabatic quantum computation)* [72].
[38] http://aqua.dwavesys.com



Rys. 6.16. Mostki do łączenia kart graficznych zgodnych z CrossFire i SLI.

Karty zbudowane z dwóch układów cechują się teoretycznie dwa razy większą wydajnością od swoich jednoukładowych odpowiedników. W praktyce jednak ze względu na bardzo wysoką częstotliwość pracy i dużą ilość wydzielanego ciepła zmniejsza się wydajność układów tak, aby temperatura podczas pełnego obciążenia nie przekraczała progu krytycznego[39].

Programować tego typu karty można także z wykorzystaniem CUDA lub OpenCL[40]. Mimo, że takie podejście jest trudniejsze od programowania systemów z jednym akceleratorem często znajduje zastosowanie [22], [180].

6.6. Pozostałe urządzenia

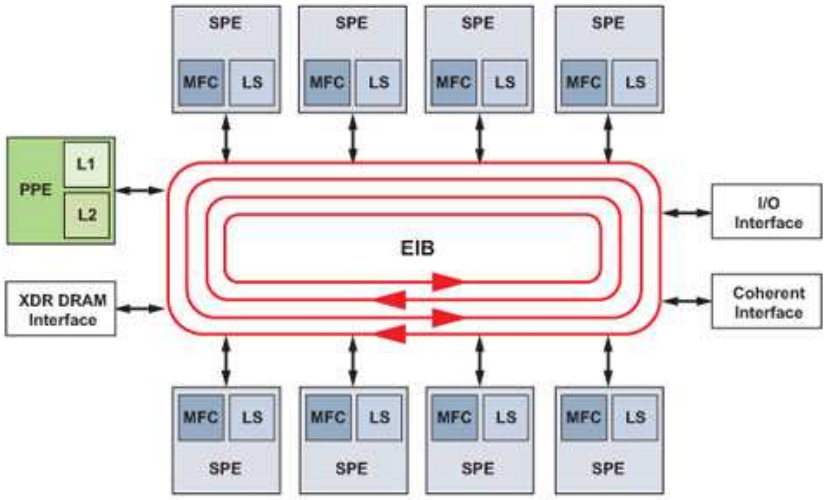
Oprócz komputerów z pamięcią wspólną, kart graficznych nVidia i AMD oraz klastrów komputerowych istnieje wiele innych urządzeń, które można wykorzystać do programowania. Jednymi z nich są procesory IBM Cell zastosowane w modułach QS22 oraz konsoli PlayStation 3, a także procesory ARM, które wykorzystano na platformie BOINC.

6.6.1. IBM Cell Broadband Engine Architecture

Procesor Cell został opracowany w sojuszu STI zawiązanym przez firmy Sony, Sony Computer Entertainment, Toshiba i IBM. Opracowanie architektury trwało w STI Design Center w Austin *(stan Texas)* 4 lata i kosztowało 400 mln dolarów [215]. Pierwszym wielkoskalowym zastosowaniem procesora Cell była konsola do

[39] Rdzeń potrafi wytrzymać od 100 do nawet 120°C w zależności od rodzaju.
[40] Można sobie to najprościej wyobrazić jako wywołanie asynchroniczne dwóch kerneli.

gier PlayStation 3. Układ ten jest powszechnie uznawany jako jeden z najtrudniejszych w programowaniu [37] lecz mimo to wykazano jego skuteczność w niektórych zastosowaniach naukowych [41]. Produkt miał pełnić rolę rozwiązania pośredniego pomiędzy zwykłymi procesorami AMD Athlon X2 i Intel Core 2 Duo, a kartami graficznymi. Procesor składa się z odpowiednich struktur I/O, głównego elementu przetwarzania o nazwie PPE *(ang. Power Processing Element)* oraz ośmiu w pełni funkcjonalnych układów SPE *(ang. Synergistic Processing Elements)*. Całość łączy wyspecjalizowana magistrala o wysokiej przepustowości EIB *(ang. Element Interconnect Bus)*. Uproszczony schemat procesora przedstawiono na Rys. 6.17.



Rys. 6.17. Architektura procesora IBM Cell. *Źródło: NASA [241].*

Z punktu widzenia logiki PPE jest najbardziej zaawansowanym układem. Za jego pomocą możemy na platformie z procesorem Cell uruchomić normalną dystrybucję Linux'a[41]. PPE ma kontrolę nad SPE - potrafi uruchamiać, zatrzymywać i przerywać wszystkie procesy na nich wykonywane *(posiada listę dodatkowych instrukcji)*.

Zarówno PPE jak i SPE są wykonane w architekturze RISC. PPE posiada 64KB pamięci cache pierwszego poziomu *(32KB dla danych i tyle samo dla instrukcji)* i 512 KB pamięci drugiego poziomu. Potrafi on zrealizować dwie operacje o podwójnej precyzji w jednym cyklu zegarowym, co przy częstotliwości taktowania 3.2 GHz przekłada się na 6.4 GFLOPS. Podczas wykonywania obliczeń na liczbach zmiennoprzecinkowych pojedynczej precyzji wydajność wzrasta do 25.6 GFLOPS, ponieważ układ może teoretycznie zrealizować aż osiem tego typu instrukcji w jednym cyklu.

Każdy SPE posiada MFC, w którego skład wchodzi DMA[42], MMU[43] i interfejs magistrali. SPE posiadają 256 KB wbudowanej pamięci SRAM dla instrukcji i danych *(Shared Memory)*. Układ ten potrafi realizować obliczenia na ośmiu 16-

[41] Musi być to odpowiednia wersja dla procesorów PowerPC.
[42] Direct memory access. Technika bezpośredniego dostępu do zasobów.
[43] Memory management unit.

bitowych liczbach całkowitych, czterech 32-bitowych liczbach całkowitych lub czterech liczbach zmiennoprzecinkowych pojedynczej precyzji w jednym cyklu zegarowym. Dzięki takim parametrom może on uzyskać wydajność aż 25.6 GFLOPS [61], co w całym procesorze Cell daje 230.4 GFLOPS *(8x SPE + 1x PPE)*. W przypadku podwójnej precyzji wydajność spada do 1.8 GFLOPS co całkowicie daje 20.8 GFLOPS. W 2008 roku IBM wprowadziło wersję procesorów oznaczoną jako PowerXCell 8i w której poprawiono wsparcie dla liczb o podwójnej precyzji. Ogólna wydajność w tym trybie dla nowych układów wynosi 108.8 GFLOPS. Testy przeprowadzone przez inżynierów IBM ukazały, że SPE mogą osiągnąć aż *98%* swojej teoretycznej wydajności [61].

Ciekawym rozwiązaniem jest magistrala EBI. Jest ona zrealizowana za pomocą czterech pierścieni. Podłączony do nich jest PPE, osiem SPE, kontroler pamięci MIC, i dwa układy odpowiadające za operacje I/O. Daje to w sumie 12 bloków funkcyjnych z którymi niezbędna jest komunikacja. Topologia pierścienia powoduje, że odległość pomiędzy dwoma dowolnymi układami podłączonymi do magistrali nie może być większa niż 6. Za pomocą jednego pierścienia mogą być realizowane maksymalnie trzy transakcje o rozmiarze 16B równocześnie. EBI pracuje z połową częstotliwości zegara systemowego. Przepustowość EBI może więc wynosić $\frac{16B \cdot 12}{2}$ co daje 96B na cykl. Przy częstotliwości 3.2 GHz jest to ponad 300 GB/s. Ulepszone procesory PowerXCell 8i stały się podstawą m.in. modułów Blade QS22, które zastosowano w superkomputerze Roadrunner zainstalowanym w Los Alamos National Laboratory, mieszczącym się w Narodowym Laboratorium Los Alamos na potrzeby amerykańskiego Departamentu Energii [155]. W praktyce z wykorzystaniem benchmarka Linpack [119] udało się uzyskać *77.8%* teoretycznej wydajności dla operacji ograniczających się do jednego węzła i *74.6%* dla całego systemu Roadrunner[44] [153].

6.6.2. Intel HD Graphics

Najnowsze procesory firmy Intel posiadają wbudowane układy graficzne. Nie umożliwiają one wykonywania tak zaawansowanych operacji jak dedykowane rozwiązania AMD lub nVidia jednak można je wykorzystać do obliczeń. Intel HD Graphics *(tak nazywa się rodzina układów graficznych wbudowanych w CPU Intela)* jest wspierane przez OpenCL. Nie przeprowadzono żadnych doświadczeń na tych rozwiązaniach, ponieważ w tym przypadku obsługiwany jest jedynie system Windows.

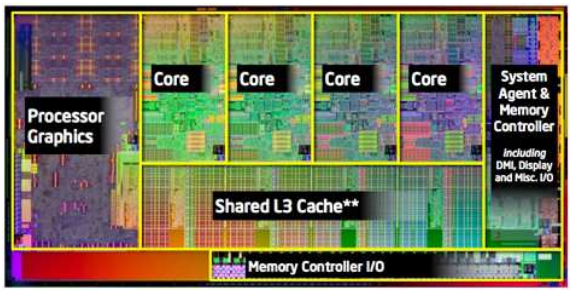
Podstawowym elementem w GPU, który został wyodrębniony w architekturze opracowanej przez Intela jest EU *(ang. Execution unit)*[45]. Przykładowe rozwiązanie

[44] Wyczerpującym źródłem informacji na temat IBM Cell jest RedBook [4], a od strony sprzętowej o wszystkich produktach BladeCenter można przeczytać w RedPaper [54].

[45] W inżynierii komputerowej jest to jednostka funkcjonalna będąca częścią procesora, która może mieć m.in. własną jednostkę sterowania i rejestry.

działające w tandemie w procesorze Core i7-2700K jest wyposażone w 12 EU działających z częstotliwością taktowania 1350 MHz. Każdy z nich posiada dwie jednostki teksturujące, z których każda potrafi wykonywać równolegle 4 wątki. Wydajność więc w tym przypadku będzie wynosiła $12 \cdot 2 \cdot 4 \cdot 1350\ MHz = 129.6\ GFLOPS$ co pokrywa się z danymi Intela dla *HD Graphics 3000* [127]. Jeżeli uwzględnimy fakt, że pierwsza architektura zintegrowanych GPU z 2010 roku w procesorach Intela o nazwie kodowej *Westmare* cechowała się wydajnością do *43 GFLOPS* [127] i nie umożliwiała programowania to możemy przyjąć dość duże zmiany w architekturze.

Na Rys. 6.18 przedstawiono strukturę procesora Core 2 Duo drugiej generacji wraz z układem graficznym.



Rys. 6.18. Struktura procesora z rodziny Intel Sandy Bridge. *Źródło Intel.*

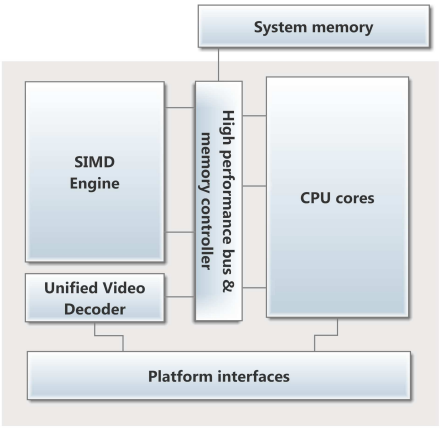
6.6.3. AMD APU

Analogiczne rozwiązanie do HD Graphics wprowadziło także AMD. Dużo można było usłyszeć o projekcie *Fusion*, który miał na celu integrację CPU z GPU. Pierwszy układ Fusion do desktopów pochodzi z architektury *Llano*. Jest on połączeniem Radeona z serii HD6000 i procesora k10[46]. GPU posiada 400 procesorów strumieniowych pracujących z częstotliwością 600 MHz dzięki czemu uzyskuje aż 480 GFLOPS. Jest to znacznie więcej od układów graficznych zintegrowanych z procesorami Intela i najprawdopodobniej dlatego rozwiązania AMD z tego segmentu są częściej wykorzystywane do obliczeń.

Efektywność połączenia CPU z GPU w wykonaniu AMD oraz empiryczną wydajność układów *Zacate* opisał Mayank [176], a na rodzinie *Llano* skupił się Branover [25].

Dość podstawowe informacje o architekturze APU można znaleźć także w dokumencie AMD [9]. Schemat blokowy AMD APU został przedstawiony na Rys. 6.19.

[46] Architektura procesorów AMD w której skład wchodzą m.in. Phenom, Phenom II i Athlon II.



Rys. 6.19. Diagram przedstawiający schemat układu AMD APU z wydzielonymi głównymi blokami funkcyjnymi *(jako 'SIMD Engine' oznaczono zintegrowany prosecor graficzny).*

6.6.4. Procesory oparte o architekturę ARM - komputery jednopłytkowe

ARM *(ang. Advanced RISC Machine)* jest 32-bitową architekturą procesorów RISC. Są one stosowane głównie w systemach wbudowanych ze względu na niskie zapotrzebowanie na energię. Procesory ARM są jednymi z najczęściej wykorzystywanych układów do budowy telefonów komórkowych lub routerów.

Już w 1996 roku zaprojektowano układ StrongARM, który osiągał częstotliwość 233 MHz i pobierał mniej niż 1W mocy, na bazie którego potem Intel opracował układ Xscale [43]. Lista instrukcji ARM jest rozwinięciem możliwości 8-bitowego mikroprocesora MOS Technology 6502, który był wykorzystywany w latach 80. XX wieku. Rozkazy są tak skonstruowane aby były wykonywane w jednym cyklu zegarowym. Klasycznym tego przykładem jest algorytm Euklidesa realizowany na ARM. Na Listingu 16 przedstawiono kod asemblera x86 realizujący algorytm Euklidesa.

```
1  loop:
2      CMP eax,    ebx
3      JG  greater
4      JL  less
5      JMP end
6  greater:
7      SUB eax,    ebx
8      JMP loop
9  less:
10     SUB ebx,    eax
11     JMP loop
12 end:
```

Listing 16. Algorytm Euklidesa w asemblerze NASM x86.

Dla porównania kod wykorzystujący instrukcje ARM przedstawiono na Listingu 17.

```
1 gcd CMP r0, r1
2     SUBGT   r0, r0, r1
3     SUBLT   r1, r1, r0
4     BNE gcd
```

Listing 17. Algorytm Euklidesa w asemblerze ARM.

W implementacji wykorzystującej możliwości procesora ARM rozgałęzienia instrukcji *IF* są niewidoczne.

Procesory ARM często nie posiadają FPU, a operacje na liczbach zmiennoprzecinkowych są jedynie emulowane przez CPU. Bardziej zaawansowane modele moga realizować operacje zmiennoprzecinkowe w sposób sprzętowy *(np. Raspberry Pi)*. Wydajność układów Intel Atom i ARM Cortex-A8 została porównana przez Katie Roberts-Hoffman i Pawankumar Hegde z UT Dallas [144]. Na platformach sprzętowych ARM zazwyczaj jest instalowany system operacyjny oparty na jądrze Linux'a [280].

6.7. Komputery heterogeniczne

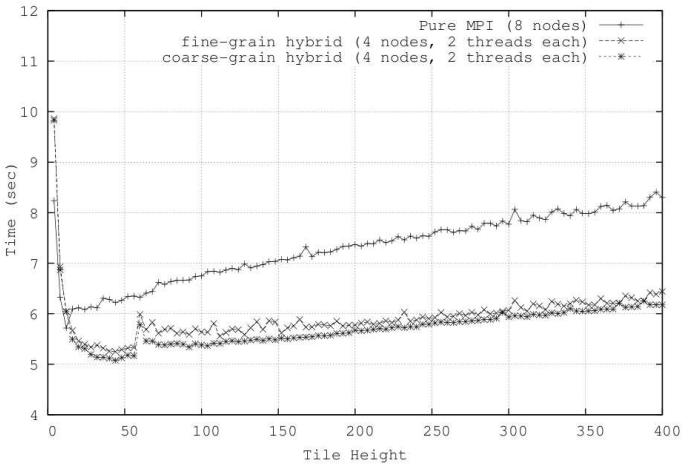
Komputery często posiadają wiele różnego rodzaju jednostek obliczeniowych. Poziom heterogeniczności najnowszych maszyn wzrasta, ponieważ mimo zauważalnej w pewnych obszarach unifikacji[47], procesory potrafiące realizować większe spektrum obliczeń w rzeczywistości składają się z wielu wyspecjalizowanych jednostek[48].

Wszystkie opisane wcześniej architektury obliczeniowe zostały scharakteryzowane z pominięciem możliwości ich łączenia. W praktyce jednak w nowoczesnym komputerze znajdziemy wielordzeniowy procesor z akceleratorem graficznym. Co więc zrobić jeżeli w jednym programie zechcemy wykorzystać jednocześnie zarówno wiele rdzeni CPU, jak i GPU? Problem ma większą skalę, ponieważ pod znakiem zapytania staje budowanie klastrów komputerowych z akceleratorami graficznymi - MPI wspiera jedynie zwykłe procesy wykonywane na CPU. W środowisku zróżnicowanym pod względem sprzętu możemy jednak pisać programy które będą wykorzystywały zarówno OpenMP i CUDA, jak i MPI i OpenCL lub inne kombinacje. Spotykane są także obliczenia na klastrach komputerowych, w których na poziomie węzła jednym wielowątkowym procesem zarządza OpenMP, a samą komunikację pomiędzy różnymi maszynami obsługuje MPI. Daje to często lepsze wynik niż zastosowanie tylko MPI [111] [225]. Aspekt ten w pewnych przypadkach dotyczących obliczeniowej dynamiki płynów [73] na komputerze SGI Origin 2000 sprawdziła NASA [120].

[47] Integracja GPU z CPU.

[48] Przykładowo IBM Cell lub procesory posiadające zintegrowany układ graficzny. Kiedyś wiele funkcji było implementowane przez oddzielne układy SOC *(ang. system-on-chip)*, które zostały z czasem przeniesione do nowych procesorów.

Przykładowe porównanie MPI z hybrydowym MPI + OpenMP zaprezentowano na Rys. 6.20.



Rys. 6.20. Porównanie wydajności MPI z połączeniem hybrydowym MPI + OpenMP. *Źródło: [111]*

6.8. Systemy obliczeń rozproszonych

Obliczenia rozproszone są dziedziną nauki skupiającą się na analizie działania systemów rozproszonych. O tym rodzaju obliczeń wspomniano już krótko w rozdziale 6.2. System rozproszony składa się najczęściej z wielu niezależnych komputerów, które komunikują się ze sobą za pomocą sieci komputerowej. Nazwa *obliczenia rozproszone* wywodzi się głównie z idei rozproszenia fizycznego komputerów na różnych obszarach geograficznych. Termin ten jednak jest często wykorzystywany w różnych kontekstach[49] [1].

Idea systemów rozproszonych ma swój początek wraz z projektem ARPANET[50]. Aktualnie wiele tego typu rozwiązań jest często opartych o komputery wolontariuszy, które są połączone za pomocą sieci Internet. W takim przypadku maszyny nie należą fizycznie do jednego właściciela. Podobnie także czasem korporacje łączą pewne zespoły obliczeniowe *(często złożone z zaawansowanych stacji roboczych lub klastrów komputerowych)* zlokalizowane np. w różnych miastach. Tego typu system w literaturze często nazywane są *gridem*. Z punktu widzenia nauki jednym z najbardziej znanych systemów rozproszonych jest WLCG[51] *(ang. Worldwide LHC Computing Grid)* [235], [18], który obejmuje globalną współpracę 170 centrów komputerowych

[49] Autorzy czasem procesami rozproszonymi określają nawet te, które wykonują się na jednym komputerze jednak zachowują swoją autonomię.

[50] Advanced Research Projects Agency Network. System rozproszony opracowany dzięki inicjatywie Pentagonu w celu decentralizacji wojskowych systemów łączności, które powinny działać nawet w przypadku zniszczenia infrastruktury telekomunikacyjnej państwa [224].

[51] http://wlcg.web.cern.ch

zlokalizowanych w 36 krajach na całym świecie. Tak zaawansowany system rozproszony umożliwia przesyłanie, analizę oraz przechowywanie około 15 PB[52], które są każdego roku generowane przez Wielki Zderzacz Hadronów [213]. Zarządzanie takimi ilościami danych nie jest wcale nowością i nie dotyczy tylko nauki. Już w 2008 roku Google analizowało 24 PB danych dziennie [128] i potrafiło posortować 1 PB w 6 godzin i 2 minuty wykorzystując do tego celu 4000 komputerów [98].

Systemem rozproszonym z którym większą styczność miało duże grono internautów jest BOINC[53] *(ang. Berkeley Open Infrastructure for Network Computing)*. Platforma rozwijana przez Uniwersytet Kalifornijski w Berkeley początkowo stanowiła zaplecze dla projektu naukowego SETI[54] mającego na celu nawiązanie kontaktu z pozaziemskimi cywilizacjami, o którym napisano wiele publikacji naukowych [53], [156], [248].

Aktualnie platforma BOINC jest wykorzystywana nie tylko na potrzeby UC Berkeley, ale także innych ośrodków badawczych na całym świecie.

Ze względu na główny temat tej pracy w pierwszej kolejności warto wspomnieć o projekcie AQUA@Home, który był tworzony przez D-Wave Systems. Celem projektu było przewidzenie możliwości nadprzewodzących adiabatycznych komputerów kwantowych[55]. W ramach wyników została opublikowany artykuł [140]. IBM wspiera także projekt World Community Grid mający na celu m.in. szukaniu lekarstwa na AIDS [229], leczenie dystrofii mięśniowej [268] i wydajniejsze wykorzystanie naturalnych źródeł energii [130]. Projekt ten ma jeden z największych dorobków naukowych[56]. Swoje projekty z wykorzystaniem BOINC posiada także University of Oxford[57] i University of Washington[58].

System rozproszony opierający się na komputerach wykorzystywanych przez osoby postronne musi posiadać dodatkowe cechy takie jak:

- możliwość pracy na różnych platformach sprzętowych i programowych,
- uwzględnienie niepewności zwracanych wyników i ich odpowiednia weryfikacja gdyż mogą być one błędne,
- uwzględnienie możliwości całkowitej utraty łączności komputerów z systemem i nie zwrócenia wyników cząstkowych realizowanych przez dane maszyny,
- uwzględnienie dodatkowych preferencji użytkowników pracujących na komputerach *(np. niskie wykorzystanie procesora, łącza)*,

[52] Dane aktualne na październik 2008 roku.
[53] http://boinc.berkeley.edu
[54] http://setiathome.ssl.berkeley.edu
[55] Istnieje model adiabatyczny i obwodowy komputera kwantowego.
[56] Na dzień 12 września 2012 roku udostępnione 34 publikacje naukowe.
[57] Climateprediction.net
[58] http://ralph.bakerlab.org

- trudny do określenia aktualny stan systemu *(komputery kontaktują się z wykorzystaniem różnych łącz w różnych topologiach, aktualny ich stan jako części składowych systemu może być nieznany).*

Na platformie tej klasy nie uruchomimy wszystkich algorytmów możliwych do realizacji na komputerach równoległych. Algorytmy rozproszone są pewnym podzbiorem algorytmów równoległych. Procesy realizowane na jednej maszynie mogą w niektórych przypadkach nie dysponować informacjami odnośnie całego problemu, a jedynie ich bardzo drobną częścią[59]. W szczególności nie mogą one dysponować w czasie rzeczywistym danymi wygenerowanymi przez inne podsystemy, a więc i na ich podstawie wykonywać rozgałęzień programu. O przetwarzaniu rozproszonym w systemie UNIX napisał Michel Gabassi [183]. Problematyka algorytmów rozproszonych została poruszona w wielu publikacjach [247], [1], [209].

W momencie gdy poznamy możliwości programowania na wielu platformach *(o większości z nich krótko wspomniano we wcześniejszych podrozdziałach)* możemy spróbować uruchomić tego typu projekt.

[59] W skrajnym przypadku wszystkie dane mogą nie zmieścić się w pamięci fizycznej jednego komputera.

7. Realizacja symulacji obliczeń kwantowych

Algorytm Grovera i algorytm Shora przetestowano na wielu komputerach i systemach operacyjnych.

7.1. Środowisko programistyczne i wykorzystywane biblioteki

Podczas pomiarów wykorzystano głównie systemy operacyjne FreeBSD [163] [160] oraz Linux [172] [133] w dystrybucjach Ubuntu Server [158], Debian [52], Slackware [56], Fedora [262] i openSUSE [178] i innych.

Na klastrze AGH Cyfronet *Zeus* dostępne środowisko programowe to system Scientific Linux[60] rozwijany przez CERN i Fermilab[61]. Kompilator Intela 11.1 i MPICH w wersji 1.2.7. W konfiguracji HP Cluster Platform 3000 BL 2x220 dostępnych 13944 rdzeni Intel Xeon L/X/E56XX zgrupowanych m.in. w 256 dwuprocesorowych węzłach Intel Xeon L5420, 256 dwuprocesorowych węzłach Intel Xeon L5640, 342 dwuprocesorowych węzłach Intel Xeon X5650, 234 dwuprocesorowych węzłach Intel Xeon E5645.

W wirtualnym laboratorium Intela MTL[62] zainstalowany system operacyjny to Redhat Enterprise Linux [210]. Dostępne kompilatory GNU pochodziły z GCC w wersji 4.1.2, a kompilatory Intela były oznaczone jako 11.1. Sprzętowo dostępne w jednym węźle cztery dziesięciordzeniowe, dwudziestowątkowe *(dzięki technologi HT)* procesory Intel Xeon E7-4860 oraz 256 GB pamięci RAM.

W IBM CSDL[63] dostęp do procesora IBM/S390 instalowanego w maszynach zSeries [109] system operacyjny RedHat i pakiet GCC w wersji 4.4.6.

Na systemie Tesla UMCS dostępny system operacyjny CentOS [34]. Pakiet GCC w wersji 4.1.2, CUDA 3.2 i OpenCL 1.1. Platforma sprzętowa wyposażona w procesor Intel Core i7 950, 24 GB pamięci RAM i dwie karty graficzne nVidia Tesla C2050 i GeForce GTS250.

Na platformie IBM Blade QS22 zainstalowany system operacyjny Fedora X. Dostępne dwa standardowe procesory IBM PowerXCell 8i i 8 GB pamięci RAM. Pakiet GCC w wersji 4.3.0. Środowisko OpenCL 1.1.

Do drobnych obliczeń wykorzystano moduł Tesla S1070 wyposażony w cztery karty graficzne Tesla C1060 mający do dyspozycji 4 GB pamięci RAM.

[60] https://www.scientificlinux.org

[61] Fermi National Accelerator Laboratory. http://www.fnal.gov

[62] Intel Manycore Testing Lab. http://software.intel.com/en-us/intel-manycore-testing-lab

[63] Linux Community Development System.
http://www-03.ibm.com/systems/z/os/linux/support/community.html

Do analizy dynamicznej i profilowania programów wykorzystano Valgrind [196] i Gprof [250].

Platforma BOINC uruchomiona dnia 13 sierpnia 2012 roku. Pierwsza zastosowana rewizja systemu Berkeley to 26000 *(z późniejszymi aktualizacjami)*. Platforma BOINC OProject@Home[64] dodatkowo wspiera aplikacje oparte na systemie Windows, OS X [74] i Android [49].

Wykorzystane zasoby sprzętowe zostały zestawione w załączniku E.

7.2. Algorytm Grovera

Algorytm Grovera bazuje głównie na operacji mnożenia wektora przez macierz. W przypadku tego algorytmu będziemy więc potrzebować efektywnych funkcji z dziedziny algebry liniowej. Wersje równoległe tego algorytmu zaimplementowano w OpenMP, CUDA oraz OpenCL.

7.2.1. Komputery z pamięcią wspólną

Listing 18 prezentuje funkcję mnożenia wektora przez macierz napisaną w standardzie OpenMP.

```
1  template<typename type>
2  void matrix_mul_vector_openmp(type matrix_A[], type vector_input[], type ↩
       vector_output[], unsigned long long int size, int number_of_threads)
3  {
4          unsigned long long i;
5          unsigned long long j;
6
7          for(i = 0; i < size; i++)
8          {
9                  vector_output[i] = 0;
10         }
11
12         #pragma omp parallel for num_threads(number_of_threads) shared(↩
               vector_output, matrix_A, vector_input, size) private(i, j) reduction ↩
               (+: vector_output)
13         for (i = 0; i < size; i++)
14         {
15                 for (j = 0; j < size; j++)
16                 {
17                         vector_output[i] += matrix_A[i*size+j] * vector_input[j];
18                 }
19         }
20 }
```

Listing 18. Szablon funkcji równoległej napisanej w środowisku OpenMP mnożenia wektora przez macierz dla liczb rzeczywistych.

[64] http://oproject.info

Głównym elementem tej funkcji jest tzw. pragma zapisana w 14 linijce. Definiuje ona ilość wątków za pomocą, których ma zostać zrównoleglona pętla *for* oraz zmienne prywatne i współdzielone w ramach tego bloku obliczeń. Dodatkowo wymagana jest także operacja redukcji dla tablicy *vector_output[]*.

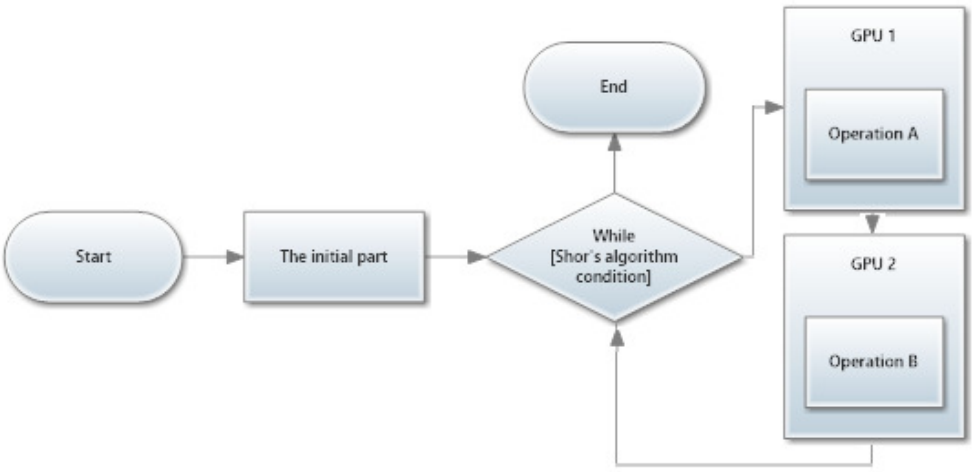
7.2.2. Akceleratory graficzne - środowisko CUDA

Na Listingu 19 przedstawiono funkcję mnożenia wektora przez macierz dla standardu CUDA.

```
1  __global__ void matrix_mul_vector_cuda_kernel( float *x, float *A, float *y, ↩
       unsigned long long int n)
2  {
3      unsigned long long int i, j;
4      i = blockIdx.x * blockDim.x + threadIdx.x;
5
6      if(i < n)
7      {
8                  float t = 0.0f;
9
10                 for(j = 0; j < n; j++)
11                 {
12                         t += A[i + j * n] * y[j];
13                 }
14
15                 x[i] = t;
16     }
17 }
```

Listing 19. Kernel implementacji procedury mnożenia wektora przez macierz dla liczb zmiennoprzecinkowych typu float w środowisku CUDA.

Funkcja ta w porównaniu do wersji OpenMP została w widoczny sposób pozbawiona jednej pętli ze względu na wykonywanie operacji na różnych danych przez wątki CUDA. Dodatkowa funkcja warunkowa *if(i < n)* zabezpiecza przed wyjściem obliczeń poza dany zakres. Operacja ta na akceleratorach graficznych wykonuje się bardzo szybko. Głównym ograniczeniem w tym przypadku będzie ilość pamięci dostępnej na karcie graficznej. Procedurę określoną wzorem 5.11 możemy jednak łatwo podzielić na dwie części A (5.12) i B (5.13) i rozbić obliczenia na dwa akceleratory graficzne. Dzięki takiemu rozwiązaniu będziemy dysponować dwa razy większą ilością pamięci na kartach graficznych. Schemat ideowy takiego rozwiązania zaprezentowano na Rys. 7.1.



Rys. 7.1. Schemat ideowy zrównoleglenia algorytmu Grovera z wykorzystaniem dwóch akceleratrów graficznych. *Źródło: Wykonanie własne.*

7.2.3. Pozostałe urządzenia - środowisko OpenCL

Implementacja w OpenCL jest bardzo podobna do implementacji z CUDA. Została ona przedstawiona na Listingu 20.

```
1  __kernel void matrix_mul_vector_float(__global float *x, __global float *A, ↩
       __global float *y, __global unsigned long *q)
2  {
3
4      unsigned long int idx = get_global_id(0);
5
6      if(idx < *q)
7      {
8              unsigned long int j;
9              float t;
10             t = 0;
11
12             for(j = 0; j < *q; j++)
13             {
14                     t += A[idx + j * *q] * y[j];
15             }
16
17             x[idx] = t;
18     }
19 }
```

Listing 20. Kernel implementacji procedury mnożenia liczb zmiennoprzecinkowych typu float w środowisku OpenCL.

Jedyną znaczącą różnicą jest zastosowanie innej funkcji zwracającej globalne ID wątku *get_global_id()*. Dodatkowo zmienne w nagłówku kernela posiadają dodatkowy specyfikator *__global* określający, że znajdują się one w pamięci globalnej urządzenia.

7.3. Algorytm Shora

Opis algorytmu Shora został przedstawiony w rozdziale 5.3. Bazuje on głównie na kwantowej transformacie Fouriera opisanej w 3.6. W tym podrozdziale zostaną opisane głównie techniki wykorzystane do optymalizacji *(implementacja sekwencyjna)* i zrównoleglenia *(implementacje równoległe)* obliczeń.

7.3.1. Implementacja sekwencyjna

Sekwencyjna implementacja kwantowej transformaty Fouriera została przedstawiona na Listingu 21.

```
1  void dqft_sp(complex_float q_register[], unsigned long long int q)
2  {
3      complex_float init[q];
4
5      unsigned long long i;
6      for(i = 0; i < q; i++)
7      {
8          init[i].real = 0;
9          init[i].imag = 0;
10     }
11
12     complex_float tmpcomp;
13
14     tmpcomp.real = 0.0;
15     tmpcomp.imag = 0.0;
16
17     float term_1;
18     term_1 = pow(q,-.5);
19
20     float term_2;
21
22     float epsilon;
23
24     epsilon = pow(10,-14);
25
26     unsigned long long int a;
27     unsigned long long int c;
28
29     float sin_value;
30     float cos_value;
31
32
33     for (a = 0 ; a < q ; a++)
34     {
35         term_2 = 2*PI*a/q;
36
37         if ((pow(q_register[a].real, 2) + pow(q_register[a].imag, 2)) > epsilon)
38         {
39             for (c = 0 ; c < q ; c++)
40             {
41                 term_2 = 2*PI*a*c/q;
42
```

```
43                  # ifdef __linux__
44                      sincosf(term_2, &sin_value, &cos_value);
45                  # else
46                      sin_value = sin(term_2);
47                      cos_value = cos(term_2);
48                  # endif
49
50                  tmpcomp.real = term_1 * cos_value;
51                  tmpcomp.imag = term_1 * sin_value;
52
53                  init[c] = complex_add(init[c], complex_mul(q_register[a], tmpcomp↩
                        ));
54              }
55          }
56      }
57
58
59      set_state_sp(q_register, q, init);
60
61      vector_normalization_sp(q_register, q);
62
63  }
```

Listing 21. Sekwencyjna implementacja kwantowej transformaty Fouriera.

Dla funkcji realizującej transformatę na wejściu podajemy wektor określający stan rejestru kwantowego oraz jedną zmienną określającą długość tego wektora. Po przekształceniach opisanych we wzorze 3.30 wynik jest zapisywany w wektorze wejściowym. Za pomocą makr zostało zdefiniowane wykorzystywanie funkcji *sin()* i *cos()*. W przypadku, gdy kompilujemy program na systemie operacyjnymi Linux zostanie wykorzystana funkcja *sincosf()* umożliwiająca w jednym momencie obliczenie dwóch wartości. W przeciwnym wypadku zostaną wykorzystane oddzielnie dwie funkcje *sin()* oraz *cos()*.

Implementacja ta bazuje głównie na pomyśle Hayward'a. Wyniki optymalizacyjne zostały przedstawione w tabelach 4.1, 4.2 oraz 4.3 w rozdziale 4.3.

7.3.2. Implementacje równoległe

Algorytm Shora zaimplementowano w wersjach dla komputerów z pamięcią wspólną *(OpenMP)*, klastrów komputerowych *(standard MPI)*, kart graficznych wspierających CUDA oraz innych urządzeń wykorzystujących środowisko OpenCL.

7.3.2.1 Komputery z pamięcią wspólną

Wersja równoległa kwantowej transformaty Fouriera napisana w standardzie OpenMP będzie musiała mieć rozwiązaną niedogodność operacji dodawania przedstawionej w linii 53 Listingu 21 *(implementacja sekwencyjna)*. W przypadku gdybyśmy pozostawili tą operację bez zmian mogłoby dojść do momentu gdy wartość

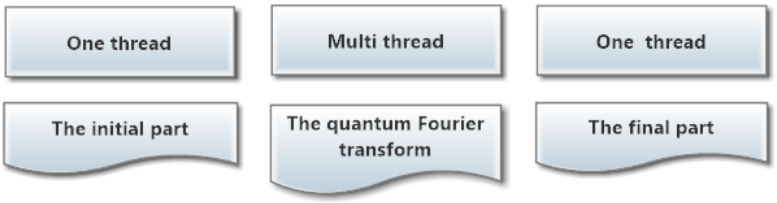
jednej komórki w tablicy będzie zwiększana w tym samym czasie przez dwa lub więcej wątków. W takim przypadku wynik ostateczny może być nieprawidłowy *(jedna lub kilka wartości może zostać nie dodana)*. Problem ten rozwiązano za pomocy tablicy pomocniczej. Alokowana jest dodatkowa dwuwymiarowa tablica. Każdy wątek posiada przydzielony jeden wiersz w tej tablicy *(numer wątku)* i to do niego zapisuje dane. Ostatecznie wykorzystywana jest dyrektywa OpenMP *shared* dla tablicy *init* i to do jej zerowego wiersza są sumowane wszystkie dane z poszczególnych wierszy tablicy pomocniczej. Operację, którą tutaj zasymulowano dla całej tablicy nazywa się *operacją atomową*. Standard OpenMP wspiera operacje atomowe jedynie względem zmiennych i dlatego zostało wykorzystane własne rozwiązanie. Implementację wykorzystującą OpenMP przedstawiono na Listingu 22, a poszczególne operacje alokacji, wstępnego i ostatecznego sumowania zostały zapisane w liniach 11, 46 oraz 57.

```
1  void dqft_openmp(complex_float q_register[], unsigned long long int q)
2  {
3      int nthreads;
4      nthreads = omp_get_num_threads();
5
6
7      complex_float **init;
8
9      unsigned long long i;
10
11     init = ( complex_float** ) malloc( sizeof( complex_float* ) * nthreads );
12
13     for ( i = 0; i < nthreads; i++ )
14     {
15         init[i] = (complex_float*) malloc(q * sizeof(complex_float));
16     }
17
18     unsigned long long j;
19     for(i = 0; i < q; i++)
20     {
21         for(j = 0; j < nthreads; j++)
22         {
23             init[j][i].real = 0;
24             init[j][i].imag = 0;
25         }
26
27     }
28
29     // ...
30
31     #pragma omp parallel for shared(q, q_register, init, epsilon, term_1) private↩
           (a, c, tmpcomp, term_2, sin_value, cos_value)
32     for (a = 0 ; a < q ; a++)
33     {
34
35         thread_id = omp_get_thread_num();
36
37         if ((pow(q_register[a].real, 2) + pow(q_register[a].imag, 2)) > epsilon)
38         {
39             for (c = 0 ; c < q ; c++)
40             {
```

```
41                  // ...
42
43                  tmpcomp.real = term_1 * cos_value;
44                  tmpcomp.imag = term_1 * sin_value;
45
46                  init[thread_id][c] = complex_add(init[thread_id][c], complex_mul(↩
                        q_register[a], tmpcomp));
47              }
48          }
49
50      }
51
52      #pragma omp parallel for shared(init, q, nthreads) private(a, j)
53      for (a = 0 ; a < q ; a++)
54      {
55          for(j = 1; j < nthreads; j++)
56          {
57              init[0][a] = complex_add(init[0][a], init[j][a]);
58          }
59      }
60
61      // ...
62
63  }
```

Listing 22. Implementacja kwantowej transformaty Fouriera w standardzie OpenMP.

Schemat ideowy zrównoleglenia algorytmu Shora w środowisku OpenMP został zaprezentowany na Rys. 7.2.

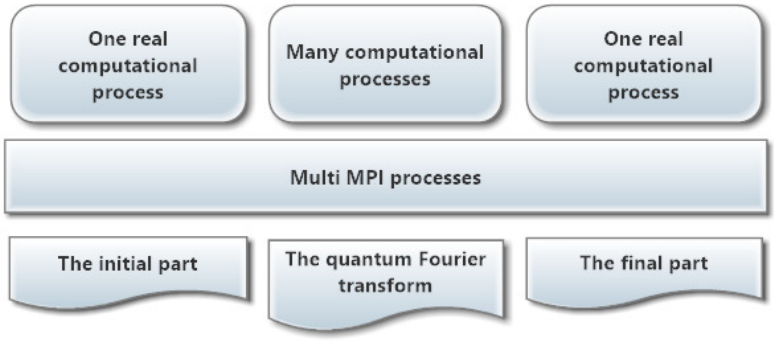


Rys. 7.2. Schemat ideowy zrównoleglenia algorytmu Shora w środowisku OpenMP. *Źródło: Wykonanie własne.*

7.3.2.2 Komputery z pamięcią rozproszoną

Idea wersji rozproszonej algorytmu Shora wykorzystującej MPI jest bardzo podobna do tej zaprezentowanej przy podejściu do OpenMP. Jedyna istotniejsza różnica polega na tym, że zainicjowane w tym przypadku procesy istnieją podczas wykonywania całego programu i nie są powoływane do życia *(potem usuwane)* tylko gdy istnieje potrzeba równoległych obliczeń. W przypadku gdy obliczenia są wykonywane sekwencyjnie to interesuje nas tylko proces zerowy *(faktycznie wtedy każdy z procesów wykonuje te same obliczenia)*, a do samego zrównoleglenia dochodzi w przypadku kwantowej transformaty Fouriera, gdzie każdy proces realizuje obliczenia dla wydzielonych fragmentów pętli. Schemat ideowy zrównoleglenia algorytmu

Shora w środowisku MPI przedstawiono na Rys. 7.3.



Rys. 7.3. Schemat ideowy zrównoleglenia algorytmu Shora w środowisku MPI. *Źródło: Wykonanie własne.*

7.3.2.3 Akceleratory graficzne - środowisko CUDA

Wersja funkcji realizującej kwantową transformatę Fouriera dla akceleratorów graficznych wspierających standard CUDA została przedstawiona na Listingu 23.

```
1  __global__ void dqft_cuda_kernel(complex_float q_register[], complex_float init↩
       [], unsigned long long int epsilon_table[], unsigned long long int q)
2  {
3
4      unsigned long long int idx;
5      idx = blockIdx.x * blockDim.x + threadIdx.x;
6
7      idx = epsilon_table[idx];
8
9      if (idx < q)
10     {
11
12         complex_float tmpcomp;
13         complex_float tmpcomp_2;
14
15         float tmp;
16         complex_float tmp2;
17
18         tmp2.real = q_register[idx].real;
19         tmp2.imag = q_register[idx].imag;
20
21         float term_1;
22         term_1 = __powf( (float) q, (float) -.5);
23
24         unsigned long long int c;
25
26         for (c = 0 ; c < q ; c++)
27         {
28             tmp = 2*PI*idx*c/q;
29
30             tmpcomp.real = term_1 * __cosf(tmp);
31             tmpcomp.imag = term_1 * __sinf(tmp);
32
```

```
33              tmpcomp_2.real = (tmp2.real * tmpcomp.real) - (tmp2.imag * tmpcomp.↩
                    imag);
34              tmpcomp_2.imag = (tmp2.imag * tmpcomp.real) - (tmp2.real * tmpcomp.↩
                    imag);
35
36              olib_atomic_add_cuda(&init[c].real, tmpcomp_2.real);
37              olib_atomic_add_cuda(&init[c].imag, tmpcomp_2.imag);
38
39          }
40
41      }
42
43  }
```

Listing 23. Implementacja kwantowej transformaty Fouriera w standardzie CUDA.

Jest ona całkowicie pozbawiona jednej pętli, a jedna instrukcja warunkowa *if*, została zastąpiona inną znajdującą się w innym miejscu funkcji. Pętla została usunięta, ponieważ w przypadku środowiska CUDA kolejne jej iteracje przejął podział na równoległe wątki. Instrukcja warunkowa *if ((pow(q_register[a].real, 2) + pow(q_register[a].imag, 2)) > epsilon)* została usunięta ze względu na optymalizację. Akceleratory graficzne słabo radzą sobie z tego typu rozgałęzieniami. Zamiast tego zastosowano tzw. tablicę przejść *(nazwa w kodzie: epsilon_table)*, która definiuje dla których indexów w tablicy będą wykonywane operacje.

Także nie wszystkie karty garficzne wspierają operacje atomowe na liczbach zmiennoprzecinkowych. W programie wykorzystano emulację programową tej instrukcji tak aby zachować zgodność ze wszystkimi kartami graficznymi. Operacje atomowe *(na pamięci globalnej)* na liczbach całkowitych o długości 32-bit wspierają wszystkie karty graficzne *(wymagane jest Compute capability 1.0)*, ale dla liczb zmiennoprzecinkowych już tylko karty z Compute capability większym bądź równym 2.0. Funkcję emulującą programowo operację atomowego dodawania zaprezentowano na Listingu 24.

```
1  __device__ inline void olib_atomic_add_cuda(float *address, float value)
2  {
3      int oldval, newval, readback;
4
5      oldval = __float_as_int(*address);
6      newval = __float_as_int(__int_as_float(oldval) + value);
7      while ((readback=atomicCAS((int *)address, oldval, newval)) != oldval)
8      {
9          oldval = readback;
10         newval = __float_as_int(__int_as_float(oldval) + value);
11     }
12 }
```

Listing 24. Implementacja funkcji emulującej operację atomowego dodawania dla urządzeń nie posiadających sprzętowego wsparcia dla tej operacji w standardzie CUDA.

7.3.2.4 Pozostałe urządzenia - środowisko OpenCL

Kernel napisany w standardzie OpenCL jest analogiczny dla tego w CUDA. Dużym problemem może być konieczność zastosowania emulacji operacji atomowej *(tak samo jak w przypadku CUDA)*, której kod został przedstawiony na Listingu 25.

```
1  float atom_add_float(__global float* const address, const float value)
2  {
3          uint oldval, newval, readback;
4
5          *(float*)&oldval = *address;
6          *(float*)&newval = (*(float*)&oldval + value);
7          while ((readback = atom_cmpxchg((__global uint*)address, oldval, newval))↩
               != oldval)
8          {
9                  oldval = readback;
10                 *(float*)&newval = (*(float*)&oldval + value);
11         }
12
13         return *(float*)&oldval;
14 }
```

Listing 25. Implementacja funkcji emulującej operację atomowego dodawania dla urządzeń nie posiadających sprzętowego wsparcia dla tej operacji w standardzie OpenCL.

Funkcja w przeciwieństwie do swojego odpowiednika w CUDA wykorzystuje operację OpenCL *atom_cmpxchg() (wcześniej była to atomicCAS()).*

7.3.3. Obliczenia rozproszone

Implementacja algorytmu w wersji na platformę rozproszoną BOINC nie będzie znacząco różnić się od tej przeznaczonej do obliczeń sekwencyjnych. Główną różnicą jest konieczność zastosowania bibliotek i API BOINC. W skład podstawowego pakietu funkcji wchodzą:

- boinc_init(),
- boinc_fraction_done(),
- boinc_time_to_checkpoint(float),
- boinc_checkpoint_completed(),
- boinc_finish().

Program dodany do platformy BOINC ma także nieco odmienną strukturę. Wydzielono w nim funkcję główną *shor_algorithm(unsigned long long int, unsigned long*

long int). Funkcja ta jako pierwszy argument przyjmuje liczbę, która ma zostać poddana procesowi faktoryzacji, a jako drugi ilość prób, które będą wykonane w przypadku niepomyślnego rozkładu na czynniki pierwsze. Próby te będą wykonywane aż do momentu odnalezienia prawidłowego rozwiązania lub przekroczenia możliwej ilości testów.

Oprogramowanie dane wejściowe pobiera z parametrów przekazanych podczas uruchamiania. Pierwszy parametr określa liczbę podlegającą faktoryzacji, drugi maksymalną ilość prób w jednym podejściu *(jest to wartość drugiego argumentu funkcji shor_algorithm())*, a trzeci ilość podejść do procesu faktoryzacji *(ilość wywołań funkcji shor_algorithm())*.

Z wykorzystaniem BOINC uruchamiano zadania posiadające jako parametry 511, 5 i 25. Oznacza to, że rozkładowi na czynniki pierwsze podlegała liczba 511.

Na samym początku programu jest wywoływana funkcja *boinc_init()*, która ma na celu zainicjować pracę aplikacji. W przypadku niepomyślnej inicjalizacji funkcja ta zwraca zero. Wyjątek ten jest następnie obsługiwany, wyświetlana jest odpowiednia informacja na wyjściu kontrolnym i kończymy działanie programu zwracając przez funkcję główną *main()* wartość zwróconą przez *boinc_init()*. Odpowiedni kod prezentujący zastosowanie *boinc_init()* został przedstawiony na Listingu 26.

```
1     int rc = boinc_init();
2     if (rc)
3     {
4         fprintf(stderr, "APP: boinc_init() failed. rc=%d\n", rc);
5         exit(rc);
6     }
```

Listing 26. Inicjacja środowiska BOINC API.

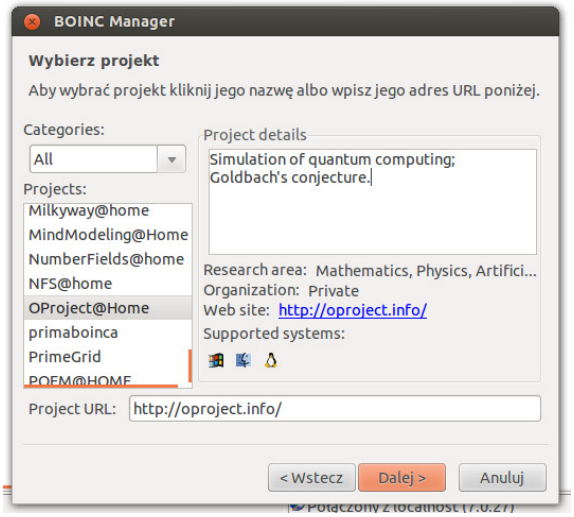
Funkcja *boinc_fraction_done(float)* aktualizuje stan paska postępu obliczeń w programie klienckim BOINC Manager. Jako argument przyjmuje ona ułamek określający postęp prac. Przykładowo dla zakończenia *75%* programu będzie to *0.75*. Funkcja ta została umieszczona w pętli, w której obsługiwane są kolejne wywoływania *shor_algorithm()*. Dzięki temu dysponując numerem aktualnego wywołania funkcji *shor_algorithm()* oraz maksymalną ich ilością możemy łatwo obliczyć stan procesu obliczeń. Wycinek kodu odpowiedzialnego za realizację aktualizacji paska postępu został przedstawiony na Listingu 27.

```
1 while(actual_iteration < number_of_iterations)
2 {
3      boinc_fraction_done(((double)actual_iteration)/number_of_iterations);
4      //...
5      shor_algorithm(n, max_tests);
6      //...
```

```
7 }
```

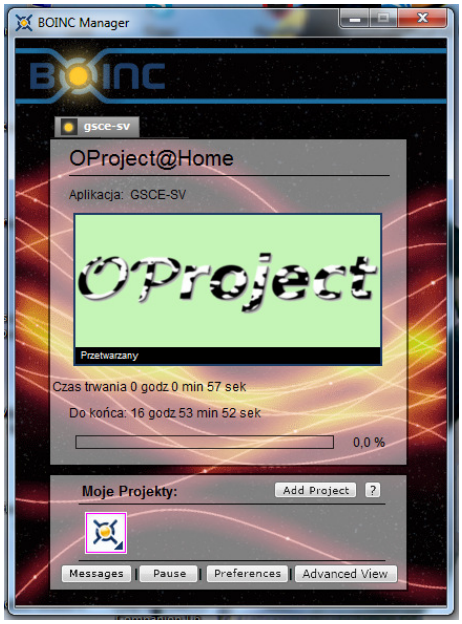
Listing 27. Realizacja aktualizacji paska postępu za pomocą BOINC API.

Okno BOINC Managera'a podczas wyboru projektu OProject@Home przedstawiono na Rys. 7.4.



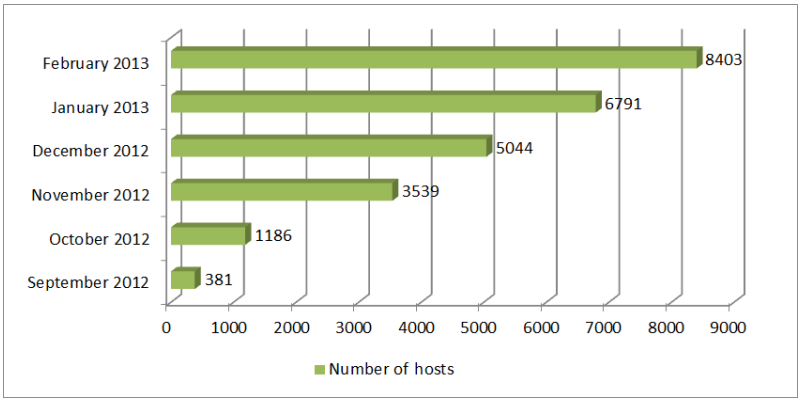
Rys. 7.4. Okno BOINC Manager'a w środowisku Linux podczas wyboru projektu OProject@Home. *Źródło: Wykonanie własne.*

Uproszczony wygląd BOINC Manager'a podczas przetwarzania danych zaprezentowano na Rys. 7.5.

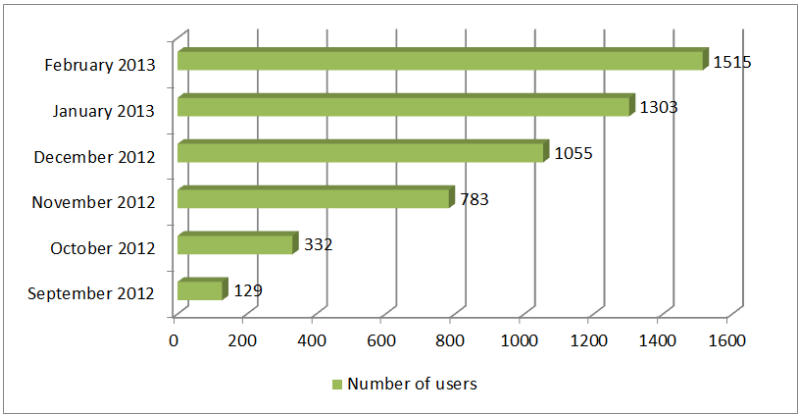


Rys. 7.5. Uproszczony wygląd BOINC Manager'a w środowisku Windows podczas przetwarzania danych w projekcie OProject@Home. *Źródło: Wykonanie własne.*

Na Rys. 7.6 oraz Rys. 7.7 przedstawiono wzrost ilości użytkowników i komputerów podłączonych do platformy OProject@Home w okresie od września 2012 roku do lutego 2013 roku.



Rys. 7.6. Wzrost ilości komputerów w systemie OProject@Home *(uwzględnione tylko komputery, które zwróciły przynajmniej jedno obliczone zadanie). Źródło: [252].*



Rys. 7.7. Wzrost ilości użytkowników w systemie OProject@Home. *(uwzględnieni tylko użytkownicy, którzy zwrócili przynajmniej jedno obliczone zadanie) Źródło: [252].*

Na dzień 4 lutego 2012 roku OProject@Home to 5391 użytkowników z 69 krajów w 1805 zespołach. W systemie zarejestrowano 10064 komputerów *(61070 rdzeni CPU i 5682 kart graficznych)*. Do dnia 4 lutego całkowity czas obliczeń na wszystkich procesorach wyniósł ponad 158 lat.

8. Wyniki końcowe

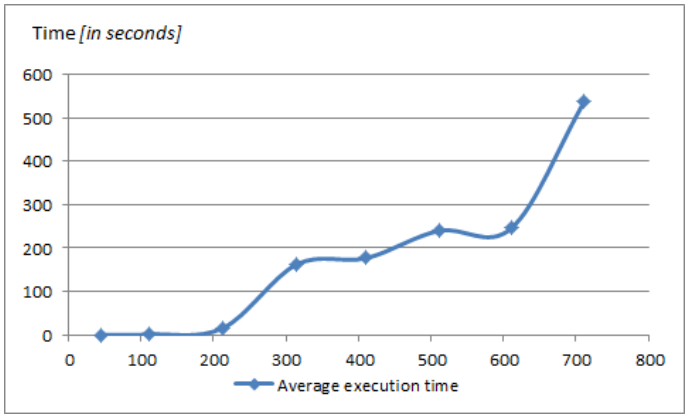
W wynikach końcowych zaprezentuję przedstawienie możliwych do uzyskania przyśpieszeń, porównanie wydajności środowiska CUDA z OpenCL oraz dynamiczną analizę wykorzystania czasu procesora i pamięci. Jeden podrozdział został także poświęcony na porównie wydajności systemów operacyjnych FreeBSD i Linux.

8.1. Przyśpieszenie uzyskane dzięki przetwarzaniu równoległemu

Wyniki uzyskane dzięki programowaniu równoległemu są różne. Najwyższe przyśpieszenie uzyskano dzięki zastosowaniu klastra komputerowego AGH *'Zeus'* i wynosiło ono *58.23*.

8.1.1. Algorytm Shora

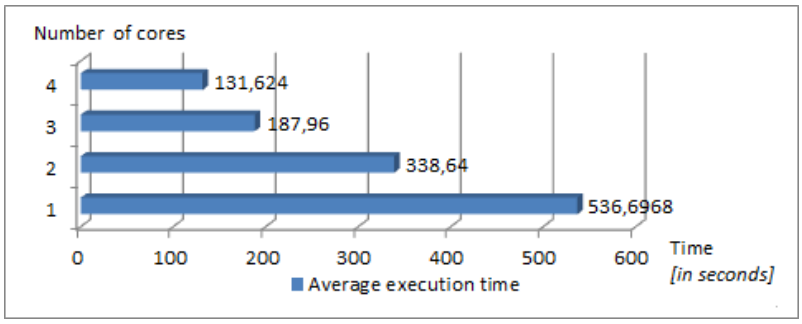
Wzrost czasu relizacji algorytma Shora dla różnych liczb wejściowych na procesorze Intel Xeon E3 1225 zaprezentowano na Rys. 8.1.



Rys. 8.1. Wzrost czasu realizacji algorytmu Shora wraz ze wzrostem rozmiaru liczby wejściowej na procesorze Intel Xeon E3 1225. *Źródło: Wykonanie własne.*

Średnio wykonanie faktoryzacji liczby 611 zajęło 245.868 sekund. W przypadku liczby 711 czas ten wzrośnie o dodatkowe 290.8288 sekund do wartości 536.6968.

Wynik zrównoleglenia za pomocą środowiska OpenMP i z wykorzystaniem procesora Intel Xeon E3 1225 zaprezentowano na Rys. 8.2.



Rys. 8.2. Czas realizacji algorytmu Shora dla liczby 711 i różnej ilości rdzeni procesora Intel Xeon E3 1225. *Źródło: Wykonanie własne.*

Przy wykorzystaniu jednego rdzenia czas faktoryzacji liczby 711 wyniósł 536.6968 sekund. Dzięki zastosowaniu programowania równoległego i czterech rdzeni czas ten spadł do 131.624 sekund. Uzyskano więc w pomiarach przyśpieszenie równe *4.077*. Przyśpieszenie nie powinno przekroczyć progu 4.0 *(wykorzystaliśmy jedynie cztery rdzenie procesora)* jednak w tym przypadku został uzyskany taki rezultat ze względu na losowy charakter czasu wykonywania. Średni czas symulacji we wszystkich analizach przeprowadzonych w tym rozdziale dotyczy wartości obliczonej na podstawie 250 symulacji.

Wyniki czasowe dla akceleratora graficznego nVidia Tesla C2050 i środowisk OpenCL oraz CUDA przedstawiono w Tabeli 8.1.

Tab. 8.1. Czasy wykonywania faktoryzacji określonych liczb za pomocą symulacji algorytmu Shora z wykorzystaniem OpenCL i CUDA oraz akceleratora graficznego nVidia Tesla C2050. *Źródło: Wykonanie własne.*

	OpenCL			CUDA		
Number	Min	Avg	Max	Min	Avg	Max
45	3 s	3.864 s	5 s	5.54 s	5.703 s	6.12 s
111	3 s	4.856 s	13 s	5.53 s	6.105 s	9.98 s
213	3 s	9.276 s	112 s	5.57 s	7.957 s	63.49 s
315	17 s	58.428 s	381 s	8.74 s	23.415 s	199.15 s
411	4 s	26.656 s	1420 s	5.71 s	12.708 s	691.36 s
511	5 s	37.896 s	1418 s	6.32 s	23.023 s	715.75 s
611	4 s	27.896 s	1413 s	5.81 s	24.597 s	2510.74 s
711	3 s	81.068 s	5134 s	7.12 s	24.568 s	832.80 s

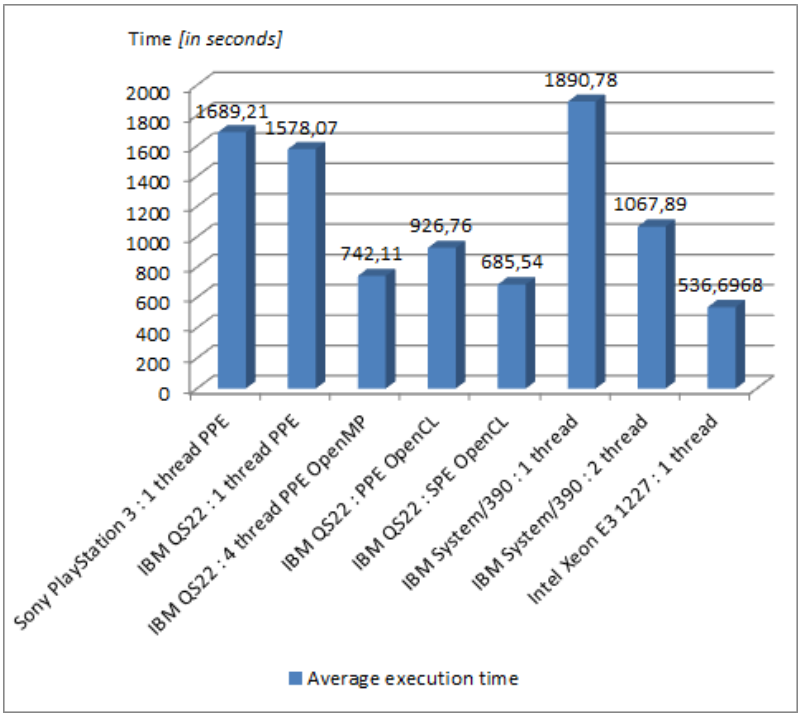
W przypadku środowiska OpenCL do pomiaru czasu zastosowano funkcję *time()* przez co uzyskano precyzję jedynie co do jednej sekundy. Natomiast w Tabeli 8.2 zaprezentowano uzyskane rezultaty dla kilku akceleratorów graficznych.

Tab. 8.2. Czasy wykonywania faktoryzacji liczby 711 za pomocą symulacji algorytmu Shora z wykorzystaniem OpenCL *(dla AMD)* i CUDA *(dla nVidii)*. *Źródło: Wykonanie własne.*

	Min	Avg	Max
nVdia Tesla C2050	7.12 s	24.568 s	832.80 s
nVidia Tesla C1060	8.08 s	26.342 s	976.50 s
nVidia GeForce GTS250	12.67 s	58.786 s	1864.32 s
nVidia GeForce 9800GT	58.09 s	276.895 s	2654.44 s
nVidia GeForce 9800GX2	60.84 s	290.438 s	2743.01 s
AMD Radeon 7770	5.54 s	25.875 s	743.76 s

Zastosowane implementacje umożliwiły uzyskanie znacznego przyśpieszenia jedynie dla akceleratorów nVidia Tesla C2050, C1060 oraz AMD Radeon 7770. W przypadku tych kart przyśpiesznie w porównaniu do pełnych czterech rdzeni procesora Intel Xeon E3 1225 wyniosło odpowiednio *5.36*, *5.00* oraz *5.09*. W przypadku gorszych kart graficznych klasy GeForce 9800 nie uzyskano satysfakcjonujących wyników. Tak słabe wyniki osiągnięto prawdopodobnie ze względu na zastosowanie emulacji funkcji atomowego dodawania.

Wyniki czasowe realizacji algorytmu Shora dla konsoli PS3, IBM Blade QS22 oraz IBM System/390 zaprezentowano na Rys. 8.3.



Rys. 8.3. Czas realizacji algorytmu Shora dla liczby 711 i konsoli PlayStation 3, IBM Blade QS22 oraz IBM System/390. *Źródło: Wykonanie własne.*

Na konsoli PlayStation wykonano tylko test wydajności na jednym wątku wy-

konywanym na PPE. Uzyskana w ten sposób wydajność nie różni się znacząco od tej dla jednego wątka PPE wykonywanego na procesorze zainstalowanym w IBM Blade QS22. Po zastosowaniu środowiska OpenMP i czterech wątków na IBM Blade udało się uzyskać przyśpieszenie równe *2.12*. Jest to dobry wynik ponieważ mimo, że posiadamy w tym przypadku cztery wątki to są one realizowane fizycznie przez dwie jednostki obliczeniowe posiadające technologię Hyper-Threading. Dla PPE środowisko OpenCL okazało się o ok. *18%* wolniejsze od OpenMP. Procesory SPE okazały się być tylko nieznacząco szybsze od PPE mimo, że dzieli je duża różnica pod względem wydajności teoretycznej *(na korzyść SPE)*. Nie zastosowano wykorzystania jednocześnie SPE i PPE ze względu na konieczność użycia dodatkowych technik programowania OpenCL. Na platformie IBM System/390 były dostępne jedynie dwa fizyczne procesory *(wydzielona maszyna wirtualna)*. Na dwóch procesorach udało się uzyskać przyśpieszenie równe *1.77* względem pojedynczego CPU IBM System/390.

Na klastrze komputerowym AGH *'Zeus'* udostępnionym w ramach projektu PL-Grid wykonano najbardziej zaawansowaną symulację z wykorzystaniem środowiska MPI i 64 rdzeni procesora Intel Xeon X5650[65]. Wyniki przedstawiono w Tabeli 8.3.

Tab. 8.3. Wynik faktoryzacji liczby 711 uzyskany dla środowiska MPI i superkomputera AGH *'Zeus'*. *Źródło: Wykonanie własne.*

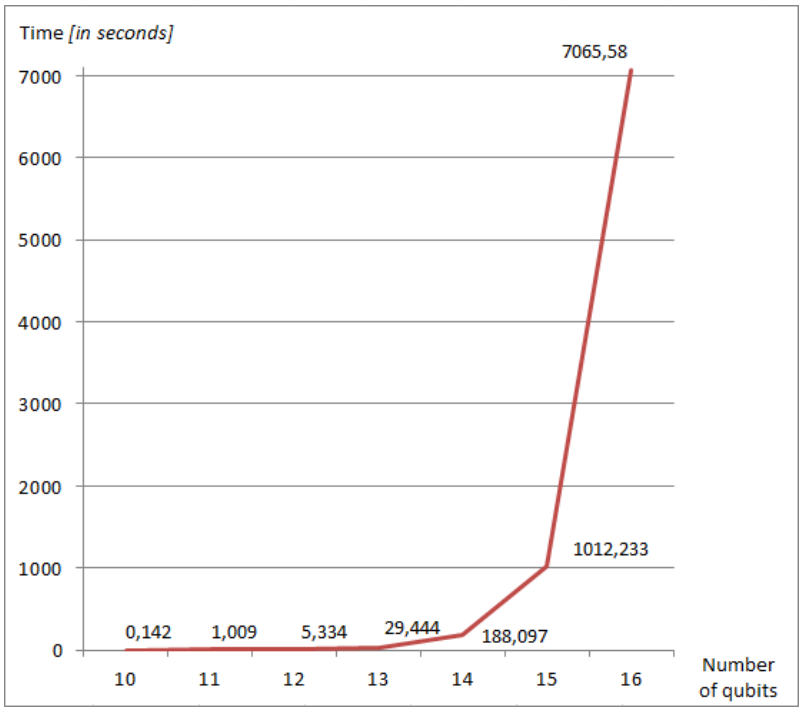
	1 process	64 MPI processes	Speedup
Intel Xeon X5650	498.453 s	8.56 s	58.23

Udało się uzykać przyśpieszenie wynoszące aż *58.23*.

8.1.2. Algorytm Grovera

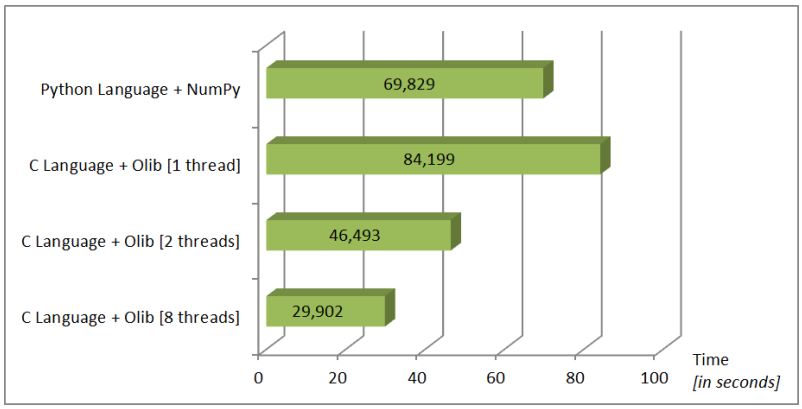
Symulacja algorytmu Grovera ma złożoność wykładniczą, co potwierdzają dane empiryczne, które zostały zaprezentowane na Rys. 8.4.

[65] Wykorzystana topologia podczas wykonywania zadania: osiem węzłów po osiem rdzeni w każdym z nich.



Rys. 8.4. Wzrost czasu przetwarzania sekwencyjnego algorytmu Grovera na procesorze Intel Xeon E7-4860 przy różnym rozmiarze rejestru kwantowego. *Źródło: [253].*

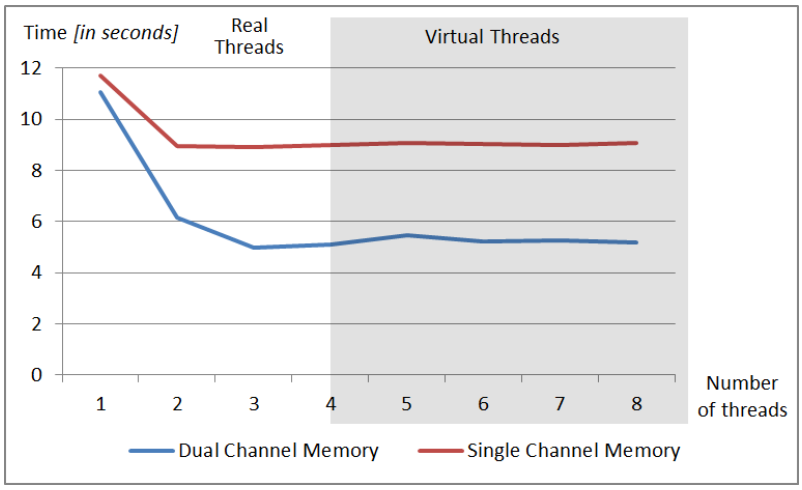
Wydajność rozwiązania napisanego w języku C w porównaniu do języka Python i biblioteki NumPy przedstawiono na Rys. 8.5.



Rys. 8.5. Porównanie wydajności równoległych funkcji biblioteki OLib z NumPy (test dla 14 kubitów i procesora Intel Core i7 920). *Źródło: [253].*

Jak się okazało program w Pythonie na pojedynczym rdzeniu wykonywał się znacznie szybciej. Biblioteka NumPy nie umożliwia jednak zrównoleglenia obliczeń.

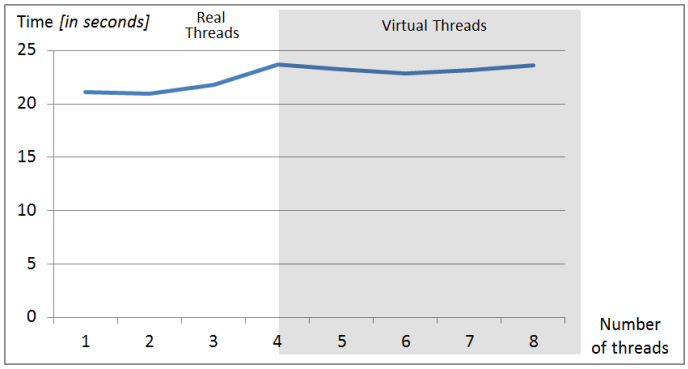
W przypadku algorytmu Grovera testy wydajnościowe przeprowadzono na czterech różnych platformach opartych o procesory Intel Core 2 Quad Q8400, Intel Core i5-2400, Intel Core i7 920 i systemie bazującym na czterech procesorach Intel Xeon E7-4860. Na Rys. 8.6 zaprezentowano wyniki czasowe dla symulacji układu 13 kubitów z wykorzystaniem procesora Intel Core i5-2400.



Rys. 8.6. Zmiana czasu realizacji symulacji układu 13 kubitów dla różnej ilości wątków procesora Intel Core i5-2400 i pamięci jedno lub dwukanałowej DDR3. *Źródło: [253].*

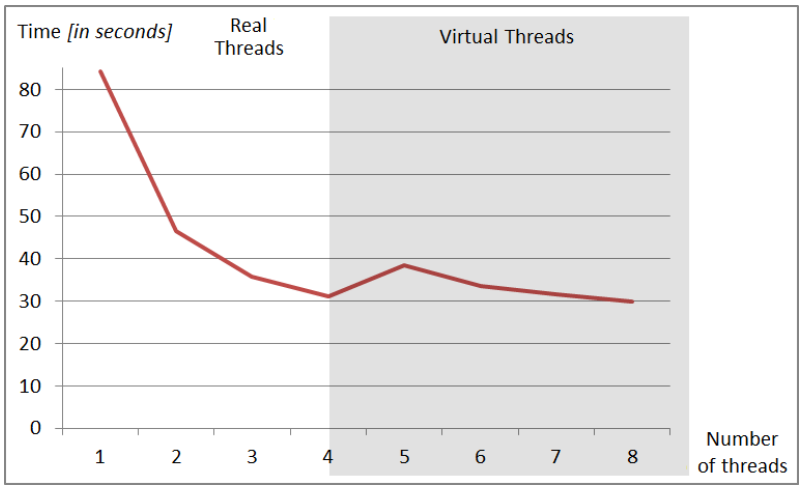
Symulacja ta do wykonania wymagała minimum 1.5 GB pamięci operacyjnej RAM. Widoczne są różnice przy porównywaniu pamięci jedno i dwukanałowej DDR3. Pamięć działająca w trybie dwukanałowym przy częstotliwości 1333 MHz posiada maksymalną przepustowość 21.2 GB/s. Tego typu pamięć jednokanałowa charakteryzuje się przepustowością rzędu 10.6 GB/s. Różnica w transferze pamięci RAM gra kluczową rolę podczas wykonywania tego algorytmu. Najmniejszy czas realizacji dla tego procesora przy pamięci dwukanałowej wynosi 5.107 sekundy, a dla pamięci jednokanałowej już 8.916 sekundy.

Pomimo zastosowania programowania równoległego nie zauważono znacznego przyśpieszenia wydajności na platformie opartej o procesor Intel Core 2 Quad Q8400. Testowana platforma wykorzystywała jedynie jednokanałową pamięć DDR3 o przepustowości 10.6 GB/s. Ponadto procesor Core 2 Quad nie posiada zintegrowanego kontrolera pamięci, co prawdopodobnie dodatkowo obniża wydajność układu. Wynik dla procesora Intel Core 2 Quad został przedstawiony na Rys. 8.7.



Rys. 8.7. Zmiana czasu realizacji symulacji układu 13 kubitów dla różnej ilości wątków procesora Intel Core 2 Quad Q8400. *Źródło: [253].*

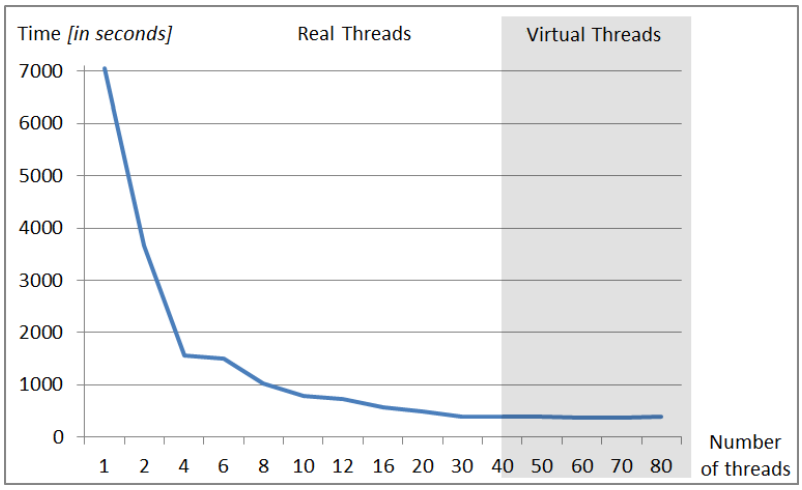
Wyniki testu innej platformy bazującej na Intel Core i7 920 zaprezentowano na Rys. 8.8.



Rys. 8.8. Zmiana czasu realizacji symulacji układu 14 kubitów dla różnej ilości wątków procesora Intel Core i7 920. *Źródło: [253].*

Platforma ta wykorzystuje trójkanałową pamięć DDR3 1066 MHz umożliwiającą osiągnięcie przepustowości na poziomie 25.6 GB/s. W tym przypadku przyśpieszenie nie musi ograniczać się jedynie do czterech fizycznych rdzeni, ponieważ ten procesor obsługuje technologię Hyper-Threading. Przy wykorzystaniu czterech wątków osiągnięto czas wykonywania algorytmu 31.049 sekundy, a dla pełnego wykorzystania technologii HT czas ten spadł do 29.902 sekund. Wcześniejsze testy dla platform Core 2 Quad Q8400 (rysunek 8.7) i Intel Core i5-2400 (rysunek 8.6) wykonano dla maksymalnie ośmu wątków, pomimo że platformy te posiadały fizycznie jedynie cztery rdzenie i nie wspierały technologii HT. Dodatkowe wątki na rysunkach zostały oznaczone jako *Virtual Threads*.

Najbardziej zaawansowana platforma wykorzystywała cztery procesory Intel Xeon E7-4860. Procesory te wspierają technologię HT dzięki czemu tego typu system dysponuje możliwością równoległej realizacji aż 80 wątków. Najniższy czas realizacji algorytmu uzyskano dla 30 wątków i wynosił on 384.308 sekund. Program wykonywany sekwencyjnie potrzebował na wykonanie wszystkich obliczeń aż 7065.580 sekund co daje przyśpieszenie ok. 18.385. Redukcja czasu wykonywania dla różnej ilości wątków i tej platformy zaprezentowano na Rys. 8.9.

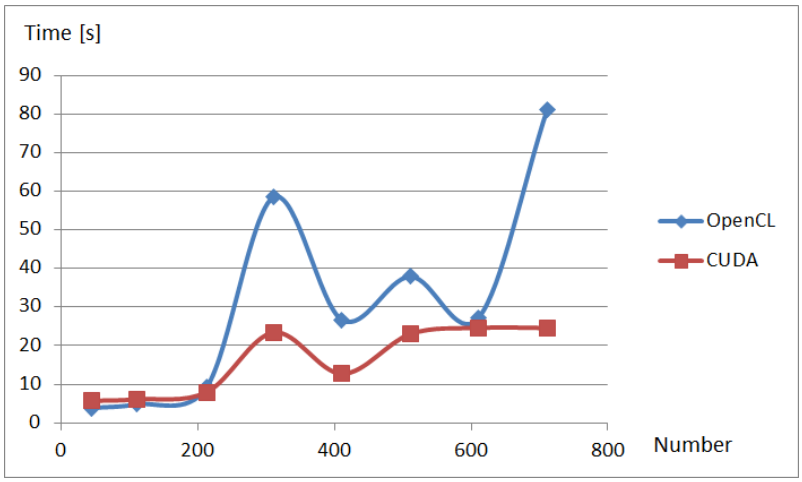


Rys. 8.9. Spadek czasu realizacji symulacji układu 16 kubitów dla platformy złożonej z czterech procesorów Intel Xeon E7-4860. *Źródło: [253].*

Platforma ta dysponowała aż 256 GB pamięci operacyjnej RAM. Do przeprowadzenia symulacji na układzie 16 kubitów potrzeba było aż ok. 96 GB wolnej przestrzeni pamięci RAM.

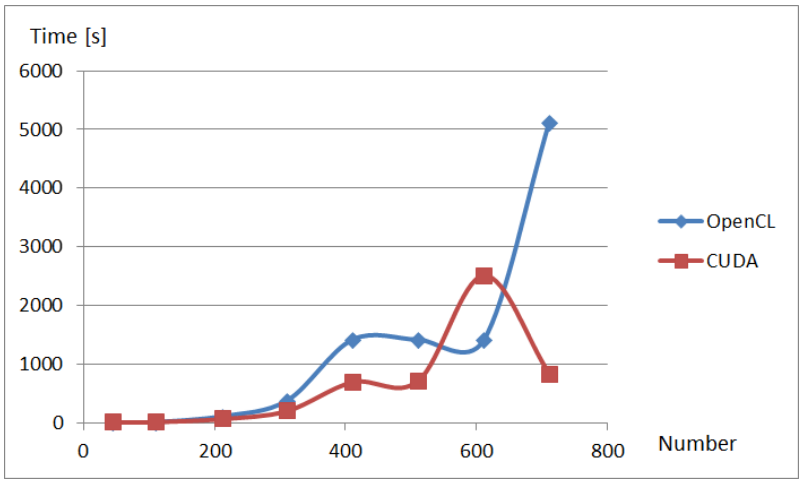
8.2. CUDA vs OpenCL - Algorytm Shora

Implementacje napisane w środowisku OpenCL można bardzo łatwo porównać z tymi napisanymi w standardzie CUDA. Analizie wystarczy poddać Tabelę 8.1. Gdy będziemy porównywać średnie wyniki czasowe dla OpenCL i CUDA zaobserwujemy, że zawsze lepsze rezultaty uzyskano za pomocą tego drugiego standardu. OpenCL okazał się w tym przypadku szybszy tylko podczas faktoryzowania najmniejszych liczb: 45 oraz 111. Tego typu ujęcie zostało zaprezentowane na Rys. 8.10.



Rys. 8.10. Porównanie czasów wykonywania programów napisanych w CUDA i OpenCL *(czasy średnie dla 250 symulacji). Źródło: Własne.*

Do podobnych wniosków dojdziemy także analizując Rys. 8.11, który przedstawia zestawienie dla maksymalnych czasów wykonywania symulacji.



Rys. 8.11. Porównanie czasów wykonywania programów napisanych w CUDA i OpenCL *(czasy maksymalne dla 250 symulacji). Źródło: Własne.*

8.3. Porównanie wydajności systemów operacyjnych

Przeprowadzone w tym rozdziale testy wykonano na platformie opartej o procesor Intel Atom N2800. Podczas pomiarów wykorzystano funkcję systemową *time()* zwracającą trzy czasy: *real*, *user* i *sys*. W tym podrozdziale głównie będziemy skupiać się na czasie systemowym[66] i czasie użytkownika[67].

Testy wykonano na ośmiu systemach operacyjnych: FreeBSD 7.4, FreeBSD 8.3, FreeBSD 9.0, Debian 6.0, Fedora 17, Slackware 13, OpenSUSE 11 oraz Ubuntu 10.04.

Czasy użytkownika dla systemów operacyjnych z rodziny FreeBSD przedstawiono w Tabeli 8.4.

Tab. 8.4. Czasy użytkownika dla symulacji faktoryzacji liczby 213 *(z pominięciem instrukcji warunkowej)* dla różnych wersji systemu operacyjnego FreeBSD.

Threads	FreeBSD 7.4	FreeBSD 8.3	FreeBSD 9.0
1	2184.6025 s	2186 s	2184.4525 s
2	1237.075 s	1180.6337 s	1151.1312 s
4	641.9762 s	641.9637 s	642.7562 s
...			
1024	663.95 s	664.0087 s	663.8737 s

[66] Czas poświęcony przez procesor na komunikację z jądrem systemu *(tzw. praca ńa zewnątrz procesu")*.

[67] Czas poświęcony na pracę procesora *"wewnątrz procesu"(tzw. user-mode)*.

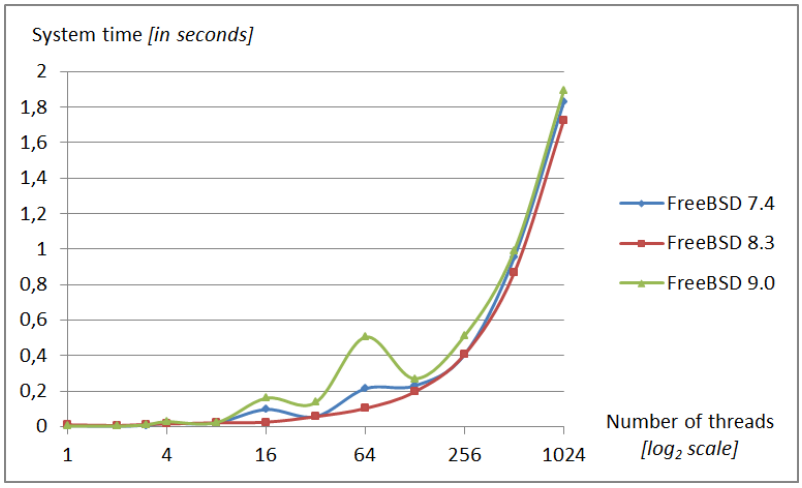
Analogiczną tabelą dla różnych dystrybucji systemów operacyjnych Linux jest 8.5.

Tab. 8.5. Czasy użytkownika dla symulacji faktoryzacji liczby 213 *(z pominięciem instrukcji warunkowej)* dla różnych dystrybucji systemu operacyjnego Linux.

Threads	Debian 6.0	Fedora 17	Slackware 13	OpenSUSE 11	Ubuntu 10.04
1	1498.1335 s	1498.2546 s	1498.099 s	1498.2548 s	1501.6127 s
2	752.8928 s	753.3881 s	752.8938 s	753.1166 s	755.4858 s
4	423.6761 s	424.213 s	424.06437 s	423.9758 s	424.8716 s
...					
1024	434.0202 s	435.772 s	433.9232 s	434.0218 s	434.2227 s

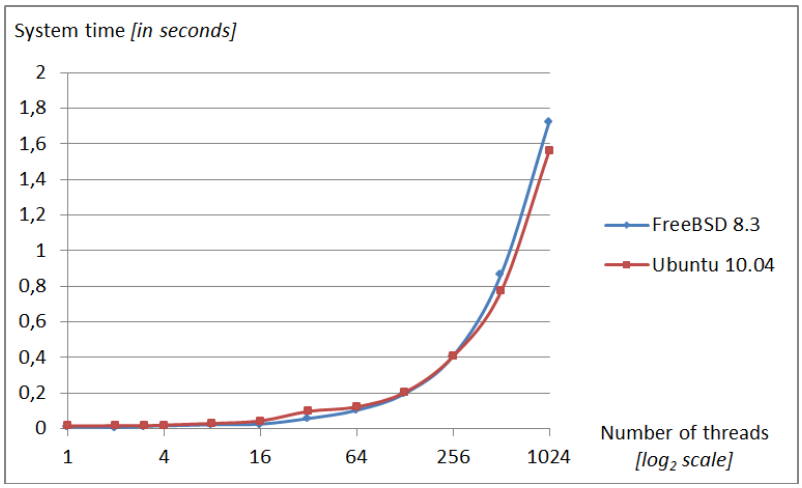
Można zauważyć, że programy testowane na Linux'ie wykonywały się o ok. *30%* szybciej niż na systemach klasy FreeBSD. Spowodowane jest to najprawdopodobniej tym, że FreeBSD posiada znacznie więcej funkcji odpowiedzialnych za bezpieczeństwo systemu[68]. Widoczny jest także zauważalny wzrost czasu wykonywania w przypadku wzrostu ilości wątków z czterech do 1024. Świadczy to głównie o wzrastającym wraz z ilością wątków i coraz większym nakładzie operacji na ich obsługę. Dla przykładu na systemie operacyjnym Debian 6.0 i czterech instacjach wykonawczych czas użytkownika wynosił 423,6761 sekund. Przy 1024 wątkach czas ten wzrósł do 434,0202 s, czyli o ok. *2,5%*.

W przypadku czasów systemowych nie ma większych różnic pomiędzy wszystkimi systemami operacyjnymi. Przypadek ten prezentują Rys. 8.12 oraz 8.13.



Rys. 8.12. Zmiany czasu systemowego dla algorytmu Shora *(liczba 213)* realizowane przez różną ilość wątków i systemów operacyjnych FreeBSD. *Źródło: Wykonanie własne.*

[68] Także jeżeli chodzi o moduły wkompilowane w jądro.



Rys. 8.13. Porównanie czasów systemowych dla algorutmu Shora *(liczba 213)* i systemów operacyjnych FreeBSD 8.3 oraz Ubuntu 10.04. *Źródło: Wykonanie własne.*

Bardzo widoczny jest jednak wzrost czasu systemowego wraz z wzrostem ilości obsługiwanych wątków.

8.4. Dynamiczna analiza wykorzystania czasu procesora

Do analizy wykorzystania czasu procesora posłużono się oprogramowaniem *gprof*. Gprof znajduje zastosowanie głównie w systemach UNIX'owych i jest następcą starszego programu *prof*. Umożliwia m. in. wyszczególnienie ile czasu spędził procesor na danej funkcji. W przypadku realizacji implementacji algorytmu Shora najwięcej czasu *(90.21%)* wydzielono dla realizacji kwantowej transformaty Fouriera. Dość istotnie zajmującą czas procesora funkcją jest także *sincosf()* razem z wywołaniami kerneli *__kernel_cosf* oraz *__kernel_sinf*. Dokładny wynik dla tego algorytmu przedstawiono na Listingu 28.

```
1  Each sample counts as 0.01 seconds.
2    %   cumulative   self              self     total
3   time   seconds   seconds    calls  ms/call  ms/call  name
4  90.21      4.22     4.22        1   190.00   240.00  dqft_sp()
5   2.45      4.32     0.14                             __kernel_cosf
6   2.11      4.42     0.10                             __kernel_sinf
7   1.70      4.45     0.08                             __ieee754_rem_pio2f
8   1.69      4.50     0.08                             sincosf
9   1.27      4.56     0.06                             floorf
10  1.27      4.62     0.06                             scalbnf
11  0.53      4.65     0.03  7454720     0.00     0.00  complex_add()
12  0.53      4.67     0.03  7454720     0.00     0.00  complex_mul()
13  0.42      4.69     0.02                             finite
14  0.21      4.70     0.01                             __ieee754_pow
15  0.21      4.71     0.01                             __profile_frequency
16  0.21      4.72     0.01                             brk
17  0.21      4.73     0.01                             rint
18  0.11      4.74     0.01        1     5.00   245.00  main
```

```
19   0.11      4.74     0.01                              complex_sub()
```

Listing 28. Wynik analizy implementacji algorytmu Shora dla liczby 111 przeprowadzonej za pomocą programu gprof.

Algorytm Grovera, jak wielokrotnie wspominano, wykorzystuje głównie funkcję mnożenia wektora przez macierz. W przypadku realizacji obliczeń dla 13 kubitów wywołanie tego typu funkcji zajmuje *90.57%*. Dokładne informacje dla algorytmu Grovera przedstawiono w Listingu 29.

```
1 Each sample counts as 0.01 seconds.
2   %   cumulative   self              self     total
3  time   seconds   seconds    calls   s/call   s/call  name
4  90.57     13.92  gprof  13.92      142     0.10     0.10  matrix_mul_vector()
5   9.17     15.33     1.41        1     1.41    15.33  main
6   0.10     15.35     0.02                             mmap
7   0.10     15.36     0.02                             munmap
8   0.03     15.37     0.01                             __sysconf_check_spec
9   0.03     15.37     0.01                             sysconf
```

Listing 29. Wynik analizy implementacji algorytmu Grovera dla 13 kubitów przeprowadzonej za pomocą programu gprof.

8.5. Dynamiczna analiza zarządzania pamięcią podręczną procesora

Jedną z najbardziej cennych informacji podczas analizy zarządzania pamięcią podręczną procesora jest współczynnik nietrafionych odczytów instrukcji lub danych *(ang. cache missing)*. Opisuje on jak często procesor nie mógł odnaleźć wymaganych danych w pamięci cache i musiał pobrać je z pamięci operacyjnej. Program *Valgrind* umożliwia przeprowadzenie takiej analizy. Aplikacja ta automatycznie wykrywa strukturę pamięci podręcznej procesora, na której jest uruchamiana. Na Listingu 30 przedstawiono strukturę cache dla procesora Intel Core 2 Quad Q8400.

```
1 desc: I1 cache:         32768 B, 64 B, 8-way associative
2 desc: D1 cache:         32768 B, 64 B, 8-way associative
3 desc: L2 cache:         2097152 B, 64 B, 8-way associative
```

Listing 30. Struktura pamięci cache wykryta przez Valgrind dla procesora Intel Core 2 Quad Q8400.

Procesor ten posiada 64 KB pamięci pierwszego poziomu i 2 MB pamięci drugiego poziomu[69].

[69] 2 MB dla jednego rdzenia. 4 MB dla dwurdzeniowego procesora

Na Listingu 31 przedstawiono wynik profilowania pamięci cache dla implementacji algorytmu Grovera.

```
1  ==9639== I    refs:      2,447,834,149
2  ==9639== I1   misses:            1,100
3  ==9639== L2i  misses:            1,081
4  ==9639== I1   miss rate:          0.00%
5  ==9639== L2i  miss rate:          0.00%
6  ==9639==
7  ==9639== D    refs:        918,170,157  (610,875,414 rd   + 307,294,743 wr)
8  ==9639== D1   misses:       21,257,561  ( 19,943,007 rd   +   1,314,554 wr)
9  ==9639== L2d  misses:       21,256,389  ( 19,942,092 rd   +   1,314,297 wr)
10 ==9639== D1   miss rate:           2.3% (        3.2%     +         0.4%   )
11 ==9639== L2d  miss rate:           2.3% (        3.2%     +         0.4%   )
12 ==9639==
13 ==9639== L2 refs:          21,258,661  ( 19,944,107 rd   +   1,314,554 wr)
14 ==9639== L2 misses:        21,257,470  ( 19,943,173 rd   +   1,314,297 wr)
15 ==9639== L2 miss rate:             0.6% (        0.6%     +         0.4%   )
```

Listing 31. Wynik profilowania pamięci cache dla implementacji algorytmu Grovera

Współczynnik chybień dla pamięci instrukcji pierwszego poziomu wynosi *0.00%*, a dla pamięci danych już *2.3%*. Współczynnik miss rate pamięci drugiego poziomu wynosi *0.6%*. Wzrost ilości wątków dzięki zastosowaniu środowiska OpenMP praktycznie nie zmienia w większy sposób tych współczynników.

Wynik działania Valgrind dla algorytmu Shora i faktoryzowanej liczby 45 został zaprezentowany na Listingu 32.

```
1  ==9756== I    refs:      2,145,900,412
2  ==9756== I1   misses:            1,109
3  ==9756== L2i  misses:            1,084
4  ==9756== I1   miss rate:          0.00%
5  ==9756== L2i  miss rate:          0.00%
6  ==9756==
7  ==9756== D    refs:        531,267,390  (365,400,340 rd   + 165,867,050 wr)
8  ==9756== D1   misses:            5,956  (      2,039 rd   +       3,917 wr)
9  ==9756== L2d  misses:            2,262  (        210 rd   +       2,052 wr)
10 ==9756== D1   miss rate:           0.0% (        0.0%     +         0.0%   )
11 ==9756== L2d  miss rate:           0.0% (        0.0%     +         0.0%   )
12 ==9756==
13 ==9756== L2 refs:              7,065  (      3,148 rd   +       3,917 wr)
14 ==9756== L2 misses:            3,346  (      1,294 rd   +       2,052 wr)
15 ==9756== L2 miss rate:             0.0% (        0.0%     +         0.0%   )
```

Listing 32. Wynik profilowania pamięci cache dla implementacji algorytmu Shora

W tym przypadku wszystkie współczynniki wynoszą ok. *0.00%* co oznacza prawie nie występowanie problemu *cache miss*. Jest tak ponieważ liczba 45 jest dość mała i większość danych wymaganych do pracy algorytmu mieści się w pamięci cache. W przypadku tego algorytmu praca nawet na stosunkowo dużych liczbach przyniesie w większym stopniu wydłużenie jedynie czasu wykorzystania procesora, przy wykorzystaniu dość małych obszarów pamięci.

9. Podsumowanie

Praca inżynierska okazała się prostsza niż przewidywano. Początkowe problemy sprawiał głównie automatyczny sposób kompilacji w systemie Linux bardzo skomplikowanych projektów. Wszystkie kody umieszczono w publicznym repozytorium SVN serwisu Google Code. Po pobraniu oprogramowania kompilacja wszystkich modułów powinna przebiegać bezproblemowo *(użytkownik musi posiadać zainstalowane wymagane pakiety)* i inicjowana jest wydaniem jednego polecenia.

Przedstawiono symulację układów kwantowych złożonych z maksymalnie kilkunastu *(algorytm Grovera)* lub kilkudziesięciu *(algorytm Shora)* kubitów. Sam rozwój techniki w znaczącym stopniu nie wpłynie na możliwość poprawy tych rezultatów. Uruchamiając zaprezentowany w tej pracy kod na najnowocześniejszych procesorach za kilka lat nie uzyska się dużo większych możliwości symulacji. Możliwości zwiększenia rozmiaru układu kwantowego, który możemy zasymulować dają modyfikacje kodu oprogramowania, a nie rozwój techniczny komputerów bazujących na technologii krzemowej.

Jednym z bardzo interesujących aspektów jest możliwość wykorzystania metod sztucznej inteligencji do projektowania algorytmów kwantowych. Można w tym miejscu zastosować algorytmy genetyczne i z ich pomocą próbować *wygenerować* odpowiedni algorytm kwantowy. Algorytmy genetyczne wykonują obliczenia na określonej populacji dzięki temu operacje wykonywane na poszczególnych osobnikach mogą być zrównoleglone [251]. Daje to doskonałe pole do wykorzystania programowania równoległego.

Literatura

[1] Lynch Nancy A. *Distributed Algorithms.* Morgan Kaufmann, ISBN 1-55860-348-4, 1996.

[2] N. Rosen A. Einstein, B. Podolsky. *Can Quantum-Mechanical Description of Physical Reality Be Considered Complete?.* Phys. Rev. 47.

[3] Scott Aaronson. *BQP and the polynomial hierarchy.* STOC '10 Proceedings of the 42nd ACM symposium on Theory of computing, 2010.

[4] Ricardo M. Matinata Abraham Arevalo et al. *Programming the Cell Broadband Engine Architecture Examples and Best Practices.* IBM, August 2008.

[5] Blumrich M.A. Adiga N.R. et al. *Blue Gene/L torus interconnection network.* IBM Journal of Research and Development, Volume: 49, Issue: 2.3, March 2005.

[6] G. Adiga N.R., Almasi et al. *An Overview of the BlueGene/L Supercomputer.* Supercomputing, ACM/IEEE 2002 Conference, 2002.

[7] William Gropp Al Geist et al. *MPI-2: Extending the message-passing interface.* Euro-Par'96 Parallel Processing, Lecture Notes in Computer Science, Volume 1123/1996, 128-135, DOI: $10.1007/3\text{-}540\text{-}61626\text{-}8_16, 1996$.

[8] Ralph Grishman Allan Gottlieb et al. *The NYU Ultracomputerâ"designing a MIMD, shared-memory parallel machine (Extended Abstract).* ISCA '82 Proceedings of the 9th annual symposium on Computer Architecture, April 1982.

[9] AMD. *APU 101: All about AMD Fusion Accelerated Processing Units.* AMD, 2011.

[10] AMD. *Southern Islands Series Instruction Set Architecture.* Reference Guide, August 2012.

[11] Jun an Yang. *Multi-universe parallel quantum genetic algorithm its application to blind-source separation.* Neural Networks and Signal Processing, 2003. Proceedings of the 2003 International Conference, 2003.

[12] E. R. Andrew. *Nuclear Magnetic Resonance.* Cambridge Monographs on Physics., 2009.

[13] Chinhyun Kim i Mitsuhisa Sato Andrew Sohn. *Multithreading with the EM-4 distributed-memory multiprocessor.* PACT '95 Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques, ISBN:0-89791-745-6, 1995.

[14] J.D. Ulman A.V. Aho, J.E. Hopcroft. *Projektowanie i analiza algorytmow komputerowych.* Wydawnictwo Naukowe PWN, 1983.

[15] M. Muldoon B. Korber et al. *Timing the Ancestor of the HIV-1 Pandemic Strains.* Science 9, Vol. 288 no. 5472, DOI: 10.1126/science.288.5472.1789, June 2000.

[16] Berg B.A. *Markov Chain Monte Carlo Simulations and Their Statistical Analysis.* World Scientific Publishing, 2004.

[17] Itzhak Bars. *Four-dimensionalM-theory and supersymmetry breaking.* Phys.Rev. D55, 1997.

[18] Casey J. Baud J.-P. et al. *Performance analysis of a file catalog for the LHC computing grid.* High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium, July 2005.

[19] R. vanderPas B.Chapman, G. Jost et al. *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, October 31, 2007.

[20] J. S. Bell. *On the Einstein-Podolsky-Rosen paradox.* Physics 1, 1964.

[21] Michel Le Bellac. *Wstep do informatyki kwantowej.* Wydawnictwo Naukowe PWN, Warszawa 2012.

[22] Peter Virnau i Tobias Preis Benjamin Block. *Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model.* Computer Physics Communications, Volume 181, Issue 9, 2010.

[23] Aftosmis M. Biswas R. et al. *Petascale computing: Impact on future NASA missions.* Petascale computing. Algorithms and applications, Chapman & HAll/CRC, Boca Raton, FL, 2008.

[24] Shekhar Borkar. *Thousand core chips: a technology perspective.* DAC '07 Proceedings of the 44th annual Design Automation Conference, 05 2011.

[25] Steinman M. Branover A., Foley D. *AMD Fusion APU: Llano.* Micro, IEEE, Volume: 32, Issue: 2, March-April 2012.

[26] A Brent R., Cleary et al. *Implementation and performance of scalable scientific library subroutines on Fujitsu's VPP500 parallel-vector supercomputer.* Scalable High-Performance Computing Conference, May 1994.

[27] Knudson B. Budnik T. et al. *Blue Gene/Q resource management architecture.* Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop, November 2010.

[28] Branson K Buyya R. et al. *The virtual laboratory: a toolset to enable distributed molecular modelling for drug design on the world-wide grid.* Concurrency and Computation: Practice and Experience, 15, 1, January 2003.

[29] Rice M.D. Carmona E.A. *Modeling the serial and parallel fractions of a parallel algorithm.* Jurnal of Parallel and Distributed Computing, 13, 1991.

[30] Leiserson C.E. *Fat-trees - University networks for hardware-efficient supercomputing.* IEEE Transactions on Computers. Vol. C-34, October 1985.

[31] Rohit Chandra. *Parallel Programming In Openmp (Edition 1).* Elsevier Science ISBN-13: 9781558606715, October 2000.

[32] Stephen J. Chapman. *Fortran 90/95 for Scientists and Engineers.* ISBN-13: 978-0072825756, July 31, 2003.

[33] Zahi S. Abuhamdeh Charles E. Leiserson et al. *The network architecture of the Connection Machine CM-5 (extended abstract).* SPAA '92 Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, 1992.

[34] Timothy Boronczyk Christopher Negus. *CentOS Bible.* Wiley ISBN:047048165X 9780470481653, 2009.

[35] R. Cleve. *Fast parallel circuits for the quantum Fourier transform.* Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium, 2000.

[36] E. Kilgariff C.M. Wittenbrink et al. *Fermi GF100 GPU architecture.* IEEE Computer Society, 2011.

[37] CNET.com. *Óctopiler seeks to arm Cell programmers"http://news.cnet.com/Octopiler-seeks-to-arm-Cell-programmers/2100-1007*$_3$−6042132.*html.CNET.com*(*Ostatnidostep* : 10.09.2012), 2006.

[38] Francis S. Collins. *The Language of God. A Scientist Presents Evidence for Belief.* Free Press, New York - London - Toronto - Sydney, ISBN 0-7432-8639-1, 2006.

[39] Bitz M.L. Collins W.D. et al. *The community Climate System Model version 3 (CCSM3).* Journal of Climate, 19, 2006.

[40] President's Information Technology Committee. *Computational science: Ensuring America's competitiveness.* June 2005.

[41] Lawrence Berkeley National Laboratory Computational Research Division. *The Potential of the Cell Processor for Scientific Computing.* Association for Computing Machinery, CFâ^{TM}06, Ischia, Italy, 2006.

[42] Intenational Human Genome Sequencing Consortium. *Initial sequencing and analysis of the human genome.* Nature, 409, 15 February 2001.

[43] Martonosi M. Contreras G. *Power prediction for Intel XScaleÂ® processors using performance monitoring unit events.* Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium, August 2005.

[44] Convex Computer Corporation. *Exemplar architecture.* Richardson, TX, 1993.

[45] Singht J.P i Gupta A. Culler D.E. *Parallel computer architecture.* Morgan Kaufamann, San Francisco, CA, 1999.

[46] Zbigniew Czech. *Wprowadzenie do obliczen rownolegĹ‚ych.* Wydawnictwo Naukowe PWN, Warszawa 2010.

[47] R. Joza D. Deutsch. *Quantum computational networks.* Proceedings of the Royal Society of London, 1992.

[48] Bader D.A. *Petascale computing. Algorithms and applications, (Subsection 2.4) Chapman & HAll/CRC, Boca Raton, FL.* 2008.

[49] Takayuki Kanda Daisuke Sakamoto et al. *Android as a telecommunication medium with a human-like presence.* HRI '07 Proceedings of the ACM/IEEE international conference on Human-robot interaction, 2007.

[50] Michael O. Rabin Daniel Lehmann. *On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem.* POPL '81 Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1981.

[51] Ajith K. Illendula David A. Bader et al. *Using PRAM Algorithms on a Uniform-Memory-Access Shared-Memory Architecture.* Algorithm Engineering, Lecture Notes in Computer Science, Volume 2141/2001, 129-144, DOI: 10.1007/3-540-44688-$5_1 1, 2001$.

[52] Jaldhar Vyas David B. Harris, Benjamin Mako Hill. *Debian GNU/Linux 3.1 Bible.* John Wiley & Sons, ISBN: 978-0-7645-7644-7, August 2005.

[53] Jeff Cobb David P. Anderson et al. *SETI@home: an experiment in public-resource computing.* Communications of the ACM, Volume 45 Issue 11, 2002.

[54] Randall Davis David Watts et al. *IBM BladeCenter Products and Technology.* IBM, Feburary 2007.

[55] Skillicorn D.B. *A taxonomy for computer architectures.* IEEE Computer, 21, 1988.

[56] Daniel de Kok. *Slackware Linux Basics For Slackware Linux 12.0.* Daniel de Kok, 2008.

[57] Virginie Marion-Poty i Cyril Fonlupt Denis Robilliard. *Population Parallel GP on the G80 GPU.* Genetic Programming, Lecture Notes in Computer Science, Volume 4971/2008, 98-109, DOI: 10.1007/978-3-540-78671-9_9, 2008.

[58] Abdulla Bataineh Dennis Abts et al. *The BlackWidow High-Radix Clos Network.* SC '07 Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007.

[59] Ken Thompson Dennis M. Ritchie. *The UNIX time-sharing system.* Communications of the ACM, Volume 17 Issue 7, July 1974.

[60] David Deutsch. *The Beginning of Infinity.* Allen Lane, 2011.

[61] IBM developerWorks. *Ćell Broadband Engine Architecture and its first implementation"http://www.ibm.com/developerworks/power/library/pa-cellperf/.* IBM developerWorks (Ostatni dostep: 10.09.2012), 2005.

[62] Bryce Seligman DeWitt. *The Many-Universes Interpretation of Quantum Mechanics.* Proceedings of the International School of Physics 'Enrico Fermi' Course IL: Foundations of Quantum Mechanics, Academic Press, 1972.

[63] PAM Dirac. *A new notation for quantum mechanics.* Mathematical Proceedings of the Cambridge Philosophical Society 35, 1939.

[64] Christian Terbovenand Dirk Schmidl and others. *How to scale Nested OpenMP Applications on the ScaleMP vSMP Architecture.* IEEE Cluster 2010 / Heraklion, September 21, 2010.

[65] David P. DiVincenzo. *Quantum Computation.* Science 270 (5234)., 1995.

[66] Thomas Sterling Donald J. Becker et al. *BEOWULF: A parallel workstation for scientific computation.* Aerospace Conference, 1998 IEEE, Volume: 4, 1998.

[67] Mora P Donnellan A. et al. *Computational earthquake science.* Birkhauser, Basel, 2004.

[68] Jones P.W. Drake J.B. et al. *Software design for petascale climate science.* Petascale computing. Algorithms and applications, Chapman & HAll/CRC, Boca Raton, FL, 2008.

[69] Michael J. Duff. *M-Theory (the Theory Formerly Known as Strings).* International Journal of Modern Physics A, 11, 1996.

[70] M. I. Dyakonov. *Is Fault-Tolerant Quantum Computation Really Possible?* Future Trends in Microelectronics. Up the Nano Creek, S. Luryi, J. Xu, and A. Zaslavsky (eds), Wiley arXiv:quant-ph/0610117, 2006.

[71] J. Brooks E. Anderson et al. *Performance of the CRAY T3E multiprocessor.* Supercomputing '97 Proceedings of the 1997 ACM/IEEE conference on Supercomputing, 1997.

[72] J. Goldstone E. Farhi et al. *Quantum Computation by Adiabatic Evolution.* arXiv; Cornell University Library, 2000.

[73] John F. Wendt (Ed.). *Computational Fluid Dynamics.* Springer, Third Edition, ISBN: 978-3-540-85055-7, 2002.

[74] Charles Edge. *Using Mac OS X Lion Server.* O'Reilly Media, March 2012.

[75] George Ellis. *Does the Multiverse Really Exist?* Scientific American 305 (2), 2011.

[76] Jim Elvridge. *The Universe - Solved!* AT Press, 2008.

[77] Christophe Grojean Emilian Dudasa. *Four-dimensionalM-theory and supersymmetry breaking.* Nuclear Physics B Volume 507, Issue 3, 15 December 1997.

[78] Brent R. Endy D. *Modelling cellular behaviour.* Nature, 409, December 2001.

[79] Hugh Everett. *The Theory of the Universal Wavefunction.* Manuscript, pp 3-140 of Bryce DeWitt, R. Neill Graham, Princeton Series in Physics, Princeton University Press, 1955 (wydrukowano w 1973).

[80] Hugh Everett. *'Relative state' formulation of quantum mechanics.* Reviews of Modern Physics 29 (3): 454-462, 1957.

[81] Van-Catledge F.A. *Towards a general model for evaluating the relative performance computer systems.* The International Jurnal of Supercomupter Applications, 3, 2, Summer 1989.

[82] Rob Farber. *CUDA Application Design and Development.* Morgan Kaufmann; 1 edition, November 2011.

[83] Richard Feynman. *The Character of Physical Law (1965).* Transcript of the Messenger Lectures at Cornell University, November 1964.

[84] Richard P. Feynman. *Simulating Physics with Computers.* International Journal of Theoretical Physics, VoL 21, Nos. 6/7, 1982.

[85] L. Fortnow. *The status of the P versus NP problem.* Communications of the ACM 52 (9): 78, 2009.

[86] Burkhardt H. i Rothnie J. Frank S. *The KSR1: Bridging the gap between shared memory and MPPs.* Proc. of the COMPCON Digest of Papers, 1993.

[87] Amdahl G. *Validity of the single processor approach to achieving large scale computing capabilities.* AFIPS Conference Proc., vol. 30, Washington D.C., Thompson Book, April 1967.

[88] Kirby Collins Gary Gostin, Jean-Francois Collard. *The architecture of the HP Superdome shared-memory multiprocessor.* ICS '05 Proceedings of the 19th annual international conference on Supercomputing, 2005.

[89] Jeremy Geelan. *Moore's Law: "We See No End in Sight", Says Intel's Pat Gelsinger.* SYS-CON. http://java.sys-con.com/node/557154 (Ostatni dostep: 2.09.2012), 2008.

[90] Heath M.T. Geist G.A. et al. *PVM: A framework for parallel distributed computing.* Technical Report, DOE Contract Number: AC05-84OR21400, Report Number(s): ESTSC–000585PSC1200, 1990.

[91] Amdahl Gene. *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities.* AFIPS Conference Proceedings (30), 1967.

[92] Francois V. Louveaux Gilbert Laporte and Helene Mercure. *Genetic Algorithms in Search, Optimization Machine Learning.* Operations Research Vol. 42, No. 3, May - Jun., 1994.

[93] Peter N. Glaskowsky. *NVIDIA's Fermi: The First Complete GPU Computing Architecture.* Electronic material, September 2009.

[94] Kurt Godel. *Uber formal unentscheidbare Satze der Principia Mathematica und verwandter Systeme, I.* Monatshefte fĂĽr Mathematik und Physik 38, 1931.

[95] D. E. Goldberg. *Genetic Algorithms in Search, Optimization Machine Learning.* New York: Addison-Wesley, 1989.

[96] Oded Goldreich. *P, Np, and Np-Completeness.* Cambridge: Cambridge University Press, 2010.

[97] Hedetniemi S.T. Goodman S.E. *Introduction to design and analysis of algorithms.* McGraw-Hill, New York, 1997.

[98] googledatacenters.blogspot.com. *Google Sorts 1 Petabyte of Data in 6 Hours.* http://googledatacenters.blogspot.com/2009/11/google-sorts-1-petabyte-of-data-in-6.html Ostatni dostÄ™p: 11.09.2012, November, 2009.

[99] Skjellum Anthony Gropp William, Lusk Ewing. *Using MPI: portable parallel programming with the message-passing interface.* Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series. ISBN 0-262-57104-8, 1994.

[100] Skjellum Anthony Gropp William, Lusk Ewing. *A High-Performance, Portable Implementation of the MPI Message Passing Interface.* CiteSeerX: 10.1.1.102.9485, 1996.

[101] Khronos OpenCL Working Group. *The OpenCL Specification.* Khronos, 2011.

[102] Lov Kumar Grover. *A fast quantum mechanical algorithm for database search.* Proceedings, 28th Annual ACM Symposium on the Theory of Computing, 1996.

[103] Montry G.R i Benner R.E. Gustafson J.L. *Development of parallel methods for a 1024-processor hypercube.* SIAM Jurnal on Scientific and Statistical Computing, 9, March 1988.

[104] L. Hales. *An improved quantum Fourier transform algorithm and applications.* Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium, 2000.

[105] R.J. Harrison. *The TCGMSG Message-Passing Toolkit.* Pacific Northwest National Laboratory, version 4.04, 1994.

[106] Stephen Hawking. *Krotka historia czasu.* Zysk i S-ka, 2000.

[107] Matthew Hayward. *Quantum Computing and Shor's Algorithm.* February 17, 2005.

[108] Markus Hegland. *Real and complex fast Fourier transforms on the Fujitsu VPP 500.* Parallel Computing, Volume 22, Issue 4, June 1996.

[109] Farrell M. S. Heller L.C. *Millicode in an IBM zSeries processor.* IBM Journal of Research and Development, Volume: 48, Issue: 3.4, May 2004.

[110] Mika Hirvensalo. *Algorytmy kwantowe.* Wydawnictwa Szkolne i Pedagogiczne Spolka Akcyjna., Warszawa 2004.

[111] Nikolaos Drosinos i Nectarios Koziris. *Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters.* Parallel and Distributed Processing Symposium. Proceedings. 18th International, 2004.

[112] Tim Foley Ian Buck et al. *Brook for GPUs: Stream Computing on Graphics Hardware.* SIGGRAPH, 2004.

[113] Carl Kesselman Ian Fostera. *The Globus project: a status report.* Future Generation Computer Systems, Volume 15, Issues 5-6, October 1999.

[114] Martijn A. Huynen Ivo L. Hofacker et al. *RNA Folding on Parallel Computers: The Minimum Free Energy Structures of Complete HIV Genomes.* Paper provided by Santa Fe Institute in its series Working Papers with number 95-10-089, October 1995.

[115] Augen J. *The evolving role of information technology in the drug discovery process.* Drug Discovery Today, 7, 5, March 2002.

[116] Quinn Michael J. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Inc., 2004.

[117] Schwartz J. *A taxonomic table of parallel computers based on 55 designs.* New York: Courant Institute, New York University, Nowember 1983.

[118] S. H. Lee J. Jahns. *Optical Computing Hardware.* Academic Press, Boston, 1994.

[119] Antoine Petitet Jack J. Dongarra1, Piotr Luszczek1. *The LINPACK Benchmark: past, present and future.* Concurrency and Computation: Practice and Experience, Volume 15, Issue 9, August 2003.

[120] Haqiang H. Jin Jahed Djomehri. *Hybrid MPI+OpenMP Programming of an Overset CFD Solver and Performance Investigations.* NASA Report, ntrs.nasa.gov, 2002.

[121] Benjamin E. Childs James H. Brodeur. *Random Search Algorithms.* Major Qualifying Project Report submitted to the Faculty of the WORCESTER POLYTECHNIC INSTITUTE, 2008.

[122] Daniel Lenoski James Laudon. *The SGI Origin: a ccNUMA highly scalable server.* ISCA '97 Proceedings of the 24th annual international symposium on Computer architecture, Volume 25, Issue 2, May 1997.

[123] Mark Horowitz James Laudon, Anoop Gupta. *Interleaving: a multithreading technique targeting multiprocessors and workstations.* ASPLOS-VI Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, 1994.

[124] Robert J. Harrison i Richard J. Littlefield Jaroslaw Nieplocha. *Global arrays: A nonuniform memory access programming model for high-performance computers.* The Journal of Supercomputing, Volume 10, Number 2, 169-189, DOI: 10.1007/BF00130708, 1996.

[125] Edward Kandrot Jason Sanders. *CUDA by Example, An Introduction to General-Purpose GPU Programming.* Addison-Wesley, 2011.

[126] James Goodman Jason Yang. *Symmetric key cryptography on modern graphics hardware.* ASIACRYPT'07 Proceedings of the Advances in Crypotology 13th international conference on Theory and application of cryptology and information security, 2007.

[127] Chris McVay nad others Jeff Freeman. *Intel HD Graphics DirectX Developer's Guide.* Intel, Document Number: 321371-002, Revision: 2.8.0, 2010.

[128] Sanjay Ghemawat Jeffrey Dean. *MapReduce: simplified data processing on large clusters.* Communications of the ACM - 50th anniversary issue: 1958 - 2008, Volume 51, Issue 1, 2000.

[129] Gustafson J.L. *Reevaluating Amdahl's law.* Communications of the ACM, 31, May 1988.

[130] Roberto Olivares-Amaya Johannes Hachmann et al. *The Harvard Clean Energy Project: Large-Scale Computational Screening and Design of Organic Photovoltaics on the World Community Grid.* J. Phys. Chem. Lett. 2, 2241â“2251, 2011.

[131] Daniel K. Price John R. Humphrey et al. *CULA: Hybrid GPU Accelerated Linear Algebra Routines.* Document Society of Photo-Optical Instrumentation Engineers, 2010.

[132] III John T. Gill. *Computational complexity of probabilistic Turing machines.* STOC '74 Proceedings of the sixth annual ACM symposium on Theory of computing, 1974.

[133] Richard Blum Jon Masters. *Professional Linux Programming (Programmer to Programmer).* ISBN-13: 978-0471776130, March 12, 2007.

[134] Keiji Matsumoto Jumpei Niwa and Hiroshi Imai. *General-Purpose Parallel Simulator for Quantum Computing.* Lecture Notes in Computer Science, 2002, Volume 2509/2002, 230-251, DOI: 10.1007/3-540-45833-$6_20, 2002$.

[135] Li K. *Shared virtual memory on loosely coupled multiprocessor.* Ph.D. thesis, Departament of Computer Science, Yale University, 1986.

[136] K. Michielsen K. De Raedta et al. *Massively parallelquantum computer simulator.* Computer Physics Communications, Volume 176, Issue 2, 15 January 2007.

[137] Michio Kaku. *Introduction to Superstring and M-Theory (2nd edition ed.).* New York, USA: Springer-Verlag, 1999.

[138] Michio Kaku. *Hiperprzestrzen.* Proszynski i S-ka, Warszawa 2012.

[139] Firas Hamze Kamran Karimi, Neil G. Dickson. *A Performance Comparison of CUDA and OpenCL.* arXiv:1005.2581v1, May 2011.

[140] Neil Dickson i Firas Hamze Kamran Karimi. *High-Performance Physics Simulations Using Multi-Core CPUs and GPGPUs in a Volunteer Computing Context.* International Journal of High Performance Computing Applications. doi:10.1177/1094342010372928, 2010.

[141] Jeff Kanipe. *Cosmic simulations.* Magazine Communications of the ACM, Volume 55 Issue 8, August 2012.

[142] J.E. Karkhanis T.S., Smith. *A first-order superscalar processor model.* Computer Architecture, 2004. Proceedings. 31st Annual International Symposium, 2004.

[143] Flatt H.P. Karp A.H. *Measuring parallel processor performance.* Communications of the ACM, 33, May 1990.

[144] Pawankumar Hegde Katie Roberts-Hoffman. *ARM Cortex-A8 vs. Intel Atom: Architectural and Benchmark Comparisons.* University of Texas at Dallas, Electronic document, 2009.

[145] Pawankumar Hegde Katie Roberts-Hoffman. *BOINC: a system for public-resource computing and storage.* University of Texas at Dallas, Electronic document, 2009.

[146] Goodman J.R. Kaxiras S. *Improving CC-NUMA performance using Instruction-based Prediction.* High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium, 1999.

[147] Robert Cooper Kenneth Birman. *The ISIS project: real experience with a fault tolerant programming system.* EW 4 Proceedings of the 4th workshop on ACM SIGOPS European workshop, 1990.

[148] Dennis M. Ritchie Kernighan, Brian W. *The C Programming Language (2nd ed.).* Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110163-3, March 1988.

[149] Yalamanchili S. Kerr A., Diamos G. *A characterization and analysis of PTX kernels.* Workload Characterization, 2009. IISWC 2009. IEEE International Symposium, October 2009.

[150] Khronos. *OpenCL API 1.0 Quick Reference Card.* Khronos, 2009.

[151] D. Kim et al. *Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors.* Code Generation and Optimization, 2004. CGO 2004. International Symposium, 1996.

[152] Hwu W-M.W. Kirk D.B. *Programming massively parallel processors. A hands-on approach.* Morgan Kaufmann, Amsterdam, 2010.

[153] Gunnels J. nad others Kistler M. *Programming the Linpack benchmark for Roadrunner.* IBM Journal of Research and Development, Volume: 53, Issue: 5, September 2009.

[154] Bangerth W. Klie H. et al. *Models, methods and middleware for grid-enabled multiphysics oil reservoir management.* Engineering with Computers, 22, 3-4, December 2006.

[155] Ken Koch. *The New Roadrunner Supercomputer: What, When, & How.* SC06, LA-UR-06-8276, November 2006.

[156] Anderson D. Korpela E. et al. *SETI@home-massively distributed computing for SETI.* Computing in Science & Engineering, Volume: 3, Issue: 1, 2001.

[157] Marcin Kaminski Krzysztof Giaro. *Wprowadzenie do algorytmow kwantowych.* EXIT, Warszawa 2003.

[158] Benjamin Hill Kyle Rankin. *The Official Ubuntu Server Book.* Prentice Hall; 1 edition, ISBN-13: 978-0137021185, 2009.

[159] Clemens Lautemann. *BPP and the polynomial hierarchy.* Information Processing Letters Volume 17, Issue 4, 8 November 1983.

[160] Greg Lehey. *The Complete FreeBSD, 4th Edition.* O'Reilly Media, April 2003.

[161] Hudak P. Li K. *Memory coherence in shared virtual memory systems.* ACM Transactions on Computer Systems, November 1989.

[162] Nickolls J. Lindholm E. et al. *NVIDIA Tesla: A Unified Graphics and Computing Architecture.* Micro, IEEE, Volume: 28, Issue: 2, March-April 2008.

[163] Michael W. Lucas. *Absolute FreeBSD: The Complete Guide to FreeBSD, 2nd Edition.* No Starch Press; 2 edition, November 14, 2007.

[164] Viktors Berstis Luis Ferreira. *Introduction to Grid Computing with Globus. RedBook.* IBM, International Technical Support Organization, December 2002.

[165] Swierczewski Lukasz. *Kwantowa przyszlosc informatyki.* Forum Akademickie 07-08/2011, 2011.

[166] Swierczewski Lukasz. *Transformations of Qubit System States within Quantum Circuits after Passing through Hadamard Gates.* Reseach Poster, Proceedings ISC 2012, Hamburg, 2011.

[167] D. Chen M. Blumrich et al. *Design and Analysis of the BlueGene/L Torus Interconnection Network.* RC23025 (W0312-022), Computer Science, December 2003.

[168] G M. Boyer et al. *Tight bounds on quantum searching.* arXiv: quant-ph/9605034, 1996.

[169] J. Nishimura M. Ozawa. *Computational complexity of uniform quantum circuit families and quantum Turing machines.* arXive: quant-ph:/9906095, 1999.

[170] L. Mandel. *Quantum effects in one-photon and two-photon interference.* Reviews of Modern Physics, Vol. 71, No. 2, 1999.

[171] Nitin Saxena Manindra Agrawal, Neeraj Kayal. *PRIMES is in P.* Annals of Mathematics 160, no. 2, 2004.

[172] Jeffrey Oldham Mark Mitchell and Alex Samuel. *Advanced Linux Programming.* New Riders Publishing, June 2001.

[173] Brent Welch Mark Weiser et al. *Scheduling for Reduced CPU Energy.* Mobile Computing, The Kluwer International Series in Engineering and Computer Science, Volume 353, 449-471, DOI: 10.1007/978-0-585-29603-$6_1$7, 1996.

[174] Furumura T. Matsu'ura M. et al. *Integrated predictive simulation system for earthquake and Tsunami disaster.* SIAM 12th Conference on Parallel Processing for Scientific Computing (PP06), San Francisco, 2006.

[175] James Clerk Maxwell. *Ether.* Encyclopedia Britannica Ninth Edition, 1878.

[176] Ashwin M. Aji i Wu-chun Feng Mayank Daga. *On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing.* Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium, 2011.

[177] Alastair D. McAulay. *Optical Computer Architectures: The Application of Optical Concepts to Next Generation Computers.* New York, NY: John Wiley & Sons., 1991.

[178] Michael McCallister. *openSUSE Linux Unleashed.* Sams, 2007.

[179] Monica S. Lam i Mark A. Horowitz Michael D. Smith. *Boosting beyond static scheduling in a superscalar processor.* ISCA '90 Proceedings of the 17th annual international symposium on Computer Architecture, 1990.

[180] Peter Zaspel Michael Griebel. *A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations.* Computer Science - Research and Development, Volume 25, Numbers 1-2, 65-73, DOI: 10.1007/s00450-010-0111-7, 2010.

[181] Isaac Chuang Michael Nielsen. *Quantum Computation and Quantum Information.* Cambridge: Cambridge University Press., 2000.

[182] Drew McCormack Michael Trent. *Beginning Mac OS X Programming.* Wiley / Wrox, October 2005.

[183] Grzegorz KarpinĔski Michel Gabassi, Bertrand Dupouy. *Przetwarzanie rozproszone w systemie UNIX.* Lupus, 1995.

[184] UEdin-EPCC Michele Weiland. *DEISA 2, Distributed European Infrastructure For Supercomputing Applications.* European Community Seventh Framework Programme, Research Infrastructures, Integrated Infrastructure Initiative, Deliverable ID: DEISA2-D7-2.3, Contract Number RI-222919, 2010.

[185] Morley Williams Michelson Abraham. *On the Relative Motion of the Earth and the Luminiferous Ether.* American Journal of Science 34, 1887.

[186] Flynn M.J. *Some computer organizations and their effectiveness.* IEEE Transactions on Computing, C-21, 1972.

[187] Quinn M.J. *Paralllel computing. Theory and practice.* McGraw-Hill, New York, 1994.

[188] Quinn M.J. *Paralllel programming in C with MPI and OpenMP.* McGraw-Hill, New York, 2004.

[189] Kim J. Moin P. *Tackling turbulence with supercomputers.* Scientific American 276, 1997.

[190] Hackenberg D. Molka D. et al. *Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System.* Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference, September 2009.

[191] Sessa F. i Milite A. Monacelli G. *An integrated approach to evaluate engineering simulations and ergonomic aspects of a new vehicle in a virtual environment: phisical and virtual correlation methods.* FISITA 2004 30th World Automotive Congress, Barcelona, Spain, 2004.

[192] Sessa F. i Milite A. Monacelli G. *Analyzing engineering simulations in a virtual environment.* IEEE Computer Graphics and Applications, 18, 16, November 1998.

[193] Gordon E. Moore. *Cramming more components onto integrated circuits.* Electronics Magazine 38 (8), 19 kwietnia 1965.

[194] Vandad Nahavandipoor. *Concurrent Programming in Mac OS X and iOS.* O'Reilly Media, May 2011.

[195] nanotech now.com. *Harris & Harris Group Notes Sale of Quantum Computing System by D-Wave Systems to Lockheed Martin Corporation.* nanotech-now.com. http://www.nanotech-now.com/news.cgi?story$_id$ = 42543($Ostatnidostep$: 2.09.2012), 2011.

[196] Julian Seward Nicholas Nethercote. *Valgrind: a framework for heavyweight dynamic binary instrumentation.* PLDI '07 Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, 2007.

[197] Lo V. Nitzberg B. *Distributed shared memory: a survey of issues and algorithms.* Computer, Volume: 24, Issue: 8, August 1991.

[198] Dally W.J. Nuth P.R. *A mechanism for efficient context switching.* Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference, October 1991.

[199] nVidia. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi.* Whitepaper, 2009.

[200] nVidia. *TESLA C2050 / C2070 GPU Computing Processor.* DATASHEET, Electronic material, Technical specyfication, 2010.

[201] nVidia. *NVIDIA CUDA C Programming Guide, Version 4.2.* Electronic material, 2012.

[202] nVidia. *CUDA Toolkit 4.2 CUBLAS Library.* nVidia, PG-05326-$041_v01, February$2012.

[203] Harris M. i Prins J. Nylan L. *Fast N-body simulations with CUDA.* GPU Gems 3, 31, Addison-Wesley, August 2007.

[204] Encyclopedia of Laser Physics and Technology. *Nonlinear Index.* $www.rp-photonics.com/nonlinear_index.html$, Ostatni dostep: 23.07.2012.

[205] BBC Online. *Chilly chip shatters speed record.* BBC Online. http://news.bbc.co.uk/2/hi/technology/5099584.stm (Ostatni dostep: 2.09.2012), 2006.

[206] Rodgers David P. *Improvements in multiprocessor system design.* ACM SIGARCH Computer Architecture News archive (New York, NY, USA: ACM) 13 (3), June 1985.

[207] Peter Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers Inc., 1997.

[208] lain S. Duff Patrick R. Amestoy. *Vectorization of a Multiprocessor Multifrontal Code.* International Journal of High Performance Computing Applications vol. 3 no. 3, doi: 10.1177/109434208900300303, September 1989.

[209] David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, ISBN 0-89871-464-8, 2000.

[210] Richard Petersen. *Red Hat Linux: The Complete Reference, Second Edition.* McGraw-Hill Professional, ISBN:0072191783, 2001.

[211] James Pierpont. *Non-euclidean geometry, a retrospect.* Bull. Amer. Math. Soc. Volume 36, Number 2, 1930.

[212] Carl Pomerance. *A Tale of Two Sieves.* Notices of the AMS 43 (12), 1996.

[213] press.web.cern.ch. *Let the number-crunching begin: the Worldwide LHC Computing Grid celebrates first data.*

http://press.web.cern.ch/press/PressReleases/Releases2008/PR13.08E.html Ostatni dostÄ™p: 11.09.2012, 2008.

[214] Tomasevic M. i Milutinovic V. Protic J. *Distributed shared memory: concepts and systems.* Parallel Distributed Technology: Systems Applications, IEEE, Volume: 4 , Issue: 2, Summer 1996.

[215] QJ.net. *Ćell Designer talks about PS3 and IBM Cell Processors"http://www.qj.net/qjnet/playstation-3/cell-designer-talks-about-ps3-and-ibm-cell-processors.html.* QJ.net (Ostatni dostep: 10.09.2012), 2006.

[216] William Gropp i Ewing Lusk Rajeev Thakur. *On implementing MPI-IO portably and with high performance.* IOPADS '99 Proceedings of the sixth workshop on I/O in parallel and distributed systems, 1999.

[217] Ewing L Lusk Ralph M Butler. *Monitors, messages, and clusters: The p4 parallel programming system.* Parallel Computing, Volume 20, Issue 4, April 1994.

[218] Abhiram Ranade. *Optimal speedup for backtrack search on a butterfly network.* Theory of Computing Systems, Volume 27, Number 1, 85-101, DOI: 10.1007/BF01187094, 1994.

[219] M. Raynal. *Algorithms for mutual exclusion.* The MIT Press,Cambridge, MA, 2008 Feb 07.

[220] Wolfgang Richtera et al. *Functional magneticresonanceimaging with intermolecular multiple-quantum coherences.* Magnetic Resonance Imaging Volume 18, Issue 5, June 2000.

[221] Christian Benjamin Ries. *BOINC ist Public-Resource Computing.* BOINC, Xpert.press, Part 1, 17-23, DOI: $10.1007/978\text{-}3\text{-}642\text{-}23383\text{-}8_2$, 2012.

[222] Ivan E. Sutherland i Charles E. Molnar Robert F. Sproull. *The counterflow pipeline processor architecture.* IEEE Design Test of Computers, Vol. 11, No. 3, SMLI TR-94-25, April 1994.

[223] Russell Taylor Robert Jacques et al. *Towards real-time radiation therapy: GPU accelerated superposition/convolution.* Computer Methods and Programs in Biomedicine, Volume 98, Issue 3, June 2010.

[224] Larry Roberts. *The Arpanet and computer networks.* HPW '86 Proceedings of the ACM Conference on The history of personal workstations, 1986.

[225] Georg Hager i Gabriele Jost Rolf Rabenseifner. *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes.* Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference, 2009.

[226] Takashi Nakamura Ryoji Tsuchiyama et al. *The OpenCL Programming Book.* Fixstars, 2012.

[227] R. Barrett S. Alam et al. *Early evaluation of IBM BlueGene/P.* SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008.

[228] Fahmy H.A.H. Samy R. et al. *A decimal floating-point fused-multiply-add unit.* Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium, August 2010.

[229] Stefano Forli Sandro Cosconati et al. *Virtual screening with AutoDock: theory and practice.* Informa Healthcare, Expert Opinion on Drug Discovery, Vol. 5, No. 6 , Pages 597-607, doi:10.1517/17460441.2010.484460, June 2010.

[230] Umesh Vazirani Sanjoy Dasgupta, Christos Papadimitriou. *Algorytmy.* Wydawnictwo Naukowe PWN, Warszawa 2010.

[231] Paul Schreier. *Supercomputing at CERN : How HPC is helping the LHC.* Scientific Computer World, August/September 2008.

[232] Rudy Setiono. *Extracting rules from pruned neural networks for breast cancer diagnosis.* Artificial Intelligence in Medicine, Volume 8, Issue 1, February 1996.

[233] Seung-Jai Min i Rudolf Eigenmann Seyong Lee. *OpenMP to GPGPU: a compiler framework for automatic translation and optimization.* PPoPP '09 Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, ISBN: 978-1-60558-397-6, 2009.

[234] Andrew A. Chien Shekhar Borkar. *The Future of Microprocessors.* Communications of ACM 54 (5), 05 2011.

[235] Jamie Shiers. *The WorldwideLHCComputingGrid (worldwide LCG).* Computer Physics Communications, Volume 177, Issues 1-2, Proceedings of the Conference on Computational Physics 2006 - CCP 2006, July 2007.

[236] Anand Lal Shimpi. *AnandTech: Intel's 90nm Pentium M 755: Dothan Investigated.* Anadtech. http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2129p=3 (Ostatni dostep: 2.09.2012), 2004.

[237] Mitsuo Yokokawa i Shigemune Kitawaki Shinichi Habata. *The Earth Simulator System.* Architecture and Hardware for HPC, 2003.

[238] Peter W. Shor. *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.* SIAM J. Comput. 26 (5) arXiv:quant-ph/9508027v2, 1997.

[239] Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition).* Addison-Wesley, 2009.

[240] Yuetsu Kodama i Yoshinori Yamaguchi Shuichi Sakai. *Design and implementation of a circular omega network in the EM-4.* Parallel Computing, Volume 19, Issue 2, February 1993.

[241] Daniel Duffy Shujia Zhou et al. *Impacts of the IBM Cell Processor on Supporting Climate Models.* ISC 2008, Research Poster, 2008.

[242] Joseph D Sloan. *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI.* O'Reilly Media, November 2004.

[243] Becker D. Sterling T. et al. *An assessment of Beowulf-class computing for NASA requirements: initial findings from the first NASA workshop on Beowulf-class clustered computing.* Aerospace Conference, 1998 IEEE, Volume: 4, 1998.

[244] Nancy B. Stern. *From ENIAC to UNIVAC: An appraisal of the Eckert-Mauchly computers.* Digital Press (Bedford, Mass.), 1981.

[245] Dennis Abts Steve Scott et al. *The BlackWidow High-Radix Clos Network.* ISCA '06 Proceedings of the 33rd annual international symposium on Computer Architecture, 2006.

[246] James Wyllie Steven Fortune. *Parallelism in random access machines.* STOC '78 Proceedings of the tenth annual ACM symposium on Theory of computing, 1978.

[247] Ghosh Sukumar. *Distributed Systems â" An Algorithmic Approach.* Chapman & Hall/CRC, ISBN 978-1-58488-564-1, 2007.

[248] III Werthimer Dan Sullivan Woodruff T. et al. *A new major SETI project based on Project SERENDIP data and 100,000 personal computers.* Conference Paper, Astronomical and Biochemical Origins and the Search for Life in the Universe, IAU Colloquium 161, Publisher: Bologna, Italy, p. 729., 1997.

[249] V. S. Sunderam. *PVM: A framework for parallel distributed computing.* Concurrency: Practice and Experience, Volume 2, Issue 4, December 1990.

[250] Peter B. Kessler i Marshall K. Mckusick Susan L. Graham. *Gprof: A call graph execution profiler.* SIGPLAN '82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction, 1982.

[251] L. Swierczewski. *Generating extrema approximation of analytically incomputable functions through usage of parallel computer aided genetic algorithms.* arXiv; Cornell University Library, 2013.

[252] L. Swierczewski. *Oproject@Home - distributed computing.* Proceedings LVEE 2013 Conference, 2013.

[253] L. Swierczewski. *Simulation of Grover's algorithm on parallel computers with shared memory and using the Olib library.* Proceedings LVEE 2012 Conference, 7-10 June, 2012.

[254] Research team at the University of Pittsburgh. *Super-small transistor created: Artificial atom powered by single electron.* Science Daily. http://www.sciencedaily.com/releases/2011/04/110418135541.htm (Ostatni dostep: 2.09.2012), 2011.

[255] Max Tegmark. *Parallel Universes.* Scientific American, 2003.

[256] Michael L. Scott i Christopher M. Brown Thomas J. LeBlanc. *Large-scale parallel programming: experience with BBN butterfly parallel processor.* PPEALS '88 Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems, September 1988.

[257] Donald J. Becker Thomas Sterling et al. *How to build a Beowulf: A Guide to the Implementation and Application of PC Clusters.* Massachusetts Institute of Technology, 1999.

[258] Tomoyoshi Ito Tomoyoshi Shimobaba et al. *Fast calculation of computer-generated-hologram on AMD HD5000 series GPU and OpenCL.* Optics Express, Vol. 18, Issue 10, 2010.

[259] TOP500. *http://www.top500.org.* TOP500, Oststni dostÄ™p: 14.09.2012, 2012.

[260] Yi Zhang Tsigas P. *A simple, fast parallel implementation of Quicksort and its performance evaluation on SUN Enterprise 10000.* Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference, 2003.

[261] Feng T.Y. *Some characteristics of associative/parallel processing.* Proc. of the 1972 Sagamore Computing Conference, 1972.

[262] Chris Tyler. *Fedora Linux: A Complete Guide to Red Hat's Community Distribution.* O'Reilly Media; 1 edition, 2006.

[263] R. Ursin. *Space-quest: experiments with quantum entanglement in space.* Int. Aeronautical Congress Proc. A2.1.3 arXiv:0806.0945, 2008.

[264] Dongarra J.J. Van der Stenn A.J. *Overview of recent supercomputers.* (www.top500.org), 2006 i 2007.

[265] Lieven M. K. Vandersypen et al. *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance.* Nature 414, 2001.

[266] Adams M.D. Venter J.C. et al. *The sequence of the human genome.* Science, 291, 16 February 2001.

[267] Alam S.R. Vetter J.S. et al. *Early evaluation of the Cray XT3.* Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, April 2006.

[268] Raphael Bolze Viktors Bertis et al. *From Dedicated Grid to Volunteer Grid: Large Scale Execution of a Bioinformatics Application.* Journal of Grid Computing, Volume 7, Number 4, 463-478, DOI: 10.1007/s10723-009-9130-7, 2009.

[269] Rizzi A. Vos J.B. et al. *Navier-Stokes solvers in European Aucraft design.* Progress in Aerospace Sciences, 38, 2002.

[270] Hendler W. *The impact of classification schemes on computer architecture.* Proc. of the International Conference on Parallel Processing, August 1977.

[271] W. Zurek W. Wootters. *A single quantum cannot be cloned.* Nature 299, 1982.

[272] B. Wang. *Engineering modular and orthogonal genetic logic gates for robust digital-like synthetic biology.* Nature Communications, 2011.

[273] Steven Weinberg. *Testing Quantum Mechanics.* Annals of Physics Vol 194 (2), 1989.

[274] Steven Weinberg. *Dreams of a Final Theory: The Search for the Fundamental Laws of Nature.* New York: Pantheon Books, 1993.

[275] Y. S. Weinstein et al. *Implementation of the Semiclassical Quantum Fourier Transform in a Scalable System.* Science Vol. 308 no. 5724, 13 May 2005.

[276] Y. S. Weinstein et al. *Implementation of the Quantum Fourier Transform.* Phys. Rev. Lett. 86, 2001.

[277] Allen Sherrod Wendy Jones. *Beginning Directx 11 Game Programming.* Microsoft Press Redmond, WA, USA, 2012.

[278] Wikipedia. *General number field sieve http://en.wikipedia.org/wiki/General_number_field_sieve.* Wikipedia (Ostatni dostep: 3.05.2013), 2013.

[279] C. G. Bell William A. Wulf. *C.mmp: a multi-mini-processor.* AFIPS '72 (Fall, part II) Proceedings of the December 5-7, 1972, fall joint computer conference, part II, 1972.

[280] Wookey and Paul Webb. *Guide to ARMLinux for Developers.* Aleph One, 2001.

[281] Droegemeier K.K. i Weber D. Xue M. *Numerical prediction of high-impact local weather: A driver for petascale computing.* Petascale computing. Algorithms and applications, Chapman & HAll/CRC, Boca Raton, FL, 2008.

[282] Zelda B. Zabinsky. *Random Search Algorithms.* April 5, 2009.

[283] A. Zeilinger et al. *Feasibility of* 300 *km quantum key distribution with entangled states.* New J. Phys. 11, 2009.

[284] A. Zeilinger et al. *High-fidelity transmission of entanglement over a high-loss free-espace channel.* Nat. Phys. 5, 2009.

[285] Fabian Chudak ZhengBing Bian et al. *Experimental determination of Ramsey numbers with quantum annealing.* arXiv:1201.1842, 2012.

Wykaz ważniejszych skrótów i oznaczeń

$f = O(g)$	notacja dużego O; funkcja f rośnie nie szybciej niż funkcja g
c^*	liczba zespolona sprzężona z c
$\mathbb{Z}_N$	pierścień liczb całkowitych modulo N
$\mathbb{Z}_N^*$	grupa reszt będących względnie pierwsza z N
$m\|n$	n jest podzielna przez m
$n \equiv m \ (mod \ N)$	kongruencja; n przystaje do m modulo N
$\mathbb{H}$	przestrzeń Hilberta
$\mathbb{H}_n$	n-wymiarowa przestrzeń Hilberta
$\|\psi\rangle$	wektor w przestrzeni Hilberta
$\langle\psi\|$	wektor w sprzężony do $\|\psi\rangle$

Spis rysunków

Spis tablic

Spis listingów

Załączniki

A. Kwantowa transformata Fouriera

```
1  void dqft_sp(complex_float q_register[], unsigned long long int q)
2  {
3      complex_float init[q];
4
5      unsigned long long i;
6      for(i = 0; i < q; i++)
7      {
8          init[i].real = 0;
9          init[i].imag = 0;
10     }
11
12     complex_float tmpcomp;
13
14     tmpcomp.real = 0.0;
15     tmpcomp.imag = 0.0;
16
17     float term_1;
18     term_1 = pow(q, -.5);
19
20     float term_2;
21
22     float epsilon;
23
24     epsilon = pow(10, -14);
25
26     unsigned long long int a;
27     unsigned long long int c;
28
29     float sin_value;
30     float cos_value;
31
32     for (a = 0 ; a < q ; a++)
33     {
34         term_2 = 2*PI*a/q;
35
36         if ((pow(q_register[a].real, 2) + pow(q_register[a].imag, 2)) > epsilon)
37         {
38             for (c = 0 ; c < q ; c++)
39             {
40                 term_2 = 2*PI*a*c/q;
41
42                 sincosf(term_2, &sin_value, &cos_value);
43
44                 tmpcomp.real = term_1 * cos_value;
45                 tmpcomp.imag = term_1 * sin_value;
46
47                 init[c] = complex_add(init[c], complex_mul(q_register[a], tmpcomp↩
                       ));
48             }
49         }
50     }
51
52     copy_vector_sp(q_register, q, init);
53     vector_normalization_sp(q_register, q);
```

```
54 }
```

Listing 33. Możliwa sekwencyjna implementacja kwantowej tranformaty Fouriera w języku C.

```
SUBROUTINE dqft_sp_fortran(q_register, q, init_re, init_img)
    TYPE complex_double
        DOUBLE PRECISION            :: re
        DOUBLE PRECISION            :: img
    END TYPE complex_double

    INTEGER(8), intent(in)          :: q
    INTEGER(8)                      :: i
    INTEGER(8)                      :: a, c
    INTEGER(8)                      :: count_variable
    DOUBLE PRECISION                :: epsilon_variable
    DOUBLE PRECISION                :: vector_sum
    DOUBLE PRECISION                :: PI = 4 * ATAN(1.0)
    DOUBLE PRECISION                :: term_1;
    DOUBLE PRECISION                :: term_2;

    TYPE(complex_double)            :: init(q)
    TYPE(complex_double)            :: q_register(q)
    TYPE(complex_double)            :: tmpcomp
    TYPE(complex_double)            :: tmpcomp_2

    DO i=1, q
        init(i)%re = 0
        init(i)%img = 0
    END DO

    tmpcomp%re = 0.0
    tmpcomp%img = 0.0

    term_1 = q**-.5

    epsilon_variable = 10.0**-14.0

    DO a=1, q

        IF (q_register(a)%re**2.0 + q_register(a)%img**2.0 > epsilon_variable) ↩
            THEN

            DO c=1, q

                term_2 = 2.0*PI*a*c/q

                tmpcomp%re = term_1 * COS(term_2)
                tmpcomp%img = term_1 * SIN(term_2)

                tmpcomp_2%re = (q_register(a)%re * tmpcomp%re) - (q_register(a)%↩
                    img * tmpcomp%img);
                tmpcomp_2%img = (q_register(a)%img * tmpcomp%re) - (q_register(a)↩
                    %re * tmpcomp%img);

                init(c)%re = init(c)%re + tmpcomp_2%re
                init(c)%img = init(c)%img + tmpcomp_2%img

            END DO

        END IF

    END DO

    DO a=1, q
        q_register(a)%re = init(a)%re
```

```
59          q_register(a)%img = init(a)%img
60      END DO
61
62      vector_sum = 0
63
64      DO a=1, q
65          vector_sum = vector_sum + q_register(a)%re**2.0 + q_register(a)%img**2.0
66      END DO
67
68      vector_sum = SQRT(vector_sum)
69
70      DO a=1, q
71
72          q_register(a)%re = q_register(a)%re / vector_sum
73          q_register(a)%img = q_register(a)%img / vector_sum
74      END DO
75
76      RETURN
77
78  END SUBROUTINE
```

Listing 34. Możliwa sekwencyjna implementacja kwantowej tranformaty Fouriera w języku Fortran 95.

B. Trywialne rozwiązanie problemu rzędów

```
1
2
3 # include <stdio.h>
4
5 # define MAX_POW_RANGE 20000000
6
7 unsigned long long int nwd(unsigned long long int a, unsigned long long int b)
8 {
9     unsigned long long int tmp;
10    while(b != 0)
11    {
12        tmp = a % b;
13        a = b;
14        b = tmp;
15    }
16    return a;
17 }
18
19 unsigned long long int fast_modulo_power(unsigned long long int a, unsigned long ↩
      long int b, unsigned long long int m)
20 {
21    unsigned long long int i;
22    unsigned long long int result = 1;
23    unsigned long long int x = a%m;
24
25    for (i=1; i<=b; i<<=1)
26    {
27        x %= m;
28        if ((b&i) != 0)
29        {
30            result *= x;
31            result %= m;
32        }
33        x *= x;
34    }
35
36    return result;
37 }
38
39
40
41 long long int ord(unsigned long long int number, unsigned long long int z)
42 {
43
44    unsigned long long int pow_number;
45
46    unsigned long long int i;
47    for(i=1; i < MAX_POW_RANGE; i++)
48    {
49        pow_number = fast_modulo_power(number, i, z);
50
51        if(pow_number == 1)
52        {
53            return i;
54        }
55    }
```

```
56 }
57 int main(int argc, char **argv)
58 {
59     unsigned long long int n;
60 
61     if(argc == 2)
62     {
63         n = (unsigned long long int) atol(argv[1]);
64     }
65     else
66     {
67         scanf("%lld", &n);
68     }
69 
70     unsigned long long int order;
71     unsigned long long int p;
72     unsigned long long int i;
73 
74     for(i=1; i < n; i++)
75     {
76         if(nwd(n, i) == 1)
77         {
78             order = ord(i, n);
79 
80             if(order%2 == 1 && pow(i, (order/2)) != -1)
81             {
82                 p = nwd(pow(i, (order/2)) - 1, n);
83                 //printf("Result r [probable]: %lld\n", p);
84             }
85         }
86     }
87 
88     return 0;
89 }
```

Listing 35. Szkielet algorytmu Shora wraz z funkcją główną main() oraz funkcją shor_algorithm() odpowiedzialną za przeprowadzanie obliczeń. Wersja wykorzystana w obliczeniach na platformie BOINC dostosowana do obliczeń sekwencyjnych równoległych OpenMP oraz na kartach graficznych i innych urządzeniach (m. in. procesor Cell).

C. Implementacja algorytmu Grovera

```
1  /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
2   * grover_algorithm.cpp
3   *
4   * OLib
5   * High performance library of mathematical functions and algorithms
6   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
7
8
9  # define BATCH_MODE
10 # define QBITS_SIZE 14
11 # define CUDA_DEVICE 0
12
13 // PL: Zdefiniuj wlasne makra jezeli nie wykorzystujesz pliku makefile
14 //      Mozliwe kombinacje:
15 //          1.  USE_SEQUENTIAL + USE_C + FLOAT
16 //          2.  USE_SEQUENTIAL + USE_C + DOUBLE
17 //          3.  USE_SEQUENTIAL + USE_FORTRAN + FLOAT
18 //          4.  USE_SEQUENTIAL + USE_FORTRAN + DOUBLE
19 //          5.  USE_OPENMP + USE_C + FLOAT
20 //          6.  USE_OPENMP + USE_C + DOUBLE
21 //          7.  USE_OPENMP + USE_FORTRAN + FLOAT
22 //          8.  USE_OPENMP + USE_FORTRAN + DOUBLE
23 //          9.  USE_CUDA + FLOAT
24 //          10. USE_CUDA + DOUBLE
25
26
27 //# define USE_SEQUENTIAL
28 //# define USE_FORTRAN
29 //# define USE_C
30 //# define USE_OPENMP
31 //# define USE_CUDA
32
33 //# define FLOAT_PRECISION
34 //# define DOUBLE_PRECISION
35
36
37 # include <stdio.h>
38 # include <stdlib.h>
39 # include <math.h>
40
41 # include "../primitives/olib_primitives.h"
42 # include "../sp/olib_linear_algebra_sp.h"
43
44 # ifdef USE_OPENMP
45
46     # include "omp.h"
47     # include "../mp/olib_linear_algebra_openmp.h"
48
49 # endif
50
51 # ifdef USE_CUDA
52
53     # include "../gpgpu/olib_linear_algebra_cuda.h"
54
55 # endif
56
```

```
57
58 int main(int argc, char **argv)
59 {
60
61     clock_t time_start_total;
62     clock_t time_end_total;
63     double time_dif_total;
64
65     clock_t time_start_mv_multiplication;
66     clock_t time_end_mv_multiplication;
67     double time_dif_mv_multiplication;
68
69     time_dif_mv_multiplication = 0;
70
71
72     time_t time_start_total_seconds;
73     time_t time_end_total_seconds;
74
75     time_start_total_seconds = time(NULL);
76     time_start_total = clock();
77
78     srand( time(NULL) );
79
80
81     unsigned long long int N;
82     unsigned long long int n;
83     unsigned long long int step;
84     unsigned long long int steps;
85
86     unsigned int qubits_size;
87
88
89     # ifdef BATCH_MODE
90
91         if(argc == 2)
92         {
93             qubits_size = (unsigned int) atoi(argv[1]);
94         }
95         else
96         {
97             qubits_size = QBITS_SIZE;
98         }
99
100    # else
101
102        information();
103
104        printf(">> ");
105        scanf("%d", &qubits_size);
106
107    # endif
108
109
110    N = (unsigned long long int) pow((long double)2, (int) qubits_size);
111
112
113    # ifdef FLOAT_PRECISION
114
115        float random_value;
116
117        float *matrix_I;
```

```
118         matrix_I = (float *)malloc( N * N * sizeof(float) );
119
120         float *matrix_A;
121         matrix_A = (float *)malloc( N * N * sizeof(float) );
122
123         float *matrix_TEMP;
124         matrix_TEMP = (float *)malloc( N * N * sizeof(float) );
125
126         float w0[N];
127         float phi0[N];
128         float phi[N];
129         float vector_TEMP[N];
130
131     # endif
132
133
134     # ifdef DOUBLE_PRECISION
135
136         double random_value;
137
138         double *matrix_I;
139         matrix_I = (double *)malloc( N * N * sizeof(double) );
140
141         double *matrix_A;
142         matrix_A = (double *)malloc( N * N * sizeof(double) );
143
144         double *matrix_TEMP;
145         matrix_TEMP = (double *)malloc( N * N * sizeof(double) );
146
147         double w0[N];
148         double phi0[N];
149         double phi[N];
150         double vector_TEMP[N];
151
152     # endif
153
154
155
156     # ifdef USE_OPENMP
157
158         int number_of_threads;
159
160         #pragma omp parallel
161         number_of_threads = omp_get_num_threads();
162
163     # endif
164
165
166     n = N - 3;
167
168     steps = (unsigned long long int) floor(PI * 1.0 / (asin(sqrt(1.0 / N))) / 4);
169
170
171     # ifndef BATCH_MODE
172
173         printf("Total number of steps: %lld\n", steps);
174
175     # endif
176
177
178     matrix_identity_init_sp(matrix_I, N);
```

```
179
180
181     unsigned long long int i;
182
183     for(i=0; i < N; i++)
184     {
185         w0[i] = 0;
186     }
187     w0[n] = 1;
188
189
190     for(i=0; i < N; i++)
191     {
192         phi0[i] = 1.0 / sqrt(N);
193     }
194
195
196     matrix_outer_product_sp(w0, w0, matrix_TEMP, N);
197
198     # ifdef FLOAT_PRECISION
199         matrix_mul_scalar_sp(matrix_TEMP, (float) 2.0, N);
200     # endif
201
202     # ifdef DOUBLE_PRECISION
203         matrix_mul_scalar_sp(matrix_TEMP, (double) 2.0, N);
204     # endif
205
206     matrix_sub_sp(matrix_I, matrix_TEMP, matrix_A, N);
207
208
209     # ifdef FLOAT_PRECISION
210
211         float *matrix_B;
212         matrix_B = (float *)malloc( N * N * sizeof(float) );
213
214     # endif
215
216     # ifdef DOUBLE_PRECISION
217
218         double *matrix_B;
219         matrix_B = (double *)malloc( N * N * sizeof(double) );
220
221     # endif
222
223
224     matrix_outer_product_sp(phi0, phi0, matrix_TEMP, N);
225
226     # ifdef FLOAT_PRECISION
227         matrix_mul_scalar_sp(matrix_TEMP, (float) 2.0, N);
228     # endif
229
230     # ifdef DOUBLE_PRECISION
231         matrix_mul_scalar_sp(matrix_TEMP, (double) 2.0, N);
232     # endif
233
234     matrix_sub_sp(matrix_TEMP, matrix_I, matrix_B, N);
235
236     free(matrix_TEMP);
237
238
239
```

```
240     for(i=0; i < N; i++)
241     {
242         phi[i] = phi0[i];
243     }
244
245
246     step = 0;
247
248     //cudaSetDevice(1);
249
250     while(step <= steps)
251     {
252
253         # ifndef BATCH_MODE
254
255             printf("\n");
256             printf("Step number: %lld", step);
257             printf("\n");
258             printf("-------------------------------");
259             printf("\n");
260             printf("Probability of search success: %lf", pow(phi[n], 2));
261             printf("\n");
262
263         # endif
264
265
266         if(step < steps)
267         {
268
269             time_start_mv_multiplication = clock();
270
271             # ifdef USE_SEQUENTIAL
272
273                 # ifdef USE_C
274                     matrix_mul_vector_sp(matrix_A, phi, vector_TEMP, N);
275                     matrix_mul_vector_sp(matrix_B, vector_TEMP, phi, N);
276                 # endif
277
278                 # ifdef USE_FORTRAN
279
280
281                     # ifdef FLOAT_PRECISION
282
283                         matrix_mul_vector_sp_fortranf_(matrix_A, phi, vector_TEMP↩
                                , &N);
284                         matrix_mul_vector_sp_fortranf_(matrix_B, vector_TEMP, phi↩
                                , &N);
285
286                     # endif
287
288
289                     # ifdef DOUBLE_PRECISION
290
291                         matrix_mul_vector_sp_fortran_(matrix_A, phi, vector_TEMP,↩
                                &N);
292                         matrix_mul_vector_sp_fortran_(matrix_B, vector_TEMP, phi,↩
                                &N);
293
294                     # endif
295
296                 # endif
```

```
297
298             # endif
299
300             # ifdef USE_OPENMP
301
302                 # ifdef USE_C
303                     matrix_mul_vector_openmp(matrix_A, phi, vector_TEMP, N, ↩
                            number_of_threads);
304                     matrix_mul_vector_openmp(matrix_B, vector_TEMP, phi, N, ↩
                            number_of_threads);
305                 # endif
306
307                 # ifdef USE_FORTRAN
308
309                     # ifdef FLOAT_PRECISION
310
311                         matrix_mul_vector_openmp_fortranf_(matrix_A, phi, ↩
                                vector_TEMP, &N, &number_of_threads);
312                         matrix_mul_vector_openmp_fortranf_(matrix_B, vector_TEMP,↩
                                 phi, &N, &number_of_threads);
313
314                     # endif
315
316                     # ifdef DOUBLE_PRECISION
317
318                         matrix_mul_vector_openmp_fortran_(matrix_A, phi, ↩
                                vector_TEMP, &N, &number_of_threads);
319                         matrix_mul_vector_openmp_fortran_(matrix_B, vector_TEMP, ↩
                                phi, &N, &number_of_threads);
320
321                     # endif
322
323                 # endif
324
325             # endif
326
327             # ifdef USE_CUDA
328                 matrix_mul_vector_cuda(vector_TEMP, matrix_A, phi, N, 512, ↩
                        CUDA_DEVICE);
329                 matrix_mul_vector_cuda(phi, matrix_B, vector_TEMP, N, 512, ↩
                        CUDA_DEVICE);
330             # endif
331
332             time_end_mv_multiplication = clock();
333             time_dif_mv_multiplication += (double)(time_end_mv_multiplication -↩
                    time_start_mv_multiplication) / CLOCKS_PER_SEC;
334
335         }
336         step++;
337
338     }
339
340
341     free(matrix_A);
342     free(matrix_B);
343
344
345     # ifndef BATCH_MODE
346
347         printf("Print final quantum register |phi> state: \n");
348         printf("\t0 - No\n");
```

```
349         printf("\t1 - Yes\n");
350
351         printf(">> ");
352         int choice;
353         scanf("%d", &choice);
354
355         if(choice == 1)
356         {
357             printf("\n");
358             printf("Final quantum register |phi> state: \n");
359             printf_vector(phi, N);
360         }
361
362
363         printf("\nProbability of search success: ");
364         printf("\n%lf\n", phi[n]);
365
366     # endif
367
368
369     random_value = ( (double) rand() * rand() ) / ( (double) RAND_MAX * RAND_MAX ↩
            );
370
371
372     # ifndef BATCH_MODE
373
374         printf("\n");
375
376         if( random_value < pow(phi[n], 2) )
377         {
378             printf("MEASURMENT!!! -> SUCCESS");
379         }
380         else
381         {
382             printf("MEASURMENT!!! -> FAILURE");
383         }
384
385         printf("\n\n");
386
387     # endif
388
389
390     time_end_total = clock();
391     time_dif_total = (double)(time_end_total-time_start_total) / CLOCKS_PER_SEC;
392     time_end_total_seconds = time(NULL);
393
394
395     # ifdef BATCH_MODE
396
397         // DATA FORMAT:
398         // RESULT[0/1] RealTimeTotal SystemTimeTotal MVMSystemTime ↩
                NoMVMSystemTime
399
400         if( random_value < pow(phi[n], 2) )
401         {
402             printf("1 %ld %lf %lf\n", (time_end_total_seconds - ↩
                    time_start_total_seconds), time_dif_total, ↩
                    time_dif_mv_multiplication, (time_dif_total - ↩
                    time_dif_mv_multiplication));
403         }
404         else
```

```
405         {
406             printf("0 %ld %lf %lf\n", (time_end_total_seconds - ↩
                    time_start_total_seconds), time_dif_total, ↩
                    time_dif_mv_multiplication, (time_dif_total - ↩
                    time_dif_mv_multiplication));
407         }
408
409     # else
410
411         printf("\n↩
                ************************************************************\n");
412         printf("Total Execution time:\n");
413         printf("\tReal time: %ld seconds\n", (time_end_total_seconds - ↩
                time_start_total_seconds));
414         printf("\tSystem time: %lf seconds\n\n", time_dif_total);
415         printf("\n↩
                ************************************************************\n");
416         printf("Matrix-Vector Multiplication time:\n");
417         printf("\tSystem time: %lf seconds\n\n", time_dif_mv_multiplication);
418         printf("\n↩
                ************************************************************\n");
419         printf("No Matrix-Vector Multiplication time:\n");
420         printf("\tSystem time: %lf seconds\n\n", (time_dif_total - ↩
                time_dif_mv_multiplication));
421
422     # endif
423
424
425
426     return 0;
427 }
```

Listing 36. Funkcja główna main() dla algorytmu Grovera. Wersja z wywołaniami dla funkcji sekwencyjnych równoległych OPenMP oraz wrapperów CUDA.

D. Implementacja algorytmu Shora

```
1  /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
2   * shor_algorithm_boinc.cpp
3   *
4   * OLib
5   * High performance library of mathematical functions and algorithms
6   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
7
8
9  // EN: Define your own macros when not using makefile
10 //     The possible combinations:
11 //          1. USE_SEQUENTIAL + FLOAT_PRECISION
12 //          2. USE_SEQUENTIAL + DOUBLE_PRECISION
13 //          3. USE_CUDA + FLOAT_PRECISION
14 //          4. USE_CUDA + DOUBLE_PRECISION
15 //          5. USE_OPENCL + FLOAT_PRECISION
16 //          6. USE_OPENCL + DOUBLE_PRECISION
17
18 //# define USE_SEQUENTIAL
19 //# define USE_CUDA
20 //# define USE_OPENCL
21
22 //# define FLOAT_PRECISION
23 //# define DOUBLE_PRECISION
24
25 # define DEVICE_ID 0
26
27 # ifdef USE_SEQUENTIAL
28     # include <stdio.h>
29     # include <math.h>
30     # include <time.h>
31     # include <stdlib.h>
32
33
34     # include "../primitives/olib_primitives.h"
35     # include "../sp/olib_quantum_sp.h"
36     # include "../sp/olib_discrete_sp.h"
37     # include "../sp/olib_linear_algebra_sp.h"
38 # endif
39
40 # ifdef USE_CUDA
41     # include <stdio.h>
42     # include <math.h>
43     # include <time.h>
44     # include <stdlib.h>
45
46     # include "../primitives/olib_primitives.h"
47     # include "../sp/olib_quantum_sp.h"
48     # include "../sp/olib_discrete_sp.h"
49     # include "../sp/olib_linear_algebra_sp.h"
50     # include "../gpgpu/olib_quantum_cuda.h"
51 # endif
52
53 # ifdef USE_OPENCL
54     # include <stdio.h>
55     # include <math.h>
56     # include <time.h>
```

```
57      # include <stdlib.h>
58
59
60      # include "../primitives/olib_primitives.h"
61      # include "../sp/olib_quantum_sp.h"
62      # include "../sp/olib_discrete_sp.h"
63      # include "../sp/olib_linear_algebra_sp.h"
64
65      # include "../opencl/olib_quantum_ocl.h"
66  # endif
67
68  int shor_algorithm(unsigned long long int n, unsigned long long int max_tests)
69  {
70     clock_t time_start_total;
71     clock_t time_end_total;
72     double time_dif_total;
73
74     clock_t time_start_dqft;
75     clock_t time_end_dqft;
76     double time_dif_dqft;
77
78     time_dif_dqft = 0;
79
80     time_t time_start_total_seconds;
81     time_t time_end_total_seconds;
82
83     time_start_total_seconds = time(NULL);
84     time_start_total = clock();
85
86     unsigned long long int x = 0;
87
88     x = 1+ (unsigned long long int)((n-1)*(double)rand()/(double)RAND_MAX);
89
90     while (gcd_sp(n,x) != 1 || x == 1)
91     {
92        x = 1 + (unsigned long long int)((n-1)*(double)rand()/(double)RAND_MAX);
93     }
94
95     unsigned long long int q;
96     q = get_q_sp(n);
97
98     complex_double *quantum_register_1;
99     quantum_register_1 = (complex_double*) malloc(sizeof(complex_double) * (↩
           unsigned long long int) pow(2, reg_size_sp(q) - 1) );
100
101
102    for(unsigned long long int i = 0; i < pow(2, reg_size_sp(q) - 1); i++)
103    {
104       quantum_register_1[i].real = 0;
105       quantum_register_1[i].imag = 0;
106    }
107
108    unsigned long long int *modex;
109    modex = (unsigned long long int*) malloc(sizeof(unsigned long long int) * q );
110
111    complex_double *collapse;
112    collapse = (complex_double*) malloc(sizeof(complex_double) * q );
113
114    complex_double tmp;
115
116    complex_double *mdx;
```

```
117    mdx = (complex_double*) malloc(sizeof(complex_double) * (unsigned long long ↩
           int) pow(2,reg_size_sp(n)) );
118
119    for(unsigned long long int i = 0; i < pow(2,reg_size_sp(n)); i++)
120    {
121       mdx[i].real = 0;
122       mdx[i].imag = 0;
123    }
124
125    complex_double *quantum_register_2;
126    quantum_register_2 = (complex_double*) malloc(sizeof(complex_double) * (↩
           unsigned long long int) pow(2, reg_size_sp(n)) );
127
128    for(unsigned long long int i = 0; i < reg_size_sp(n); i++)
129    {
130       quantum_register_2[i].real = 0;
131       quantum_register_2[i].imag = 0;
132    }
133
134    unsigned long long int tmpval;
135    unsigned long long int value;
136
137    double c;
138    double m;
139
140    unsigned long long int den;
141    unsigned long long int p;
142
143    unsigned long long int e;
144    unsigned long long int a;
145    unsigned long long int b;
146    unsigned long long int factor;
147
148    int done = 0;
149    int tries = 0;
150
151    while (!done)
152    {
153       if(tries >= max_tests)
154       {
155          time_end_total = clock();
156          time_dif_total = (double)(time_end_total-time_start_total) / ↩
               CLOCKS_PER_SEC;
157          time_end_total_seconds = time(NULL);
158
159          printf_result(factor, done, tries, time_start_total_seconds, ↩
               time_end_total_seconds, time_dif_total, time_dif_dqft, 1);
160          return 1;
161
162       }
163
164       set_average_sp(quantum_register_1, reg_size_sp(q) - 1, q - 1);
165
166       tmp.real = 1.0;
167       tmp.imag = 0.0;
168
169       for (unsigned long long int i = 0 ; i < q ; i++)
170       {
171          tmpval = fast_modulo_power_sp(x,i,n);
172          modex[i] = tmpval;
173          mdx[tmpval] = complex_add(mdx[tmpval], tmp);
```

```
174         }
175
176         set_state_sp(quantum_register_2, pow(2, reg_size_sp(n)), mdx);
177         vector_normalization_sp(quantum_register_2, pow(2, reg_size_sp(n)));
178         value = dec_measure_sp(quantum_register_2, reg_size_sp(n));
179
180         for (unsigned long long int i = 0 ; i < q ; i++)
181         {
182            if (modex[i] == value)
183            {
184               collapse[i].real = 1;
185               collapse[i].imag = 0;
186            }
187            else
188            {
189               collapse[i].real = 0;
190               collapse[i].imag = 0;
191            }
192         }
193
194         set_state_sp(quantum_register_1, pow(2, reg_size_sp(q) - 1), collapse);
195
196         vector_normalization_sp(quantum_register_1, pow(2, reg_size_sp(n)) );
197
198         time_start_dqft = clock();
199
200         # ifdef USE_SEQUENTIAL
201
202            # ifdef FLOAT_PRECISION
203
204               complex_float *quantum_register_1_float;
205
206               quantum_register_1_float = (complex_float *)malloc( q * sizeof(↩
                      complex_float) );
207
208               copy_vector_sp(quantum_register_1_float, quantum_register_1, q);
209
210               dqft_sp(quantum_register_1_float, q);
211
212               copy_vector_sp(quantum_register_1, quantum_register_1_float, q);
213
214               free(quantum_register_1_float);
215
216            # endif // END FLOAT_PRECISION
217
218            # ifdef DOUBLE_PRECISION
219
220               dqft_sp(quantum_register_1, q);
221
222            # endif
223
224         # endif // END USE_SEQUENTIAL
225
226
227         # ifdef USE_CUDA
228
229            # ifdef FLOAT_PRECISION
230
231               complex_float *quantum_register_1_float;
232
233               quantum_register_1_float = (complex_float *)malloc( q * sizeof(↩
```

```
                complex_float) );
234
235             copy_vector_sp(quantum_register_1_float, quantum_register_1, q);
236
237             dqft_cuda(quantum_register_1_float, q, 512, DEVICE_ID);
238
239             copy_vector_sp(quantum_register_1, quantum_register_1_float, q);
240
241             free(quantum_register_1_float);
242
243         # endif // END FLOAT_PRECISION
244
245
246         # ifdef DOUBLE_PRECISION
247
248             dqft_cuda(quantum_register_1, q, 512, DEVICE_ID);
249
250         # endif // END DOUBLE_PRECISION
251
252
253     # endif // END USE_CUDA
254
255
256     # ifdef USE_OPENCL
257
258         # ifdef FLOAT_PRECISION
259
260             complex_float *quantum_register_1_float;
261
262             quantum_register_1_float = (complex_float *)malloc( q * sizeof(↩
                    complex_float) );
263
264             copy_vector_sp(quantum_register_1_float, quantum_register_1, q);
265
266             dqft_ocl(quantum_register_1_float, q, 512, DEVICE_ID);
267
268             copy_vector_sp(quantum_register_1, quantum_register_1_float, q);
269
270             free(quantum_register_1_float);
271
272         # endif // END FLOAT_PRECISION
273
274         # ifdef DOUBLE_PRECISION
275
276             dqft_ocl(quantum_register_1, q, 512, DEVICE_ID);
277
278         # endif // END DOUBLE_PRECISION
279
280     # endif // END USE_CUDA
281
282
283     time_end_dqft = clock();
284     time_dif_dqft += (double)(time_end_dqft -time_start_dqft) / CLOCKS_PER_SEC;
285
286     m = dec_measure_sp(quantum_register_1, reg_size_sp(q) - 1);
287
288     done = 1;
289
290     if (m == 0)
291     {
292         done = 0;
```

```
293         }
294
295         if (m == -1)
296         {
297            done = 0;
298         }
299
300         if (done)
301         {
302            c = (double)m  / (double)q;
303
304            den = denominator_sp(c, q);
305
306            p = (unsigned long long int)floor(den * c + 0.5);
307
308            if (den % 2 == 1 && 2 * den < q )
309            {
310               p = 2 * p;
311               den = 2 * den;
312            }
313
314            e = a = b = factor = 0;
315
316            if (den % 2 == 1)
317            {
318               done = 0;
319            }
320            else
321            {
322               e = fast_modulo_power_sp(x, den / 2, n);
323               a = (e + 1) % n;
324               b = (e + n - 1) % n;
325               factor = max(gcd_sp(n,a),gcd_sp(n,b));
326            }
327         }
328
329         if (factor == -1)
330         {
331            done = 0;
332         }
333
334         if ((factor == n || factor == 1) && done == 1)
335         {
336            done = 0;
337         }
338
339         if (factor == 0 && done != 1)
340         {
341            done = 0;
342         }
343
344         tries++;
345
346      }
347
348      time_end_total = clock();
349      time_dif_total = (double)(time_end_total-time_start_total) / CLOCKS_PER_SEC;
350      time_end_total_seconds = time(NULL);
351
352      printf_result(factor, done, tries, time_start_total_seconds, ↩
          time_end_total_seconds, time_dif_total, time_dif_dqft, 1);
```

```
353
354     free(quantum_register_1);
355     free(quantum_register_2);
356
357     free(collapse);
358     free(modex);
359     free(mdx);
360
361     return 0;
362 }
363
364
365
366 int main(int argc, char **argv)
367 {
368     srand(time(NULL));
369
370     unsigned long long int n;
371
372     unsigned long long int max_tests;
373     unsigned long long int number_of_iterations;
374
375     if(argc == 2)
376     {
377        n = (unsigned long long int) atol(argv[1]);
378
379        if(n == 9)
380        {
381           clock_t time_start;
382           clock_t time_end;
383           double time_dif;
384
385           unsigned long long int test_size;
386
387           # ifdef FLOAT_PRECISION
388              test_size = 16384;
389           # endif // END FLOAT_PRECISION
390
391           # ifdef DOUBLE_PRECISION
392              test_size = 16384*2;
393           # endif // END FLOAT_PRECISION
394
395           # ifdef FLOAT_PRECISION
396
397              complex_float *quantum_register;
398              quantum_register = (complex_float*) malloc(sizeof(complex_float) * (↩
                    unsigned long long int) pow(2, reg_size_sp(test_size) - 1) );
399
400              complex_float *init;
401              init = (complex_float*) malloc(sizeof(complex_float) * (unsigned long↩
                     long int) pow(2, reg_size_sp(test_size) - 1) );
402
403              complex_float tmpcomp;
404
405              float term_1;
406              term_1 = pow(test_size,-.5);
407
408              float term_2;
409
410              float sin_value;
411              float cos_value;
```

```
412
413         # endif // END FLOAT_PRECISION
414
415         # ifdef DOUBLE_PRECISION
416
417             complex_double *quantum_register;
418             quantum_register = (complex_double*) malloc(sizeof(complex_double) * ↩
                    (unsigned long long int) pow(2, reg_size_sp(test_size) - 1) );
419
420             complex_double *init;
421             init = (complex_double*) malloc(sizeof(complex_double) * (unsigned ↩
                    long long int) pow(2, reg_size_sp(test_size) - 1) );
422
423             complex_double tmpcomp;
424
425             double term_1;
426             term_1 = pow(test_size,-.5);
427
428             double term_2;
429
430             double sin_value;
431             double cos_value;
432
433         # endif // END DOUBLE_PRECISION
434
435         printf("------------------------------------------------------------\n");
436         printf("Start benchmark\n");
437         printf("\nPlease wait...\n");
438
439         time_start = clock();
440
441         tmpcomp.real = 0.0;
442         tmpcomp.imag = 0.0;
443
444         unsigned long long int a;
445         unsigned long long int c;
446
447         for (a = 0 ; a < test_size ; a++)
448         {
449             for (c = 0 ; c < test_size ; c++)
450             {
451                 term_2 = 2*PI*a*c/test_size;
452
453                 # ifdef FLOAT_PRECISION
454                     # ifdef __linux__
455                         sincosf(term_2, &sin_value, &cos_value);
456                     # else
457                         sin_value = sin(term_2);
458                         cos_value = cos(term_2);
459                     # endif
460                 # endif // END FLOAT_PRECISION
461
462
463                 # ifdef DOUBLE_PRECISION
464                     # ifdef __linux__
465                         sincos(term_2, &sin_value, &cos_value);
466                     # else
467                         sin_value = sin(term_2);
468                         cos_value = cos(term_2);
469                     # endif
470                 # endif // END DOUBLE_PRECISION
```

```
471
472                 tmpcomp.real = term_1 * cos_value;
473                 tmpcomp.imag = term_1 * sin_value;
474
475                 init[c] = complex_add(init[c], complex_mul(quantum_register[a], ↩
                        tmpcomp));
476             }
477         }
478
479         time_end = clock();
480
481         printf("\nEnd benchmark\n");
482
483         time_dif = (double)(time_end-time_start) / CLOCKS_PER_SEC;
484
485         printf("-----------------------------------------------------------\n");
486
487         printf("Execution time: %.5lf\n", time_dif);
488         printf("%lf MFLOPS", (((test_size * test_size * 15) / time_dif) / (1000 ↩
                * 1000)) );
489         printf("\n-----------------------------------------------------------\n");
490
491         free(quantum_register);
492         free(init);
493
494         return 1;
495
496     }
497
498 }
499
500 if(argc == 4)
501 {
502     n = (unsigned long long int) atol(argv[1]);
503     max_tests = (unsigned long long int) atol(argv[2]);
504     number_of_iterations = (unsigned long long int) atol(argv[3]);
505 }
506 else
507 {
508     return 1;
509 }
510
511 if (n%2 == 0)
512 {
513     return 1;
514 }
515
516 if (isPrime6k1_sp(n))
517 {
518     return 1;
519 }
520
521 if (TestPrimePower_sp(n))
522 {
523     return 1;
524 }
525
526 unsigned long long int actual_iteration;
527 fprintf (stderr, "\n");
528 int read_checkpoint;
529 read_checkpoint =checkpoint(&actual_iteration, 0); // Read checkpoint
```

```
530
531     if(read_checkpoint == 1)
532     {
533        actual_iteration = 0;
534     }
535
536     FILE *file_progress;
537
538     while(actual_iteration < number_of_iterations)
539     {
540        if (!(file_progress = fopen("progress", "wt")))
541        {
542           fprintf(stderr,"ERROR: CANNOT OPEN FILE PROGRESS\n");
543        }
544        else
545        {
546           if(actual_iteration == 0)
547           {
548              fprintf(file_progress, "0");
549           }
550           else
551           {
552              fprintf(file_progress, "%lf", ((double) actual_iteration / (double) ↩
                     25) );
553           }
554
555           fclose(file_progress);
556
557        }
558
559        shor_algorithm(n, max_tests);
560        actual_iteration++;
561        checkpoint(&actual_iteration, 1); // Save checkpoint
562     }
563
564     fprintf (stderr, "\n");
565
566     return 0;
567 }
```

Listing 37. Szkielet algorytmu Shora wraz z funkcją główną main() oraz funkcją shor_algorithm() odpowiedzialną za przeprowadzanie obliczeń. Wersja wykorzystana w obliczeniach na platformie BOINC dostosowana do obliczeń sekwencyjnych równoległych OpenMP oraz na kartach graficznych i innych urządzeniach (m. in. procesor Cell).

E. Spis wykorzystanych zasobów sprzętowych

Tab. E.1. Spis wykorzystanego sprzętu.

1:	Nazwa: Procesor(y): System: Wersja kernela: Dodatkowe akceleratory: Środowisko programistyczne:	AGH Cyfronet *'Zeus'* Intel Xeon L/X/E56XX Scientific Linux x.x.x - GCC 4.1.2, ICC 11.1, MPICH 1.2.7
2:		UMCS Tesla *'Nikola'* Intel Core i7 950 CentOS x.x.x nVidia Tesla C2050, nVidia GeForce GTS 250 bd
3:		Intel MTL Intel Xeon E7-4860 Redhat Enterprise Linux 6.4 x.x.x - GCC 4.1.2, ICC 11.1
4:		IBM CSDL IBM/S390 Redhat x.x.x - GCC 4.4.6
5:		PWSIP Blade QS22 IBM PowerXCell 8i Fedora x.x.x - GCC 4.3.0, OpenCL 1.1

Continued on next page

Tab. E.1 – continued from previous page

Tab. E.1 – continued from previous page		
6:	Nazwa: Procesor(y): System: Wersja kernela: Dodatkowe akceleratory: Środowisko programistyczne:	PWSIP Tesla Intel Xeon xxx Microsoft Windows 2003 Server x.x.x nVidia Tesla S1070 bd
7:		PWSIP *'Alfa'* Intel Core 2 Quad Q8200 Ubuntu Server 9.04 x.x.x nVidia GeForce 9800 GT GCC 4.4.5, CUDA 3.2
8:		PWSIP PlayStation 3 IBM Cell x x.x.x - x
9:		- AMD Athlon II X2 250, Intel Celeron G540, Intel Xeon E3-1225 - x.x.x - bd
10:		- AMD Athlon 64 X2 3800+ *'Manchester'* Mandriva 2011.0 x86-64 2.6.39 AMD Radeon HD 7770 bd

Continued on next page

Tab. E.1 – continued from previous page

Tab. E.1 – continued from previous page		
11:	Nazwa: Procesor(y): System: Wersja kernela: Dodatkowe akceleratory: Środowisko programistyczne:	- Intel Pentium D 820 - x.x.x nVidia GeForce 9800 GX2 bd
12:		- Intel i5-2400 - x.x.x - bd
13:		- Intel Atom N2800 - 3.2.13 - GCC 4.1.2
14:		- AMD Athlon 3500+ - x.x.x - bd
15:		Kristech KT-SBC-SAM9-1 ARM926EJ-S rev 5 (v5l) Debian *'Squeeze'* emDebian 2.6.39 - GCC 4.4.5

Continued on next page

Tab. E.1 – continued from previous page

Tab. E.1 – continued from previous page		
16:	Nazwa: Procesor(y): System: Wersja kernela: Dodatkowe akceleratory: Środowisko programistyczne:	Netgear Stora MS2110 ARM926EJ-S rev 1 (v5l) Debian *'Squeeze'* 2.6.33 - GCC 4.4.5
17:		Twister baseboard for TAM-3517 ARMv7 Processor rev 7 (v7l) The Angstrom Distribution 2.6.37 - GCC 4.4.5
18:		Raspberry Pi ARMv6-compatible processor rev 7 (v6l) Debian *'Wheezy'* 3.1.9 - GCC 4.6.3