Michał Lipiński, Grzegorz Majchrzak

PORR – sprawozdanie końcowe projektu

Temat: Wyznaczanie ścieżki krytycznej w bardzo dużym grafie

1. Temat

W celu wyznaczenie ścieżki krytycznej, w dużym skierowanym grafie asynchronicznym, został przez nas zastosowany algorytm *Bellmana-Forda*, służący oryginalnie do wyznaczania najkrótszej ścieżki w grafie wychodzącej z danego wierzchołka. Ścieżka krytyczna, czyli najdłuższa ścieżka w grafie wychodząca z wierzchołka startowego, może zostać wyznaczona przez ten algorytm po ówczesnym przemnożeniu wag krawędzi przez -1, przez co ścieżka najkrótsza będzie tak naprawdę najdłuższa. Do tego typu wyliczeń nie nadają się inne algorytmy wyznaczania najkrótszej ścieżki, ponieważ nie wszystkie radzą sobie z wagami ujemnymi (jak na przykład algorytm *Dijkstry*).

Algorytm *Bellmana-Forda* wygląda następująco:

```
Bellman-Ford(G,w,s):

dla każdego wierzchołka v w V[G] wykonaj
    d[v] = nieskończone
    poprzednik[v] = niezdefiniowane
d[s] = 0
dla i od 1 do |V[G]| - 1 wykonaj
   dla każdej krawędzi (u,v) w E[G] wykonaj
     jeżeli d[v] > d[u] + w(u,v) to
        d[v] = d[u] + w(u,v)
        poprzednik[v] = u
```

2. Generowanie grafu *DAG (Directed Acyclic Graph)*

Pierwszym etapem programu, przed wykonaniem jakichkolwiek obliczeń, należało wygenerować graf na którym będziemy wykonywali operacje. Na potrzeby programu, należało wygenerować graf skierowany, z wagami przypisanymi do krawędzi, który jednocześnie byłby acykliczny. Struktura taka nazywana jest jako *DAG (Directed Acyclic Graph).* Aby wygenerować graf użyliśmy metody:

```
Algorithm: Layer-by-Layer method.
Require: n, p ∈ N.
Algorytm umieszcza n wierzchołków z liczbą krawędzi powiązaną z
prawdopodobieństwem p.
M – macierz sąsiedztwa o wymiarze nxn.

for all i = 1 to n do
  for all j = 1 to n do
    if j > i then
       if Random() < p then
          M[i][j] = 1
       else
          M[i][j] = 0

Zwraca losowy DAG z n wierzchołkami
```

3. Implementacja Algorytmu *Bellmana-Forda*

Cały program został napisany przez nas w języku *C++.* Stworzyliśmy architekturę łatwo rozszerzalną o nowe wersje implementacje tego samego algorytmu. Kody źródłowe programu podzielone zostały pomiędzy 3 główne katalogi: *gen*, *include* oraz *src*. Katalog *gen* zawiera pliki z wygenerowanymi grafami, katalog *include* zawiera pliki nagłówkowe, w których znajdują się deklaracje wszystkich klas użytych w projekcie, natomiast katalog *src* zawiera implementację poszczególnych klas. Za łatwą rozszerzalność projektu odpowiada klasa *AbstractGraph,* po której dziedziczą wszystkie główne klasy odpowiedzialne za poszczególne wersje implementacji algorytmu. Klasa ta ma zaimplementowane takie przydatne metody, jak generowanie grafu czy zapisywanie grafu do pliku, wczytywanie grafu z pliku.

3.1. Wersja sekwencyjna

Wersja sekwencyjna jest najprostszym przeniesieniem wcześniej przedstawionego algorytmu, do języka *C++.* Oczywiście przed jakimikolwiek operacjami na grafie, wartości przy każdej krawędzi przemnożone są przez *-1.* Wszystkie operacje są wykonywane przy użyciu najprostszych struktur danych z języka *C++,* tablic jedno i dwu wymiarowych oraz wektorów.

3.2. Wersja zrównoleglona przy użyciu C11Threads

C11 to standard języka C wprowadzony w 2011 roku, który zastępuje poprzedni standard *C99*. W tym standardzie zostały między innymi wprowadzone wątki, znajdujące się w pliku *threads.h.* Wersja zrównoleglona przy użyciu wątków pochodzących z *C11* jest bardzo podobna do wersji sekwencyjnej, jednak wewnątrz algorytmu *Bellmana-Forda,* dla każdego wierzchołka tworzony jest nowy wątek. Dodatkowo zapis do tablicy zawierającej informacje o aktualnie wyliczonych odległości do danego wierzchołka jest zabezpieczony przy użyciu semafora, aby nie powstało zjawisko wyścigów podczas wykonywania programu. Najważniejsze elementy tej wersji przedstawia następujący kod.

```
AbstractGraph::path * C11ThreadsVersion::getCriticalPath(unsigned
vertexStart)
{
        for (int i = 0; i < vertexesNumber; i++)
                for (int j = 0; j < vertexesNumber; j++)
                        matrix[i][j] = -matrix[i][j];

        for (int i = 0; i < vertexesNumber; i++) {
                distance[i] = LONG_MAX;
        }
        distance[vertexStart] = 0;

        path* res = new path();

        std::thread* threads = new std::thread[vertexesNumber];
        for (int i = 0; i < vertexesNumber; i++) {
                threads[i] =
std::thread(&C11ThreadsVersion::bellmanFord, this, i, 0);
                threads[i].join();
        }
```

```
        std::vector<long> temp_distance = std::vector<long>(distance,
distance + vertexesNumber);

        int intIndex = std::min_element(temp_distance.begin(),
temp_distance.end()) - temp_distance.begin();
        res->pathLength = -temp_distance[intIndex];

        return res;
}

void C11ThreadsVersion::bellmanFord(const int threadIndex, unsigned
row) {

        for (int j = 0; j < vertexesNumber; j++) {
                if (matrix[threadIndex][j] != 0) {
                        if (distance[j] > distance[threadIndex] +
matrix[threadIndex][j]) {
                                mutexes->at(j)->lock();
                                distance[j] = distance[threadIndex] +
matrix[threadIndex][j];
                                mutexes->at(j)->unlock();
                        }
                }
        }
}
```

3.3. Wersja zrównoleglona przy użyciu OpenMP

OpenMp jest to API, umożliwiające tworzenie programów dla systemów wieloprocesorowych operujących na pamięci dzielonej. Może być wykorzystywany w językach, takich jak C, C++ czy Fortran. Składa się między innymi z dyrektyw kompilatora, mających wpływ na sposób wykonywania się programu, z których skorzystaliśmy w naszym projekcie. Nasza wersja implementacji oparta o OpenMP wygląda bardzo podobnie do wersji sekwencyjnej, natomiast zawiera pewne dyrektywy przed fragmentami kodu który ma zostać zrównoleglony. W projekcie użyliśmy takich dyrektyw jak: *parallel, parallel for, barier* oraz *single*. Najważniejsze elementy tej wersji przedstawia następujący kod.

```
AbstractGraph::path * OpenMPVersion::getCriticalPath(unsigned
vertexStart) {

#pragma omp parallel for

        for (int i = 0; i < vertexesNumber; i++)      // ujemne wagi
                for (int j = 0; j < vertexesNumber; j++)
                        matrix[i][j] = -matrix[i][j];

        path* res = new path();

        std::pair<std::vector<long>, std::vector<unsigned>> pair;
        bellmanFord(vertexStart, &pair);
        int intIndex = std::min_element(pair.first.begin(),
pair.first.end()) - pair.first.begin();
        res->pathLength = -pair.first[intIndex];
        return res;
}
```

```
void OpenMPVersion::bellmanFord(unsigned row,
std::pair<std::vector<long>, std::vector<unsigned>>* pair) {
        std::vector<long> distance(vertexesNumber);
        std::vector<unsigned> predecessor;

#pragma omp parallel for

        for (int i = 0; i < vertexesNumber; i++) {
                distance[i] = LONG_MAX;
        }

        distance[row] = 0;

#pragma omp parallel

        for (int i = 0; i < vertexesNumber; i++) {
                for (int j = 0; j < vertexesNumber; j++) {
                        if (matrix[i][j] != 0) {
                                if (distance[j] > distance[i] +
matrix[i][j]) {
                                        distance[j] = distance[i] +
matrix[i][j];
                                }
                        }
                }
        }

#pragma omp barrier
#pragma omp single

        pair->first = distance;
        pair->second = predecessor;
}
```

3.4. Wersja zrównoleglona przy użyciu CUDA

Ostatnią przygotowaną przez nas wersją implementacji jest wersja oparta o CUDA. CUDA, to opracowana przez firmę Nvidia uniwersalna architektura procesorów wielordzeniowych. Obliczenia wykonywane w tej technologii, realizowane są nie na procesorze komputera, a na karcie graficznej. Ta wersja powinna działać teoretycznie najlepiej, ponieważ w algorytmie *Bellmana-Forda,* opiera się na wielu tych samych operacjach wykonywanych na różnych danych, a właśnie takie operacje działają najlepiej na procesorach graficznych. Problemem jednak może okazać się ograniczona pamięć na karcie graficznej oraz konieczność skopiowania danych z pamięci ram komputera, do pamięci globalnej karty graficznych, które może trwać stosunkowo długo. Najważniejsze elementy tej implementacji zawiera następujący kod. Dla tej wersji programu, przedstawimy dwa różne czasy jego wykonywania, pierwszy łącznie ze skopiowaniem pamięci na kartę graficzną oraz drugi, czas samych obliczeń wykonywanych na karcie.

```
__global__ void relax(unsigned vertexesNumber, unsigned edgesAmount,
int edgeStart, std::pair<int, int>* cuda_stab, int* cuda_distance) {
        int id = blockDim.x * blockIdx.x + threadIdx.x;

        if (id >= edgesAmount) return;

        int weight = cuda_stab[edgeStart * vertexesNumber + id].second;
```

```
        if (weight >= 0) return;

        int endVertex = cuda_stab[edgeStart * vertexesNumber +
id].first;

        if (cuda_distance[endVertex] > cuda_distance[edgeStart] +
weight) {
                atomicMin(&(cuda_distance[endVertex]),
(cuda_distance[edgeStart] + weight));
        }
}

void CUDAVersion::bf(unsigned row, std::pair<std::vector<int>,
std::vector<unsigned>>* pair) {
        int* distance = new int[vertexesNumber];
        int* return_distance = new int[vertexesNumber];

        int* cuda_distance;

        std::pair<int, int>* cuda_stab;
        std::vector<unsigned> predecessor;

        dim3 blocks(blocksNumber);
        dim3 threads(threadsNumber);

        cudaMalloc(&cuda_stab, sizeof(std::pair<int,int>) *
vertexesNumber * vertexesNumber);
        cudaMalloc(&cuda_distance, sizeof(int) * vertexesNumber);
        cudaMemcpy(cuda_stab, stab, sizeof(std::pair<int, int>) *
vertexesNumber * vertexesNumber, cudaMemcpyHostToDevice);

        for (int i = 0; i < vertexesNumber; i++) {
                distance[i] = INT_MAX;
        }

        distance[row] = 0;
        cudaMemcpy(cuda_distance, distance, sizeof(int) *
vertexesNumber, cudaMemcpyHostToDevice);

        cudaEventCreate(&start);
        cudaEventCreate(&stop);

        cudaEventRecord(start);
        int b, t, edgesAmount;

        for (int i = 0; i < vertexesNumber; i++) { // wywolujemy tyle
watkow ile mamy par

                edgesAmount = tab_sizes[i];

                if (edgesAmount == 0) {
                        continue;
                }

                b = (edgesAmount / 24) + 1; // liczba blokow
                if (b == 1)
                        t = edgesAmount;
                else
                        t = 24;
```

```
                relax <<<b, t>>> (vertexesNumber, edgesAmount, i,
cuda_stab, cuda_distance);

        }

        cudaDeviceSynchronize();

        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&miliseconds, start, stop);

        cudaMemcpy(return_distance, cuda_distance, sizeof(int) *
vertexesNumber, cudaMemcpyDeviceToHost);

        cudaFree(cuda_stab);
        cudaFree(cuda_distance);
        pair->first = std::vector<int>(return_distance, return_distance
+ vertexesNumber);
        pair->second = predecessor;
}
```

4. Dane testowe

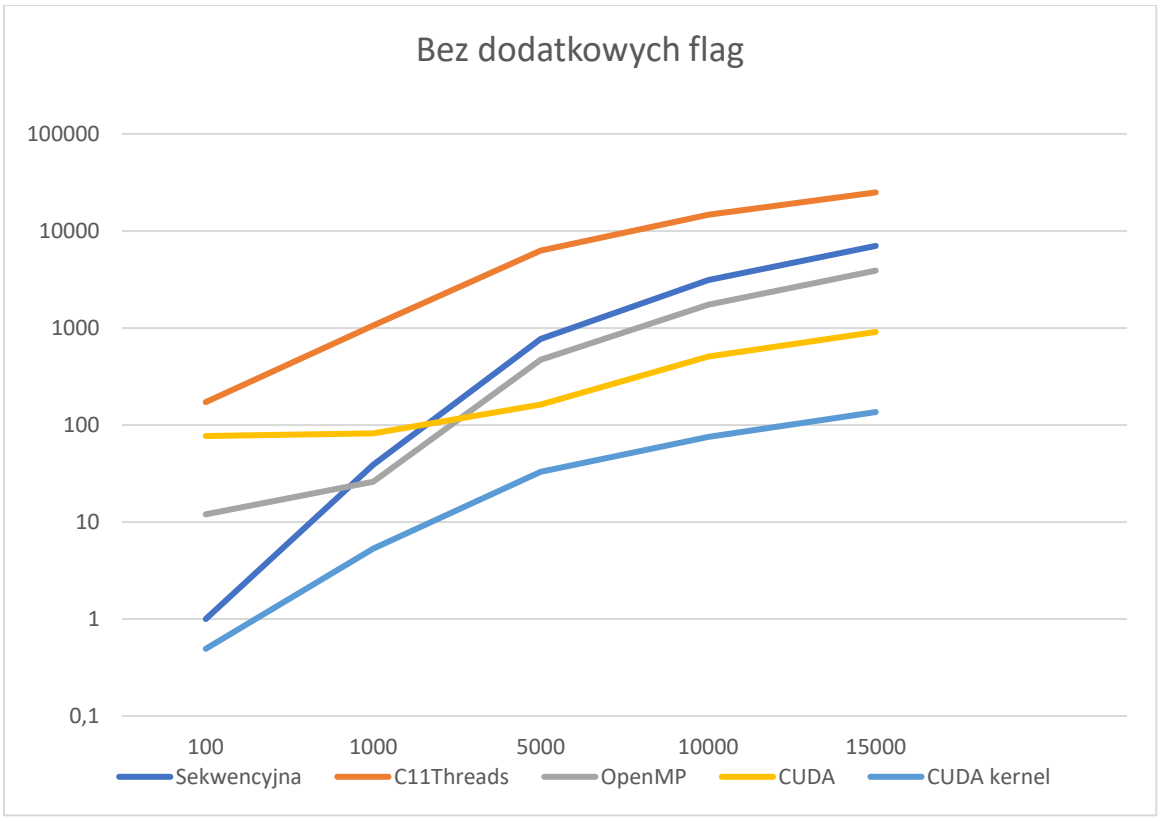
Wygenerowane przez nas grafy zapisywane są w plikach tekstowych, które zawierają pełną macierz sąsiedztwa grafu. W samym programie, graf również przechowujemy w postaci tablicy dwuwymiarowej będącą macierzą sąsiedztwa grafu. Aby przetestować działanie wykonanych przez nas implementacji algorytmu, wygenerowaliśmy grafy o różnych liczbach wierzchołków. Dzięki temu będziemy mogli ocenić korzyści płynące ze zrównoleglenia algorytmu w zależności od złożoności danych wejściowych do programu. Wygenerowane przez nas grafy mają odpowiednio po *100, 1000, 5000, 10000 i 15000* wierzchołków.

5. Wyniki

Pierwsze wyniki działania naszego programu powstały po skompilowaniu oraz uruchomieniu programu bez dodatkowych flag kompilacji. Wyniki przedstawia poniższa tabela.

Wielkość grafu [wierzchołki]	Wersja programu	Czas wykonywania obliczeń [ms]
100	Sekwencyjna	1
	C11Threads	172
	OpenMP	12
	CUDA	77
	CUDA kernel	0,493
1000	Sekwencyjna	39
	C11Threads	1058
	OpenMP	26
	CUDA	82
	CUDA kernel	5,3
5000	Sekwencyjna	771
	C11Threads	6283
	OpenMP	469
	CUDA	166
	CUDA kernel	32,83
10000	Sekwencyjna	3124
	C11Threads	14671
	OpenMP	1733
	CUDA	502
	CUDA kernel	75,55
15000	Sekwencyjna	7012
	C11Threads	24946
	OpenMP	3899
	CUDA	910
	CUDA kernel	135,87

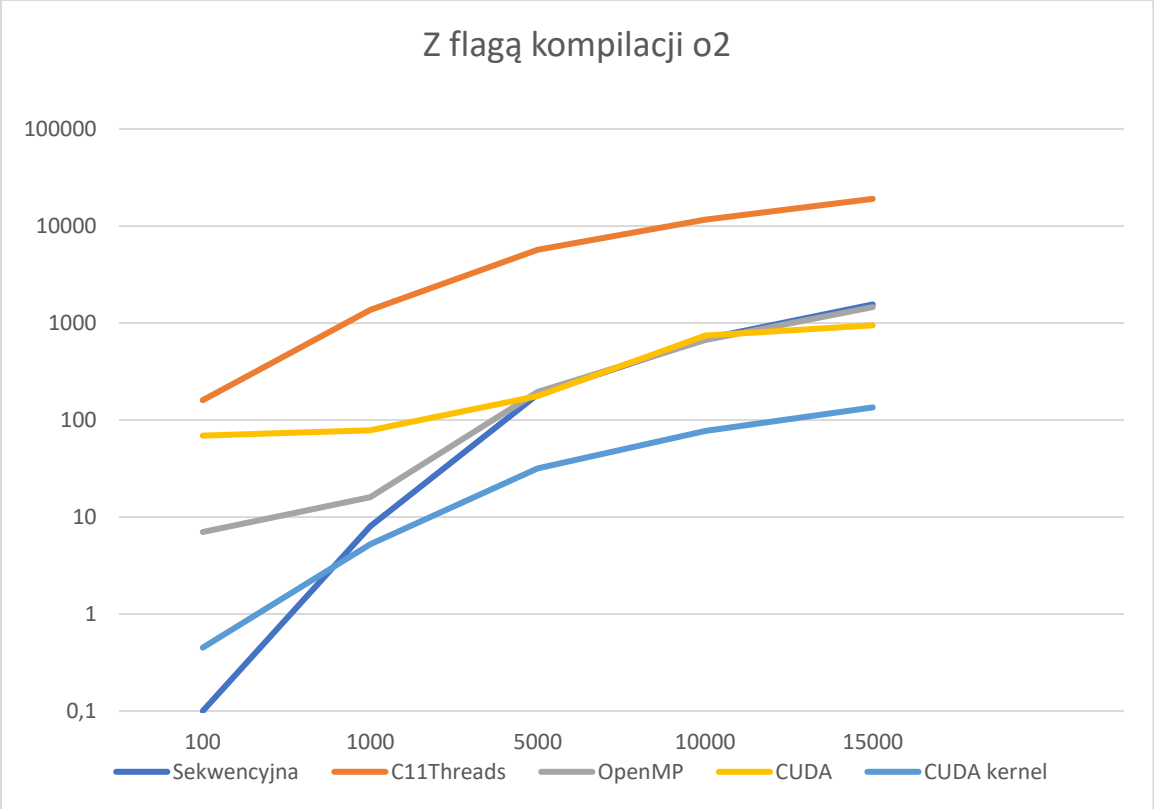
Otrzymane dane przedstawione na wykresie wyglądają następująco:



Kolejnym etapem testowania działania programu było skompilowanie go przy użyciu dodatkowej flagi kompilacji *O2*, która optymalizuje kod podczas kompilacji pod kątem szybkości działania. Wyniki przedstawia poniższa tabela.

Wielkość grafu [wierzchołki]	Wersja programu	Czas wykonywania obliczeń [ms]
100	Sekwencyjna	0,1
	C11Threads	160
	OpenMP	7
	CUDA	69
	CUDA kernel	0,45
1000	Sekwencyjna	8
	C11Threads	1361
	OpenMP	16
	CUDA	78
	CUDA kernel	5,23
5000	Sekwencyjna	183
	C11Threads	5670
	OpenMP	195
	CUDA	177
	CUDA kernel	31,61
10000	Sekwencyjna	686
	C11Threads	11582
	OpenMP	668
	CUDA	740
	CUDA kernel	77,35
15000	Sekwencyjna	1552
	C11Threads	19023
	OpenMP	1457
	CUDA	945
	CUDA kernel	134,78

Otrzymane dane przedstawione na wykresie wyglądają następująco:



W przypadku mniejszych grafów widzimy, że czas wykonania wersji sekwencyjnej programu nie odstaje od wersji wielowątkowych, a nawet jest on lepszy. Wynika to z samego inicjalizowania wątków, które też wymaga pewnego czasu. Wzrost wydajności oraz spadek czasu wykonywania programu obserwujemy dopiero przy grafie o wielkości około 5000 przy wersji wykonanej na OpenMP oraz przy użyciu technologii CUDA. Ciekawe wydaje się również to, że same operacje na grafie, z pominięciem czasu potrzebnego na przekopiowanie grafu do karty już dla grafu o wielkości pomiędzy 100 a 1000 wierzchołków są wykonywane szybciej przy użyciu technologii CUDA. Pomimo tego iż wersje *OpenMP* oraz *C11Threads* są podobne w założeniach, to wersja wykonana przy użyciu *OpenMP* działa zdecydowanie lepiej, co jest spowodowane koniecznością wykonania dodatkowej synchronizacji zrealizowanej na semaforach z języka *C++* wewnątrz naszego programu. Wątki które co chwilę działają oraz usypiają, nie są wydajne w porównaniu do *OpenMP*, gdzie bezpośrednia synchronizacja jest zaimplementowana wewnątrz biblioteki. Ciekawą obserwacją dokonaną przez nas podczas wykonywania oraz testowania tego projektu, jest to iż sam algorytm generowania grafu oraz jego wczytywanie/zapisywanie do i z pliku zajmuje nieporównywalnie więcej czasu niż samo wykonanie algorytmu *Bellmana-Forda,* niezależnie od implementacji. Spowodowane jest to zapewne ogromną wielkością grafu, szczególnie jeżeli reprezentujemy go za pomocą macierzy sąsiedztwa. Niestety w większości komórek, macierz ta ma wpisane zera, które i tak niepotrzebnie przechowujemy. Prawdopodobnie lepszym rozwiązaniem w naszym przypadku byłoby trzymanie informacji jedynie o krawędziach istniejących, na przykład za pomocą listy list sąsiadów każdego wierzchołka, oraz wagi na każdej z takich krawędzi.