NAUKI PODSTAWOWE
CZASOPISMO TECHNICZNE
TECHNICAL TRANSACTIONS
FUNDAMENTAL SCIENCES
WYDAWNICTWO POLITECHNIKI KRAKOWSKIEJ
1-NP/2011
ZESZYT 22
ROK 108
ISSUE 22
YEAR 108

JOANNA PŁAŻEK, MICHAŁ PODYMA*

RÓWNOLEGŁE ŁAMANIE HASEŁ METODĄ SŁOWNIKOWĄ W ŚRODOWISKACH MPI, OPENMP I CUDA

PARALLEL PASWORD CRACKING BY DICTIONARY METHOD IN MPI, OPENMP AND CUDA ENVIRONMENTS

Streszczenie

Algorytm łamania haseł metodą słownikową jest techniką używaną do siłowego odgadywania haseł do systemów. Jego realizacja wymaga dużych nakładów obliczeniowych, dlatego też uzasadnione jest wykorzystanie do tego celu programowania równoległego. W artykule przedstawiono i porównano ze sobą, równoległe implementacje tego algorytmu w trzech różnych środowiskach programowania równoległego. Są nimi *MPI* (*Message Passing Interface*), środowisko wykorzystujące model z przesyłaniem komunikatów, *OpenMP* (*Open Multi Processing*), realizujące zrównoleglenie obliczeń na poziomie danych oraz środowisko procesorów kart graficznych.

Słowa kluczowe: *atak słownikowy, programowanie równoległe, środowiska programowania równoległego*: *MPI, OpenMP, CUDA*

Abstract

Password cracking by the dictionary method is a technique for detecting passphrases. Its implementation requires a large computational effort, so it is justified to apply for it a parallel programming. In this article we describe and compare a parallel implementation in three different parallel programming environments, i.e. *MPI* (Message Passing Interface), the environment that uses a model with message-passing; *OpenMP* (Open Multi Processing), which is based on data-parallelization; and the graphics processing units environment.

Keywords: dictionary attack, parallel programming, parallel programming environments: *MPI, OpenMP, CUDA*

* Dr inż. Joanna Płażek, inż. Michał Podyma, Instytut Teleinformatyki, Wydział Fizyki, Matematyki i Informatyki, Politechnika Krakowska.

1. Wstęp

Większość dzisiejszych komputerów osobistych posiada procesor wielordzeniowy, a także kartę graficzną oferującą środowisko programistyczne pozwalające na wykonywanie na niej obliczeń numerycznych. Tym samym każdy z tych komputerów posiada duże możliwości i moc obliczeniową jeszcze niedawno dostępną tylko dla firm i instytucji potrzebujących dużej ilości zasobów do wykorzystania w celach komercyjnych bądź badawczych. Większość aplikacji używanych na co dzień jest w stanie pracować jedynie w trybie sekwencyjnym. Istnieje jednak wiele programów potrzebujących długiego czasu realizacji. Użycie technologii programowania równoległego, wykorzystującego w pełni moc komputerów wielordzeniowych czy kart graficznych, mogłoby zmniejszyć ten czas kilka- lub nawet kilkunastokrotnie.

Łamanie haseł jest niewątpliwie zadaniem wymagającym dużych nakładów obliczeniowych. Do niedawna przyspieszenie tego typu obliczeń uzyskiwano głównie dzięki zastosowaniu coraz nowszych procesorów o ciągle zwiększającej się liczbie wykonywanych cykli na sekundę. F. Alonso w swoim artykule *The Extinction of Password Authentication* [1] przedstawia ewolucję mocy procesorów na przestrzeni lat 1971–2008 i zagrożenia wynikające ze znacznego skrócenia czasu łamania „mocnych" haseł. Kolejnym krokiem w celu zwiększenia efektywności przetwarzania było zastosowanie procesorów wielordzeniowych, by w końcu łamać hasła przy użyciu maszyn wieloprocesorowych lub multikomputerów.

Ostatnio prawdziwą rewolucją stało się wykorzystanie procesorów kart graficznych do obliczeń ogólnego przeznaczenia, czyli GPGPU. Wyniki porównujące czasy łamania haseł z wykorzystaniem CPU i środowiska GPGPU dla haseł różnej długości, kodowanych przy pomocy różnych algorytmów, przedstawia Martijn Sprengers w pracy *GPU-based Password Cracking. On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units* [2].

Do najczęściej wykorzystywanych środowisk programowania równoległego można zaliczyć MPI (ang. *Message Passing Interface*), OpenMP (ang. *Open Multi Processing*) oraz środowisko CUDA. Porównanie równoległych implementacji algorytmu łamiącego hasła, realizowanych w różnych środowiskach obliczeń równoległych, daje możliwość odpowiedzi na pytanie, które ze środowisk pozwala na uzyskanie największego przyspieszenia i jakim nakładem pracy każda z implementacji musi być realizowana.

2. Zabezpieczenie i łamanie haseł

Hasło jest elementem bezpieczeństwa pozwalającym zabezpieczyć dane przed nieautoryzowanym dostępem przez osoby postronne. Używane jest w celu uwierzytelnienia tożsamości osoby pragnącej uzyskać dostęp do pewnych zasobów, które chronione są przed dostępem osób nieupoważnionych do korzystania z nich. Mogą to być prywatne dane jednej osoby i dostępne tylko dla niej samej, jak na przykład dostęp do konta bankowego. Innym przykładem są dane współdzielone przez grupę osób i jednocześnie chronione przed każdym spoza danej grupy, jak dane osób pracujących nad jednym projektem w firmie.

Najczęstsze przechwycenie hasła przez osoby do tego nieuprawnione odbywa się w następujących miejscach:

– **Serwer uwierzytelniania**. Miejsce, w którym znajdują się dane użytkownika potrzebne do jego identyfikacji i uwierzytelnienia. Znajduje się tam hasło, aby można je było porównać z tym, które jest podawane w celu uzyskania dostępu;
– **Medium transmisyjne**. W celu uzyskania dostępu do zasobów zdalnych hasło zostaje wysłane do serwera uwierzytelniania, co daje możliwość przechwycenia go;
– **Komputer użytkownika**. Często dla wygody hasło zostaje zapisane na komputerze użytkownika, aby oszczędzić konieczność wpisywania go przy każdorazowym logowaniu. Usługi takie oferują przeglądarki internetowe oraz inne programy, np. jak komunikatory internetowe.

2.1. Szyfrowanie haseł

Przechowywanie i transmisja hasła w postaci ciągu znaków bez żadnych dodatkowych zabezpieczeń to działanie do niedawna dość powszechnie stosowane. Podczas logowania do serwera pocztowego w sieci publicznej użytkownik jest narażony na przechwycenie jego danych. W tym celu osobie przechwytującej wystarczy zaopatrzenie się w najprostszy *sniffer* (program lub urządzenie umożliwiające przechwytywanie danych przepływających w sieci), aby przechwycić login i hasło osoby logującej.

W celu zabezpieczenia hasła przed odczytaniem z miejsca przechowywania bądź przechwyceniem podczas transmisji stosowane są dwa zabiegi:
– **Szyfrowanie transmisji**. Hasło może być przesyłane w postaci ciągu znaków, a ze względu na szyfrowanie całej transmisji danych pozostaje bezpieczne;
– **Szyfrowanie hasła**. Praktycznie nigdy hasło nie zostaje zapisane w postaci ciągu znaków. Przed zapisaniem zostaje zaszyfrowane bądź poddane działaniu funkcji hashującej co czyni proces odtworzenia hasła trudnym i czasochłonnym. Czasem dla zwiększenia efektywności hasło jest hashowane, szyfrowane kilkukrotnie lub wcześniej modyfikowane.

2.2. Dodatkowe modyfikacje

Ogólną zasadą tworzenia mocnego hasła odpornego na zgadnięcie jest utworzenie ciągu znaków zawierającego przypadkową kombinację liter, liczb i innych dozwolonych znaków. Niestety takie hasło zwykle trudno zapamiętać i łatwo się pomylić przy jego wpisywaniu. Ponadto mało kto bierze pod uwagę fakt, że ktoś mógłby zechcieć poznać jego hasło, a jego wpisywanie brane jest często za uciążliwy obowiązek. Czasami użytkownicy są zmuszeni do zabezpieczenia się silnym hasłem, np. w przypadku dostępu do danych firmowych, gdzie polityka bezpieczeństwa firmy tego wymaga. Jeśli jednak użytkownik nie widzi takiej konieczności, zabezpiecza swoje konto hasłem, które łatwo zapamięta i będzie wygodne w użyciu, szczególnie jeśli często go używa. Takie proste hasła to zwykle imiona, daty urodzenia bądź przypadkowe słowa zdające się być trudnymi do zgadnięcia właśnie ze względu na swoją przypadkowość. W celu zmniejszenia prawdopodobieństwa zgadnięcia hasła stosowane są zabiegi modyfikujące hasło przez użytkownika, bądź też modyfikacje zapisanego hasła dokonywane przez program je zapamiętujący:

a) **Modyfikacje użytkownika**. Jeśli użytkownik nie posiada silnego hasła powinien je zmienić chociażby przez jego niewielką modyfikację. Taki zabieg nie musi uczynić hasła trudniejszym do zapamiętania lub niewygodnym. Jeśli hasło jest np. imieniem to prosta modyfikacja polegająca na dopisaniu do niego liczby zwiększa jego moc. Zawarcie w haśle znaku specjalnego jest jeszcze skuteczniejsze.

Przykład:

hasło: „kasienka” zamieniamy na „kasienka.5”. Takie hasło możemy już traktować jako silne. Niektóre portale pozwalają sprawdzić siłę hasła przed założeniem konta, jak np. Gmail (*www.gmail.com*) podczas zakładania konta pocztowego. Kiedy wpisujemy hasło do formularza rejestracyjnego, jesteśmy informowani o jego sile. W przypadku naszego powyższego przykładu hasło „kasienka” zostało uznane jako słabe, a po dopisaniu do niego cyfry bądź kropki hasło miało już status silnego.

b) **Modyfikacje zapisywanych haseł**. Hasła na serwerach oraz ich kopie na komputerach użytkownika nie są zapisywane w postaci ciągu znaków, lecz zostają przetworzone funkcją hashującą. Jednak często przed hashowaniem ciąg zawierający nasze hasło jest modyfikowany. Zabieg ten jest podwójnie skuteczny:
   – utrudnia złamanie hasła – zmodyfikowany ciąg jest zwykle dłuższy od pierwotnego hasła co zwiększa czas poszukiwań, a jeśli hasło jest słabe, modyfikacje zwiększają jego siłę, przez co nie każdy program łamiący jest w stanie odgadnąć hasło,
   – znajduje niepoprawne hasło – nawet jeśli hash zostanie rozkodowany i znaleziony zostanie zakodowany ciąg znaków, to ciągle nie wiadomo jakie zabiegi zostały na nim wcześniej przeprowadzone i może się okazać, że odgadnięte hasło nie jest poprawne, a brak wiedzy o wcześniejszych modyfikacjach uniemożliwi jego odgadnięcie.

   Przykład:

   Dla użytkownika *user* i jego hasła *pass* tworzony jest ciąg wg. schematu:

   login + *salt* + hasło

   gdzie *salt* jest określonym przez program przypadkowym ciągiem.
   Przyjmijmy *salt* = 0123456789.
   Dostaniemy więc:

   user0123456789pass

   Otrzymany ciąg poddajemy działaniu funkcji hashującej. W przypadku algorytmu MD5 da ona wynik:

   c2418f49024b3908eb9ae66de66cd884

   Hasło przetrzymywane w takiej postaci w niczym nie przypomina prawdziwego hasła i wymaga szerokiej wiedzy oraz wiele pracy, aby znaleźć prawdziwy ciąg znaków reprezentujący zakodowane hasło.

2.3. Łamanie haseł

Łamanie haseł nie musi się wiązać z łamaniem prawa. Najczęstszym przykładem zupełnie legalnego łamania haseł jest sytuacja, kiedy użytkownik chce odzyskać zapomniane hasło. Istnieją oczywiście różne mechanizmy zapobiegające sytuacjom utraty hasła, ale czasem użytkownicy nie są przygotowani na taka ewentualność i hasło przepada. Programy do łamania haseł dają ponadto użytkownikom wiedzę, że ich konta nigdy nie są w pełni bezpieczne i pozwalają określić jak złożone powinno być hasło, aby jego złamanie trwało jak najdłużej oraz ustalić, z jaką częstotliwością należy hasło zmieniać, aby ustrzec się przed utratą danych. Istnieją różne metody odnajdywania hasła. Poniżej zostaną przedstawione dwie najbardziej popularne.

2.3.1. Metoda *bruteforce*

Metoda *bruteforce* jest jedyną pewną metodą łamania haseł. Gwarancję tą trzeba jednak opłacić czasem wykonania tej operacji. Metoda przewiduje porównanie ze znalezionym hasłem każdego ciągu znaków akceptowanego jako hasło. Ponieważ łamane hasło jest zapisane w zakodowanej postaci, każde z potencjalnych haseł jest przetwarzane tą samą metodą i wynik tej operacji zostaje porównany z zapisanym ciągiem znaków. Jeśli te dwa zakodowane hasła są jednakowe, znaczy to, że hasło zostało znalezione. Metoda ta wymaga wcześniejszego zapoznania się ze sposobem zapisania hasła przez program je zapamiętujący. Jeśli nie będzie znany algorytm, jakim zostało zakodowane hasło, odgadniecie go nie będzie możliwe. Czasami producenci oprogramowania jawnie udostępniają informacje o sposobie oraz miejscu zapisu hasła, tym samym dając możliwość rozkodowania go. Nie jest to sytuacja zachęcająca do łamania haseł, bo hasła zapisane są na prywatnych komputerach, do których osoby postronne nie mają dostępu. Jeśli taki dostęp jest możliwy, hasła powinny zostać dodatkowo zabezpieczone lub dostęp do komputera uniemożliwiony.

2.3.2. Metoda „słownikowa”

Metoda „słownikowa” posiada cechy wspólne z metodą *bruteforce*. Zasada jej działania jest taka sama, czyli polega na porównywaniu kolejnych słów z hasłem, które ma zostać złamane. Nie jest to jednak metoda gwarantująca złamanie hasła. Przy zastosowaniu tej metody nie zostają porównane wszystkie możliwe hasła, a jedynie zawarte w dołączonym słowniku. Metoda ta wykorzystuje słabość haseł, ponieważ większość z nich nie jest hasłami silnymi i możemy je znaleźć w zwykłym słowniku językowym. Często stosowane są modyfikacje tych słów. Już po dodaniu do hasła jednego znaku nie będącego literą użytkownik jest zapewniany o mocy swojego hasła. Wykorzystanie tego faktu i analogiczne działanie na słowach zawartych w słowniku automatycznie zwiększa pojemność słownika oraz pozwala nam na zwiększenie skuteczności wyszukiwania. Program łamiący hasła metodą „słownikową” może być bardzo skuteczny, jeśli uwzględni się w nim zwyczaje użytkowników podczas wymyślania haseł. Badania wykazują, że istnieją hasła, które powtarzają się zdecydowanie częściej niż inne. Hasła jak „123456”, „qwerty” czy „password” są tak popularne, że udział jednego słowa w liczbie wszystkich haseł jest w stanie przekroczyć 1%. Tym samym program zawierający dobrze dobrany słownik i umiejętnie nim operujący jest w stanie złamać zdecydowaną większość haseł przy niewielkim nakładzie pracy w porównaniu do podobnego programu łamiącego hasła metodą *bruteforce*.

3. Wybrane środowiska programowania równoległego

Opracowana aplikacja do łamania haseł wykorzystuje algorytm oparty na słowniku. Została zaimplementowana w trzech środowiskach programowania równoległego: MPI (ang. *Message Passing Interface*), OpenMP (ang. *Open Multi Processing*) oraz w środowisku karty graficznej. Poniżej przedstawiono ogólny opis tych środowisk oraz bardziej szczegółowo funkcje wykorzystane w programie.

3.1. Środowisko MPI

MPI (ang. *Message Passing Interface*) jest standardem interfejsu do przesyłania komunikatów na potrzeby programowania rzeczywistego na maszynach z pamięcią lokalną.

Pierwsza implementacja została przedstawiona w maju 1994 roku przez konsorcjum MPI Forum – grupę badaczy z USA i Europy, reprezentujących producentów oraz użytkowników maszyn równoległych. MPI nie jest konkretnym pakietem oprogramowania, a formalną specyfikacją interfejsu. Najbardziej znaną implementacja MPI jest MPICH. Interfejs MPI dostarcza funkcje umożliwiające odbieranie oraz wysyłanie komunikatów i synchronizację zadań wykonywanych na różnych komputerach (procesorach). Jego zaletą jest fakt, że program może być wykonywany na maszynach o różnych architekturach. MPI zupełnie rezygnuje z koncepcji pamięci dzielonej. Każdy proces, nawet uruchomiony na tym samym procesorze, posiada swoją własną kopię wszystkich danych. Jedyną drogą komunikacji i wymiany danych miedzy procesami są komunikaty, które służą nie tylko do synchronizacji i wysyłania informacji kontrolnych, a głównie do wymiany danych między procesami [3, 4].

3.1.1. Podstawowe funkcje użyte w programie

Rozpoczęcie pracy:

*int MPI_Init(int *argc, char **argv[])*

Funkcja inicjalizuje środowisko pracy i rozpoczyna pracę z biblioteką MPI. Funkcja m.in. tworzy domyślny komunikator *MPI_COMM_WORLD*. Dopiero po jej wywołaniu możemy korzystać z pozostałych funkcji MPI.

Zakończenie pracy:

int MPI_Finalize(void)

Funkcja kończy pracę z biblioteką MPI i zwalnia używane przez nią zasoby przygotowując system do zamknięcia.

Identyfikator procesu:

*int MPI_Comm_rank(MPI_Comm comm, int *rank)*

Funkcja pod adresem *rank* zapisuje identyfikator wywołującego ją procesu w obrębie komunikatora *comm*.

Ilość procesów:

*int MPI_Comm_size(MPI_Comm comm, int *size)*

Funkcja pod adresem *size* zapisuje ilość procesów uruchomionych w obrębie komunikatora *comm*.

Wysyłanie komunikatu:

*int MPI_Send(void *msg, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

Funkcja wysyła komunikat znajdujący sie pod adresem *msg* do procesu o identyfikatorze *dest*. Komunikacja odbywa się w obrębie komunikatora *comm*. Przesyłane dane są typu *datatype* i wielkości *count*, a komunikat jest oznaczony etykietą *tag*.

*int MPI_Isend(void *msg, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)*

Nieblokująca wersja funkcji *MPI Send()*. Program nie oczekuje na zakończenie działania funkcji.

Odbieranie komunikatu:

*int MPI_Recv(void *msg, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)*

Funkcja czeka na wysłanie przez proces o identyfikatorze *source* za pomocą komunikatora *comm* komunikatu z etykieta *tag*. Wartość *datatype* musi być identyczna jak w komunikacie wysyłanym. Komunikat nie większy niż *count* jest zapisywany pod adresem *msg*, a status operacji do argumentu *status*.

*int MPI_Irecv(void *msg, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *request)*

Nieblokująca wersja funkcji *MPI Recv().* Program nie oczekuje na zakończenie działania funkcji.

Wysyłanie komunikatu do grupy procesów:

*int MPI_Bcast(void *msg, int count, MPI_Datatype datatype, int root, MPI_Comm comm)*

Funkcja wysyła komunikat do grupy procesów. Wysyłającym jest proces *root*, a wszystkie pozostałe procesy w obrębie komunikatora *comm* odbierają komunikat odczytany z adresu *msg* procesu *root* zapisywany pod adresem *msg* innych procesów. Typ i wielkość komunikatu to *datatype* i *count*.

Dzielenie danych miedzy procesami:

*int MPI_Scatter(void *send_buf, int send_count, MPI_Datatype send_type, void *recv_buf, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm)*

Funkcja dzieli dane typu *send_type* umieszczone pod adresem *send* procesu *root* na części wielkości *send_count*. Dane zostają rozsyłane do wszystkich procesów, które odbierają swoją część danych typu *recv_type* umieszczając pod adresem *recv_buf* o wielkości *recv_count*. Komunikacja odbywa się w obrębie komunikatora *comm*.

Bariera:

int MPI_Barrier(MPI_Comm comm);

W miejscu wywołania funkcji program zatrzymuje swoje działanie i oczekuje aż wszystkie jego instancje „dotrą” do tego miejsca i wtedy wszystkie zaczynają dalszą pracę.

3.2. OpenMP

OpenMP (ang. *Open Multi Processing*) to standard służący do tworzenia aplikacji równoległych na komputerach z pamięcią wspólną. Opracowany w latach 90-tych XX wieku przez największych producentów maszyn równoległych, a następnie przyjęty przez wszystkich producentów oprogramowania. Możliwe jest programowanie w środowiskach obsługujących standard OpenMP w systemach Unix/Linux, a od Visual Studio 2005 również Windows. Głównym filarem standardu są dyrektywy zrównoleglające. Dodatkowymi elementami są zmienne środowiskowe oraz biblioteka procedur, służących głównie do manipulacji tymi zmiennymi. Ponadto procedury pozwalają mierzyć czas oraz identyfikować maszyny [5].

Zmienne środowiskowe

OMP SHEDULE

– określa sposób rozdzielania iteracji pomiędzy dostępne wątki, przyjmuje wartości: *static, dynamic, Guidem*

OMP DYNAMIC

– określa czy środowisko uruchomieniowe może zmienić ilość wątków, aby lepiej wykorzystać zasoby, przyjmuje wartości: *true, false*

OMP NUM THREADS

– określa ilość wątków wykorzystywanych przez program

OMP NESTED

– określa czy możliwe jest zagnieżdżanie pętli, przyjmuje wartości: *true, false*

Dyrektywy w języku C

Definicja bloku równoległego:

#pragma omp parallel [klauzule]
[instrukcje]

Po tej dyrektywie następuje blok instrukcji, którego wykonanie ma zostać zrównoleglone. Jako klauzule przy wywołaniu tej dyrektywy możemy zapisać:

private([lista zmiennych lokalnych])

– zadeklarowane zmienne zostaną zaalokowane dla każdego wątku z osobna, każdy będzie miał swoją prywatną kopię i brak dostępu do kopii należących do innych wątków,

shared([lista zmiennych globalnych])

– zadeklarowane zmienne będą współdzielone przez wszystkie wątki wykonujące zadeklarowany blok instrukcji,

default(shared/none)

– jeśli *shared* to wszystkie niezadeklarowane zmienne będą globalne, a jeśli *none* to wszystkie zmienne muszą zostać jawnie zadeklarowane jako globalne lub lokalne,

num threads([liczba])

– zadeklarowanie liczby wątków dla zrównoleglonego bloku instrukcji.

Zrównoleglenie pętli:

#pragma omp for [klauzule] [petla for]

Z dyrektywą można użyć klauzul wymienionych dla dyrektywy *parallel* oraz m.in.

schedule(sposób rozdziału iteracji, [kwant])

– klauzulą tą możemy w pewnym stopniu kontrolować sposób rozdzielania iteracji pomiędzy dostępne wątki, pierwszy parametr definiuje sposób przydzielania zbioru iteracji do wątków, a parametr drugi określa wielkość zbiorów iteracji.

3.3. CUDA

CUDA (ang. *Compute Unified Device Architecture*) to opracowana przez firmę NVIDIA, równoległa architektura obliczeniowa, która zapewnia radykalny wzrost wydajności obliczeń dzięki wykorzystaniu mocy układów GPU (ang. *Graphics Processing Unit*) [6]. Jest środowiskiem ogólnych celów obliczeniowych wykonywanych na kartach graficznych bądź specjalizowanych kartach zbudowanych na ich bazie czyli dla obliczeń GPGPU (ang. *General – Purpose computing on Graphics Processing Units*). Do niedawna procesory kart graficznych można było wykorzystywać do obliczeń numerycznych tylko za pośrednictwem API (ang. *Application Programming Interface*) dla grafiki komputerowej jak OpenGL czy DirectX. Proces zrównoleglania w tej technologii polega na wykonywaniu jednocześnie przez wiele wątków tych samych partii kodu zwanych jądrami (ang. *kernel*), czyli funkcji opatrzonych kwalifikatorem global. Ilość wątków, którym przypisane jest to samo jądro, jest zdefiniowana przy jego wywołaniu. Wątki tworzą bloki, a te z kolei składają się na gridy[7, 8].

Wykonanie jądra konfigurujemy przy pomocy potrójnych nawiasów trójkątnych w sposób następujący:

Funkcja<<<liczbaBloków, liczbaWatków>>>([parametry])

gdzie:

Funkcja – nazwa jądra,
liczbaBloków – ilość bloków w gridzie,
liczbaWatków – ilość wątków w bloku,
parametry – parametry wywołania jądra.

Ze względu na możliwości techniczne operacje wykonywane na karcie graficznej posiadają szereg ograniczeń ściśle związanych z architekturą konkretnego procesora graficznego. Ograniczone są wielkości bloków oraz gridów więc nie możemy uruchomić nieskończonej ilości wątków, a we wszystkich obecnie produkowanych układach jądro może wykonać maksymalnie 2 miliony operacji.

Kwalifikatory funkcji:

__host__

Funkcja poprzedzona kwalifikatorem *host* jest wykonywana na CPU.

__global__

Funkcja poprzedzona kwalifikatorem *global* jest wywoływana z CPU, ale wykonywana na GPU.

__device__

Funkcja poprzedzona kwalifikatorem *device* jest wywoływana z GPU oraz wykonywana na GPU.

__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)

Kwalifikator *launch_bounds()* jest wykorzystywany w sąsiedztwie kwalifikatora *global*. W nawiasie podaje się dwa parametry:

maxThreadsPerBlock – maksymalna ilość wątków w bloku,
minBlocksPerMultiprocessor – minimalna ilość bloków na jednym procesorze.

Kwalifikator ten dla podanych parametrów wymusza na etapie kompilacji optymalizację ilości wykorzystywanych 32-bitowych rejestrów przez funkcje jądra. Ilość rejestrów dla jednego procesora jest ograniczona i niewielka, dlatego tak ważne jest ich rozsądne rozdysponowanie. Użycie tego kwalifikatora podczas kompilacji informuje nas o ilości wykorzystywanych rejestrów i o tym, czy ich ilość została zmniejszona, czy może podane parametry są zbyt duże. Proces optymalizacji może zawierać w sobie m.in. zwiększenie ilości wykonywanych operacji jeśli może się to przyczynić do zmniejszenia ilości używanych rejestrów.

Podstawowe funkcje

*cudaError_t cudaMalloc (void ** devPtr, size_t size)*

Funkcja alokuje miejsce w pamięci karty graficznej dostępnej za pomocą wskaźnika *devPtr*. Rozmiar zarezerwowanej pamięci to *size* bajtów.

*cudaError_t cudaMemcpy (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)*

Funkcja kopiuje dane o rozmiarze *count* z *src* do *dst*. Należy zdefiniować, czy miejsce źródłowe oraz docelowe znajduje się w pamięci komputera, czy też w pamięci karty graficznej, służą do tego zdefiniowane stałe:

cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost oraz *cudaMemcpyDeviceToDevice*.

*cudaError_t cudaFree (void * devPtr)*

Wywołanie funkcji zwalnia używane zasoby alokowane wcześniej pod adresem *devPtr*.

4. Implementacje algorytmów łamania haseł metodą słownikową

Na potrzeby aplikacji został przyjęty sposób zapisu haseł pod postacią:

login|hash

gdzie *hash* jest wynikiem funkcji hashującej na ciągu znaków reprezentującym hasło. Dla naszego przypadku zostały przyjęte modyfikacje hasła przed poddaniem go funkcji hashującej.

Po modyfikacjach wygląda ono następująco:

login0123456789haslo

Powyższy ciąg znaków jest podawany jako argument dla funkcji hashującej. Plik z hasłem zawiera więc następującą zawartość:

login|hash(login0123456789haslo)

Powyższy opis charakteryzuje system zapisu haseł, dla którego możliwe jest złamanie hasła opisywanym programem.

Do uproszczenia przebiegu testów plik z hasłami został nieco zmodyfikowany i zawiera kolejne pary: *login/haslo*. Program przed rozpoczęciem procesu łamania hasła najpierw poddaje je przetworzeniu funkcji hashującej stosując przedstawioną wyżej metodę. Po tym zabiegu program posiada zakodowane hasło i może przystąpić do jego łamania. Jako funk-

cję hashującą został wykorzystany algorytm MD5. Wykorzystana została jego darmowa implementacja udostępniona przez firmę Aladdin Enterprises [9]. Jako baza słów użyty został słownik języka polskiego do gier słownych zawierający ponad 2,7 mln słów [10].

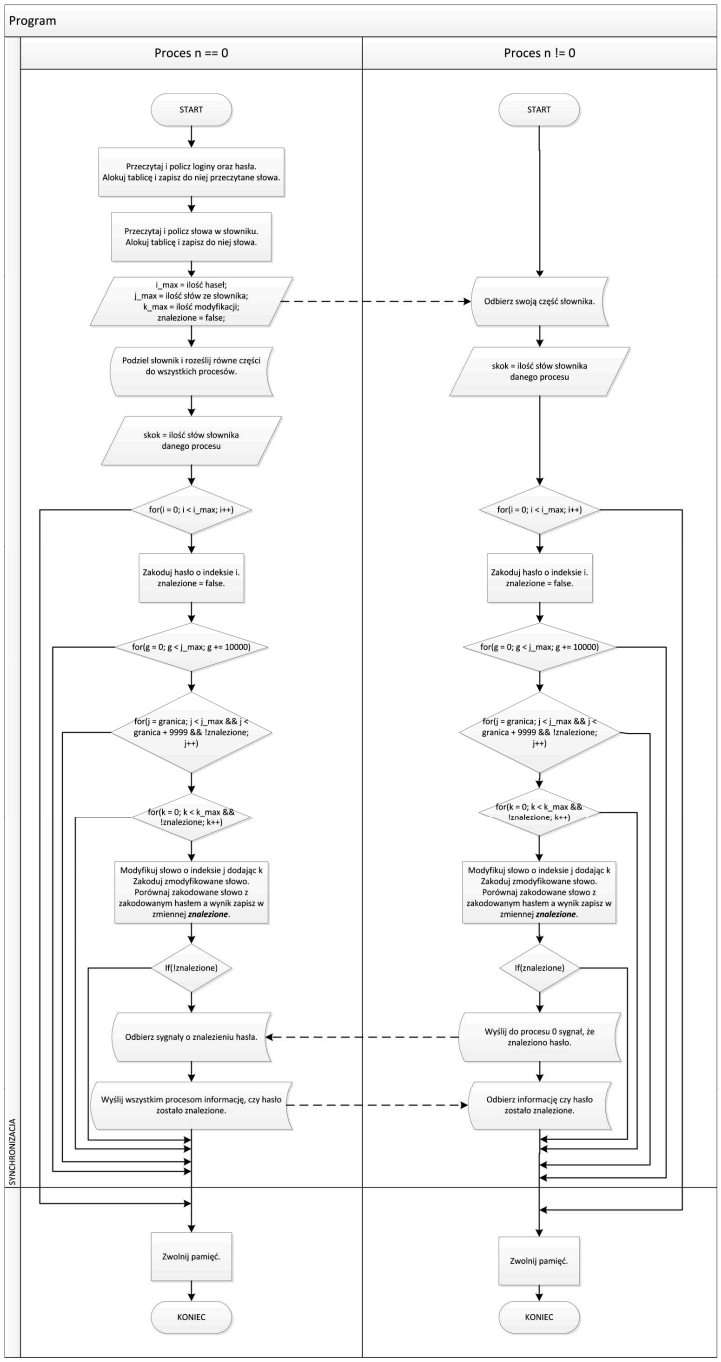
Wszystkie programy zostały zaimplementowane i uruchomione w programie Microsoft Visual Studio 2008 w języku programowania C.

4.1. Implementacja sekwencyjna

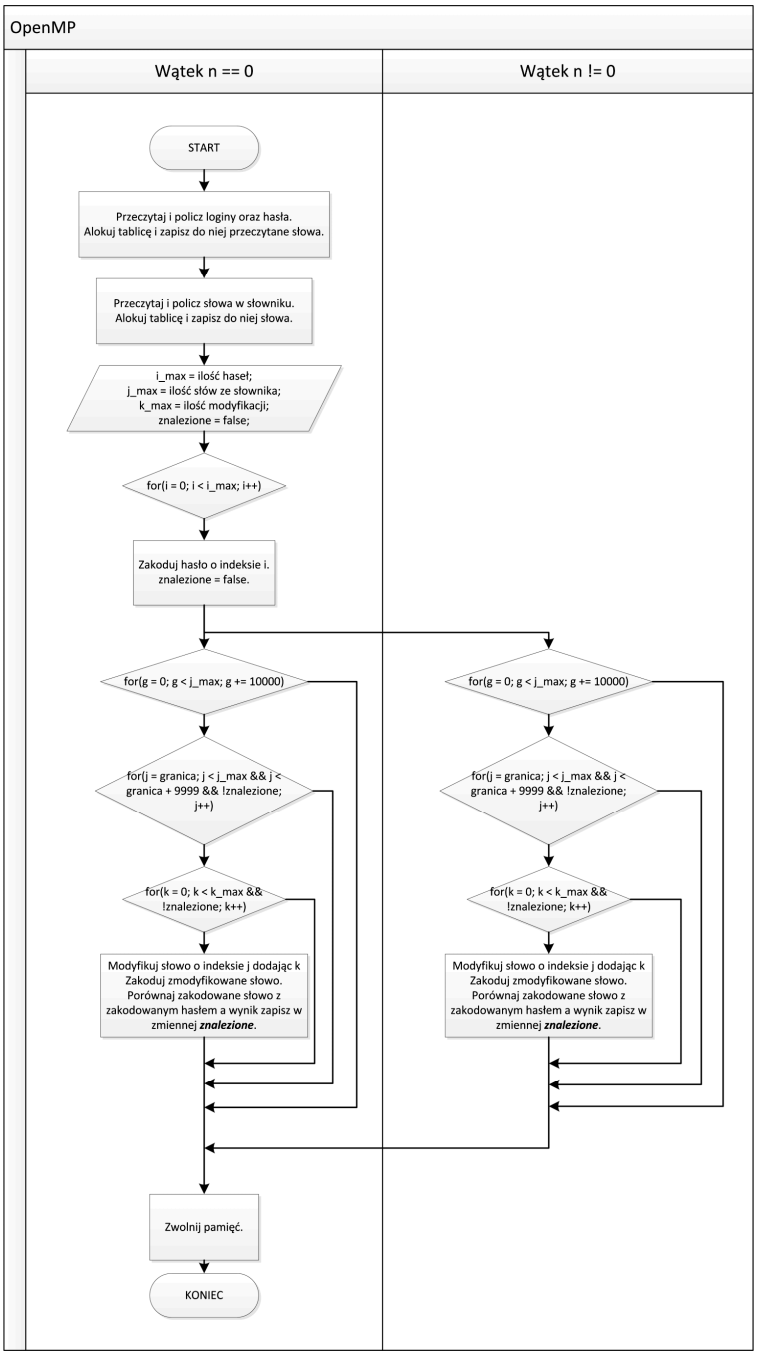
Program odczytuje z pliku pary haseł i loginów (nazw użytkowników). Ilość haseł do złamania może być dowolna – program przed wczytaniem haseł do pamięci komputera najpierw liczy je, a potem alokuje potrzebną pamięć i zapisuje w niej hasła. Słownik również czytany jest z pliku przy wykorzystaniu podobnego algorytmu jak przy wczytywaniu haseł, czyli najpierw liczona jest ilość słów, a potem alokowana potrzebna pamięć i czytany do niej słownik. Wielkość słownika, z którego pobierane są wzorce, jest dowolna. Ograniczają ją jedynie sprzętowe możliwości alokowania tablicy zawierającej wszystkie słowa. Funkcja modyfikująca słowo przed porównaniem zwiększa zakres poszukiwań – dla każdego jednego słowa ze słownika tworzonych jest 7 jego dodatkowych modyfikacji. Funkcja wywoływana jest w pętli ośmiokrotnie. Przy pierwszym wywołaniu słowo nie jest modyfikowane. Przy każdym kolejnym do słowa dopisywana jest cyfra z zakresu od 1 do 7. Fakt modyfikacji każdego słowa sprawia, że dla wykorzystanego słownika program jest w stanie dokonać prawie 22 mln porównań dla jednego hasła.

4.2. Implementacja w środowisku MPI

Słownik jest dzielony na równe części przydzielane wszystkim procesom. Proces 0 czyta z pliku słownik oraz hasła do złamania. Najpierw je liczy, po czym alokuje potrzebną pamięć i do niej zapisuje zawartość plików. Potem oblicza ile słów przypada na jeden proces i wysyła do wszystkich potrzebne informacje. Na koniec dokonywany jest transfer słownika, aby każdy proces posiadał tylko tą jego cześć, na której ma operować. Pętla badająca kolejne słowa jest podzielona na mniejsze pętle po 10 tys. przejść. Zabieg ten ma na celu przerwanie pracy innych procesów po znalezieniu hasła przez jeden z nich. Aby nie tracić czasu na synchronizację wszystkich procesów w tym samym czasie, wysyłane są komunikaty nieblokujące. Procesem decydującym o zakończeniu poszukiwań jest proces 0. Każdy inny proces w przypadku złamania hasła wysyła do niego komunikat zgłaszający znalezienie hasła, po czym przesyła znalezione hasło. Proces 0 w przypadku otrzymania komunikatu o złamaniu hasła przez inny proces odbiera także komunikat zawierający złamane hasło. Jeśli proces 0 „wie”, że hasło zostało złamane, wysyła taką informacje do wszystkich pozostałych procesów. Tylko proces 0 wyświetla informacje o tym, jakie są złamane hasła. Steruje całą komunikacją oraz zbiera informacje od wszystkich procesów. Ilość komunikacji w przypadku kilku rdzeni (procesorów) nie jest jednak na tyle duża, by wymagała osobnego procesu do wykonywania tego zadania, więc proces 0 wykonuje poszukiwania, tak jak i wszystkie inne procesy. Każdy proces co 10 tys. haseł sprawdza, czy nie został do niego wysłany komunikat informujący o złamaniu hasła przez inny proces. Jeśli otrzyma taki komunikat przerywa pracę. Diagram aktywności dla implementacji w środowisku MPI przedstawia rys. 1.



Rys. 1. Diagram aktywności dla implementacji w środowisku MPI
Fig. 1. Activity diagram for MPI implementation



Rys. 2. Diagram aktywności dla implementacji w środowisku OpenMP

Fig. 2. Activity diagram for OpenMP implementation

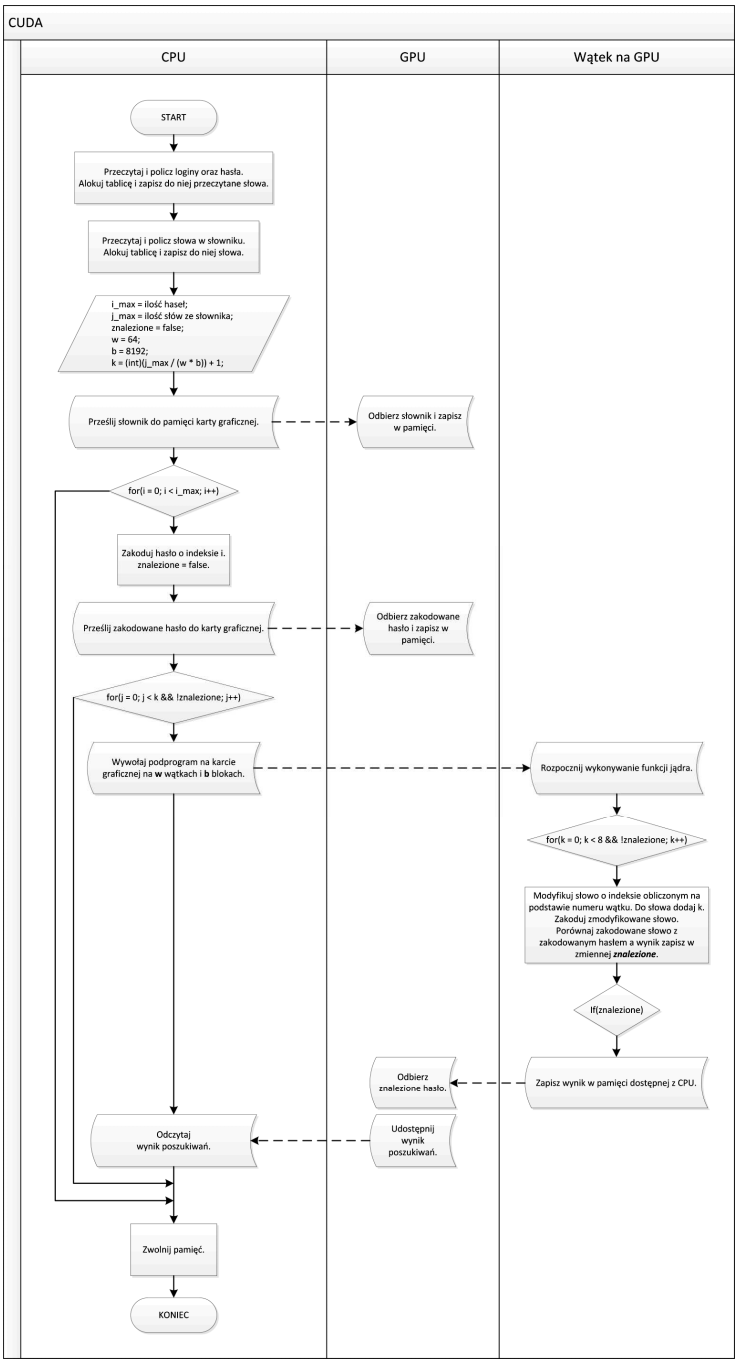
4.3. Implementacja w środowisku OpenMP

Algorytm wczytujący słownik z pliku do pamięci komputera działa w pętli. Jedno wykonanie pętli czyta słowo z pliku i zapisuje je do wcześniej zaalokowanej jednowymiarowej tablicy znaków. Dla każdego słowa przewidziane jest miejsce na 20 znaków. Słownik został sprawdzony i nie posiada dłuższych słów. Indeks położenia słowa w tablicy jest liczony przez pomnożenie numeru słowa przez 20. Program działający w ten sposób, alokuje słownik do pamięci w ciągu 2 sekund. Na rys. 2 przedstawiono diagram aktywności dla implementacji w środowisku OpenMP.

Badany algorytm posiada pętlę, przy pomocy której kolejne słowa ze słownika porównywane są z szukanym hasłem. Jeśli hasło znajduje się wśród porównywanych słów, pętla jest przerywana. Środowisko OpenMP nie umożliwia przerwania zrównoleglonej pętli. Z tego powodu pętla „przechodząca" przez wszystkie słowa została podzielona na mniejsze wczytujące z tablicy 10 tys. słów każda. Pętla główna zmienia swój licznik, dodając do niego po każdym przejściu wartość 10 tys.. Mniejsze pętle wewnętrzne nie są zrównoleglone, więc kiedy w jednym z 10 tys. przejść odnajdą właściwe hasło są przerywane. W momencie znalezienia właściwego hasła zmieniany jest również specjalny znacznik informujący program podczas dalszego wykonania, że hasło zostało już znalezione. Zrównoleglona pętla zewnętrzna musi skończyć wszystkie swoje przejścia, których liczba była znana przed jej uruchomieniem. Jednak skoro znacznik informuje, że hasło zostało już znalezione, pętle wewnętrzne są natychmiast przerywane, a tym samy cały program w krótkim czasie zostaje zakończony wykonując jedynie niewielka ilość nadmiarowych obliczeń. Opisana wyżej pętla porównująca każde słowo jest wykonywana dla wszystkich wyszukiwanych haseł, a te z kolei również odczytywane są przy pomocy pętli czytającej kolejne hasła do złamania. Zrównoleglona cześć znajduje się wewnątrz tej pętli i na jej końcu znajduje się domyślna bariera. Jest ona potrzebna, ponieważ w razie jej braku wątek, który złamał hasło, mógłby przejść do poszukiwań kolejnego wcześniej niż inne. Znacznik po odnalezieniu hasła musi zostać na nowo wyzerowany i dopiero wtedy program może przejść do wyszukiwania kolejnego hasła. Pętle zostały zrównoleglone z dyrektywą *schedule(dynamic)*, co znaczy że każdy wątek wykonuje jedną iterację pętli. Jeśli pętla nie została zakończona, a wątek zakończył swoją cześć to zostaje mu przydzielona kolejna iteracja. Słownik jest jednokrotnie zapisywany do pamięci komputera i każdy korzysta z tej samej kopii umieszczonej w pamięci dzielonej. W pamięci lokalnej poszczególnych wątków alokowane jest miejsce na kolejne pobierane ze słownika słowa oraz zmienne potrzebne do ich zakodowania.

4.4. Implementacja w technologii CUDA

Słownik został w tym przypadku w całości skopiowany do pamięci karty graficznej. Najpierw zostały policzone słowa w słowniku oraz hasła do złamania. Alokowana została przestrzeń w pamięci RAM komputera, oraz przestrzeń w pamięci karty graficznej. Słownik oraz hasła do złamania zostały wczytane do tablicy w pamięci komputera po czym w całości skopiowane do pamięci karty graficznej. Ze względu na ograniczenia techniczne funkcja poszukująca hasła wykonywana na karcie graficznej zostaje wywołana kilkakrotnie. Wywołanie tej funkcji jądra jest równoległym wywołaniem wielu jej instancji wykonywanych przez pojedyncze wątki. W trakcie testów okazało się, ze najlepsze wyniki daje wywołanie funkcji jednocześnie w 8192 blokach zawierających 64 wątki każdy.



Rys. 3. Diagram aktywności dla implementacji w środowisku CUDA

Fig. 3. Activity diagram for CUDA implementation

Pojedynczy kernel porównuje 1 słowo ze słownika. Przeprowadzone zostały również testy, gdzie kernel porównywał 1000 słów. Była to próba analogii do implementacji wykonanych w środowiskach MPI i OpenMP. W wymienionych środowiskach było to 10 tys. słów dla jednego wątku, ale funkcja jądra posiada ograniczenie ilości wykonanych operacji do 2 mln. Z tego powody możliwe było porównanie jedynie nieco ponad 1000 słów w jednym wywołaniu funkcji. Ta wersja programu została jednak zmieniona, ponieważ w trakcie testów okazało się, ze wywołując funkcje jądra dla jednego słowa szybkość wykonywania obliczeń zwiększyła się. Analogicznie jednak do innych implementacji jeśli hasło zostało złamane nie są porównywane kolejne słowa. Funkcja jądra została zoptymalizowana przy pomocy kwalifikatora *launch_bounds (64,8)* przy definicji funkcji. Kwalifikator ten zoptymalizował ilość wykorzystywanych rejestrów przez funkcje jądra, tak aby mogła działać wywołana na co najmniej 8 blokach zawierających nie więcej niż 64 wątki każdy. Ilość wątków w bloku została wybrana nieprzypadkowo. Ze względu na organizację pamięci, w celu uzyskania najlepszej wydajności, powinna to być wielokrotność liczby 64. Przyjęta została więc ilość wątków w bloku równa 64. Badana implementacja nie korzysta z możliwości dzielenia pamięci w obrębie bloku w związku z czym zwiększanie ilości wątków w bloku nie jest konieczne. Ilość bloków została wyznaczona podczas testowania programu, gdzie najlepsze wyniki zostały uzyskane przy ustawieniu ilości bloków na 8192. Na rys. 3 przedstawiono diagram aktywności dla implementacji w środowisku CUDA.

5. Badanie algorytmów

Programy zostały przetestowane na dwóch maszynach:

- programy wykonywane na CPU korzystały z komputera z procesorem AMD Athlon II X4 635 posiadającym 4 rdzenie o częstotliwości 2900 MHz każdy i 2 GB pamięci RAM,
- program wykonywany na GPU korzystał z komputera z procesorem Intel Pentium Core 2 Duo P8700 posiadającym 2 rdzenie o częstotliwości 2530 MHz każdy i 4 GB pamięci RAM oraz kartę graficzną NVIDIA GeForce 9300M GS posiadającą 8 rdzeni o częstotliwości 580 MHz, pamięć wewnętrzną wielkości 256 MB.

Oba komputery posiadały zainstalowany system operacyjny Microsoft Windows 7 Pro x64 oraz środowisko programistyczne Microsoft Visual Studio 2008 Pro.

Programy zostały przetestowane dla jednakowych danych wejściowych. Poszukiwały 8 haseł mieszczących się w różnych miejscach słownika bądź nie znajdujących się w nim. Jako źródło wzorców do wyszukiwania haseł wykorzystany został słownik zawierający ponad 2,7 mln słów opisany wcześniej.

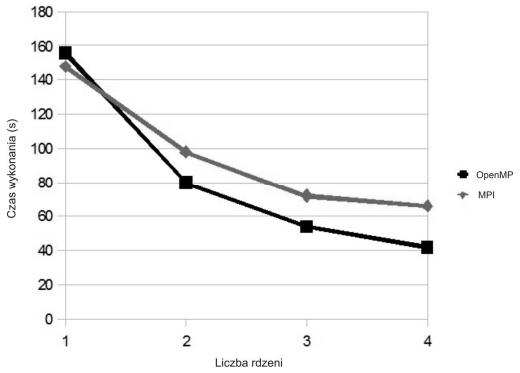
Czas wykonania sekwencyjnej wersji programu wyniósł 147 sekund i do tego wyniku zostały wykonane porównania w dalszej części artykułu.

Programy działające w środowiskach OpenMP oraz MPI zostały uruchomione kolejno na 1, 2, 3 i 4 rdzeniach procesora. Wyniki przedstawione są na wykresach zależności czasowych oraz przyspieszenia.

Rysunek 4 pokazuje czasy wykonania programów zrównoleglonych w środowiskach MPI oraz OpenMP w zależności od liczby wykorzystanych rdzeni. Widać na nim, że kolejne wątki znacznie zmniejszają czasy wykonania, co świadczy o efektywności zaimplementowanych algorytmów.

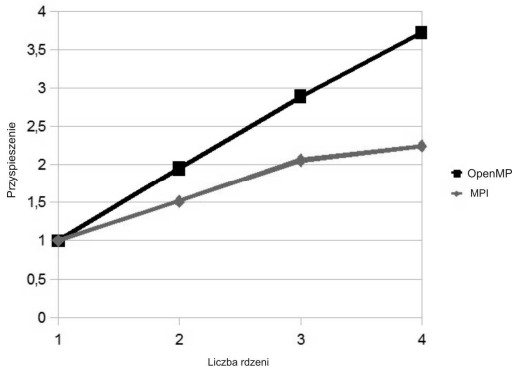
Rysunek 5 przedstawia uzyskane przyspieszenia. W przypadku środowiska OpenMP wartości są bliskie ilości wykorzystanych rdzeni, co świadczy o bardzo dobrej efektywności programu, bliskiej idealnemu przyśpieszeniu. W przypadku środowiska MPI widać, że koszty związane z komunikacją oraz synchronizacją procesów wraz ze wzrostem liczby rdzeni są coraz większe i powodują spadek wartości przyspieszenia.

Rysunek 6 porównuje czasy wykonania programu w wersji sekwencyjnej na CPU oraz zrównoleglonej na GPU. Jak widać w środowisku karty graficznej, czas wykonania został ponad trzykrotne skrócony.



Rys. 4. Czasy wykonania programu dla różnej liczby rdzeni dla środowisk MPI i OpenMP

Fig. 4. Wall-clock execution time for different number of cores for MPI and OpenMP environments



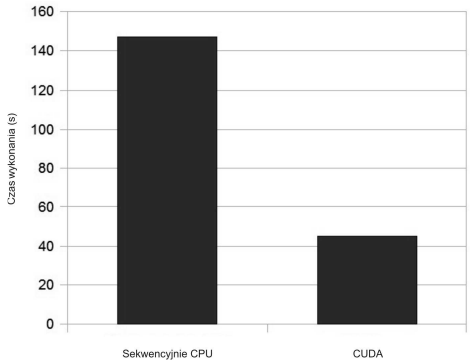
Rys. 5. Przyspieszenie programów dla środowisk MPI i OpenMP

Fig. 5. Speed-up for MPI and OpenMP environments

Aby podsumować wyniki badań, na jednym wykresie (rys. 7) porównano czasy wykonania programu sekwencyjnego oraz programów równoległych uruchomionych we wszystkich testowanych środowiskach. Dane odzwierciedlone na wykresie dla środowisk OpenMP i MPI to wyniki dla aplikacji uruchomionej na 4 rdzeniach.

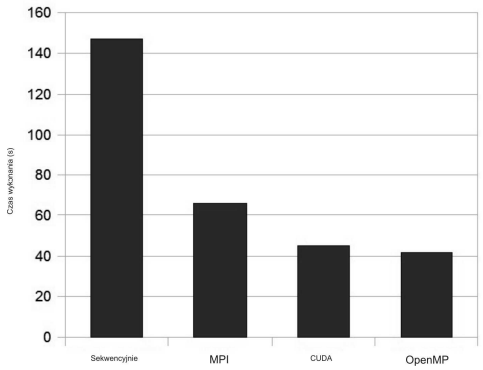
Najkrócej wykonywał się program zaimplementowany w środowisku OpenMP. Nieco dłużej trwało jego wykonanie w technologii CUDA. Środowisko MPI dało najgorszy wy-

nik. Oczywiście kolejność uzyskanych czasów może ulec zmianie, głównie jeśli chodzi o czas uzyskany w środowisku GPU, który jest silnie zależny od użytej karty graficznej i może dać efekt lepszy niż aplikacja uruchomiona w środowisku OpenMP. Natomiast należy domniemać, iż czasy uzyskane dla implementacji MPI będą zawsze gorsze od implementacji OpenMP z powodu dużych kosztów czasowych związanych z przesyłaniem komunikatów.



Rys. 6. Czasy wykonania algorytmów sekwencyjnego i zaimplementowanego w środowisku CUDA

Fig. 6. Wall-clock execution time for sequential algorithm and algorithm implemented in CUDA environment



Rys. 7. Czasy wykonania algorytmów sekwencyjnego oraz w środowiskach MPI, OpenMP i CUDA

Fig. 7. Wall-clock execution time for sequential algorithm and algorithms implemented in MPI, OpenMP and CUDA environments

6. Wnioski

Wyniki badań pokazują, że programy wykonywane w środowiskach programowania równoległego wykorzystują potencjał komputerów. Istnieją różne metody i narzędzia zrównoleglenia algorytmów. Na pewno różne środowiska programowania równoległego poz-

walają optymalnie przyśpieszyć inną klasę problemów, ale zawsze pozwalają na lepsze wykorzystanie posiadanego sprzętu niż wykonywanie jedynie programów sekwencyjnych.

Do badanego problemu sprzęt, na którym testowane były programy, pozwolił na otrzymanie bardzo dobrych wyników dla programu wykonywanego w środowisku OpenMP. Wykres przyśpieszeń pozwala sądzić, że testowany algorytm jest w stanie przyjąć kolejne procesory do równoległego wykonywania programu, dając ciągle dość dobre wyniki. Implementacja w środowisku MPI niestety nie dała tak dobrych wyników. Program, mimo że uzyskał znaczne przyspieszenie w stosunku do wersji sekwencyjnej to nie może się równać z implementacjami OpenMP czy CUDA.

Najciekawszą implementacją jest program wykonywany na procesorze karty graficznej. Co prawda nie uzyskał on najkrótszego czasu wykonania, ale należy wziąć po uwagę, że procesor komputera, na którym wykonywane były testy, jest procesorem wyższej klasy niż wykorzystana karta graficzna. Wykorzystany procesor nie występuje często w połączeniu z tą karta, a prawie zawsze z kartą o większej mocy. Niemniej jednak program wykonany na karcie graficznej „dosięgnął" niemal maksymalnych możliwości procesora komputera. Programowanie równoległe daje więc wielkie możliwości stworzenia w domowych warunkach laboratorium komputerowego i to bardzo małym kosztem. Dzisiaj praktycznie każdy procesor posiada kilka rdzeni, a karty graficzne wciąż mało znane, ale z wielkimi możliwościami oferują dużą moc obliczeniowa. Należy rozwijać gałąź informatyki związaną z programowaniem równoległym i przenosić ją do programów wykorzystywanych przez zwykłych użytkowników. Inaczej coraz więcej osób zacznie zauważać, że nowy sześciordzeniowy procesor nie działa szybciej od starego laptopa. A przecież ten procesor ma wielkie możliwości.

Literatura

[1] Alonso F., *The Extinction of Password Authentication*, ISSA Journal, December 2008.
[2] Sprengers M., *GPU-based Password Cracking. On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units*, Radboud University Nijmegen 2011.
[3] *Message Passing Interface (MPI) Tutorial* (https://computing.llnl.gov/tutorials/mpi).
[4] Karbowski A., Niewiadomska-Szynkiewicz E., *Programowanie równoległe i rozproszone*, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2009.
[5] *OpenMP Tutorial* (https://computing.llnl.gov/tutorials/openMP).
[6] NVIDIA (http://www.nvidia.pl/object/what is cuda new pl.html).
[7] *NVIDIA CUDA, C Programing Guide, Version 3.2*, NVIDIA Corporation, październik 2010.
[8] *NVIDIA CUDA, Reference Manual, Version 3.2 Beta*, sierpień 2010.
[9] Source Forge (http://sourceforge.net/projects/libmd5-rfc/files).
[10] Słownik Języka Polskiego (http://www.sjp.pl).