Minimalne drzewo rozpinające - algorytm Prim'a przy użyciu MPI.

Daria Kokot, Bartosz Konopka

Spis treści

1 Wprowadzenie

1.1 Wstęp teoretyczny

Programowanie równoległe jest to sposób pisania programów, w których wiele zadań wykonywanych jest jednocześnie. Zamiast jednego ciągu instrukcji (jak w tradycyjnym programowaniu sekwencyjnym), istnieje wiele procesów lub wątków pracujących równolegle.

MPI, czyli Message Passing Interface, to standard komunikacji między procesami używany w programowaniu równoległym – szczególnie w środowiskach klastrów komputerowych i superkomputerów. Umożliwia wielu procesom wymianę informacji poprzez przesyłanie wiadomości.

W celu zrozumienia Algorytmu Prima, niezbędne jest zrozumienie niektórych zagadnień związanych z teorią grafów:

- **Graf nieskierowany** - graf, w którym krawędzie nie mają określonego kierunku, tzn. połączenie między dwoma wierzchołkami działa w obie strony.
- **Graf spójny** - graf, w którym istnieje ścieżka łącząca dowolną parę wierzchołków.
- **Graf acykliczny** - graf, w którym nie występują żadne cykle, czyli ścieżki zaczynające i kończące się w tym samym wierzchołku bez powtarzania krawędzi.
- **Graf ważony** - graf, w którym każda z krawędzi posiada przypisaną wagę.
- **Drzewo** - graf nieskierowany, który jest jednocześnie spójny i acykliczny. Oznacza to, że można przejść z dowolnego wierzchołka do innego tylko jednym, unikalnym sposobem, bez możliwości poruszania się „w kółko".
- **Drzewo rozpinające** - drzewo zawierające wszystkie wierzchołki danego grafu G, przy czym jego krawędzie stanowią podzbiór krawędzi grafu
- **Minimalne drzewo rozpinające** - drzewo rozpinające o najmniejszej możliwej sumie wag krawędzi.

Algorytm Prima to algorytm zachłanny służący do wyznaczania minimalnego drzewa rozpinającego. Działa na grafie nieskierowanym i spójnym, czyli takim, w którym każda para wierzchołków jest połączona pewną ścieżką, a krawędzie nie mają ustalonego kierunku. Algorytm znajduje podzbiór krawędzi $E\ prim \subseteq E$, taki że graf pozostaje spójny, a łączny koszt (waga) wszystkich wybranych krawędzi jest najmniejszy z możliwych.

1.2 Sekwencyjne działanie algorytmu Prima

Wejście: graf nieskierowany, spójny i ważony (każda krawędź ma przypisaną wagę).
Wyjście: minimalne drzewo rozpinające (MST – Minimum Spanning Tree).

Kroki algorytmu:

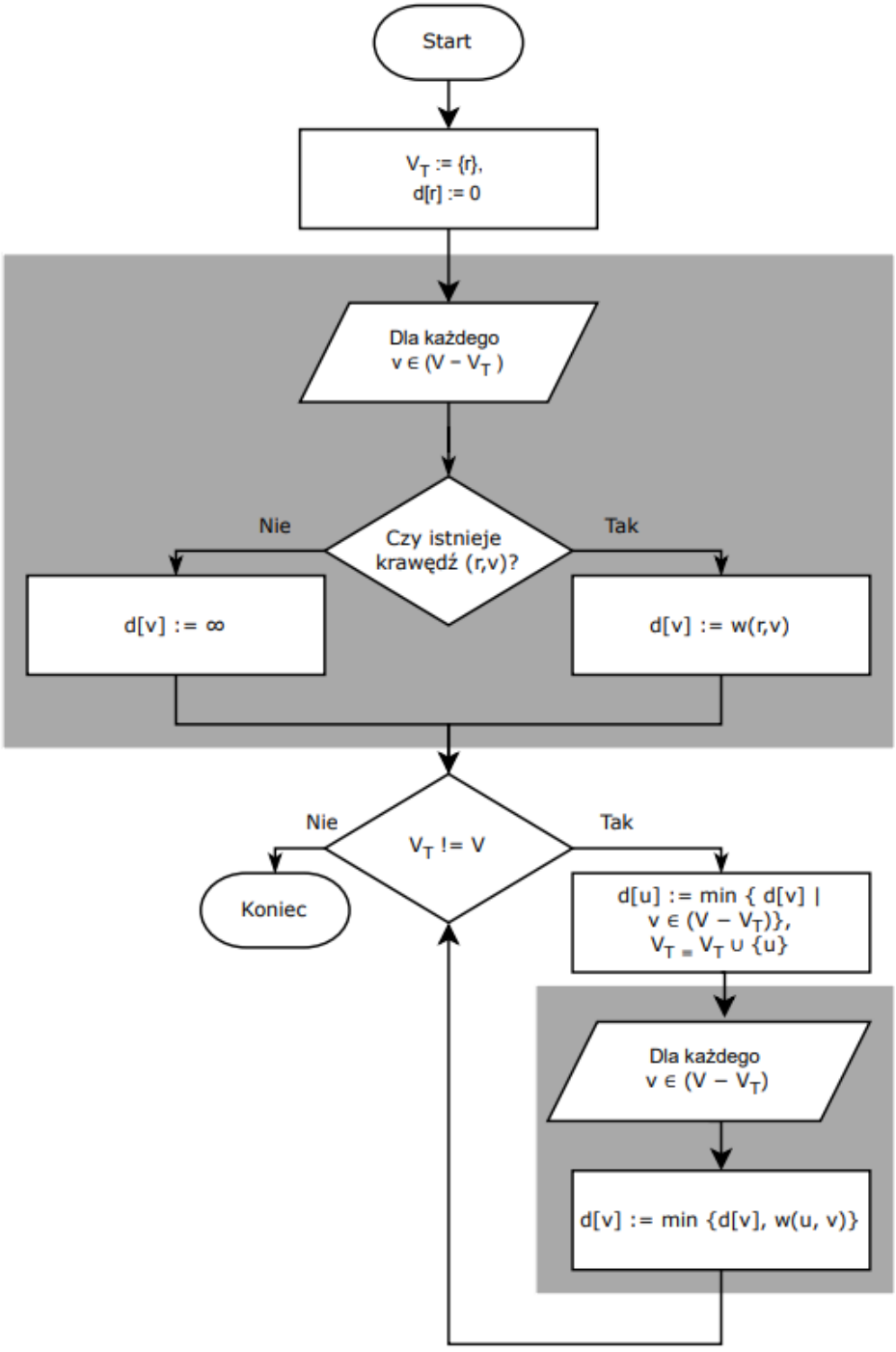
1. Wybierz dowolny wierzchołek początkowy i dodaj go do zbioru MST.
2. Znajdź wszystkie krawędzie wychodzące z MST (czyli takie, które łączą już odwiedzony wierzchołek z nieodwiedzonym).
3. Wybierz krawędź o najmniejszej wadze spośród dostępnych (której jeden koniec jest w MST, a drugi jeszcze nie).
4. Dodaj tę krawędź i jej końcowy wierzchołek do MST.
5. Powtarzaj kroki 2–4, aż wszystkie wierzchołki znajdą się w MST (czyli drzewo będzie zawierać n - *1* krawędzi dla n wierzchołków).

```
1.   procedure PRIM_MST(V, E, w, r)
2.   begin
3.       V_T := {r};
4.       d[r] := 0;
5.       for all v ∈ (V − V_T) do
6.           if edge (r, v) exists set d[v] := w(r, v);
7.           else set d[v] := ∞;
8.       while V_T ≠ V do
9.       begin
10.          find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.          V_T := V_T ∪ {u};
12.          for all v ∈ (V − V_T) do
13.              d[v] := min{d[v], w(u, v)};
14.      endwhile
15.  end PRIM_MST
```

Rysunek 1: Algorytm przedstawiający działanie algorytmu Prima, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Introduction to Parallel Computing, Addison-Wesley / Pearson Education 2003 [1]

Wyjaśnienie symobli w algorytmie:

- V – zbiór wszystkich wierzchołków grafu.
- E – zbiór wszystkich krawędzi grafu.
- $w(u, v)$ – waga (koszt) krawędzi między wierzchołkami u i v.
- r – wierzchołek początkowy, od którego zaczyna się budowanie drzewa rozpinającego.
- VT – zbiór wierzchołków już włączonych do drzewa minimalnego (MST).
- $d[v]$ – minimalny koszt krawędzi łączącej wierzchołek v z dowolnym wierzchołkiem należącym do VT (czyli do drzewa).
- u – wierzchołek spoza VT, który ma najmniejszy koszt przyłączenia do VT (czyli minimalne $d[u]$).



Rysunek 2: Schemat blokowy algorytmu Prima

1.3 Równoległe działanie algorytmu Prima

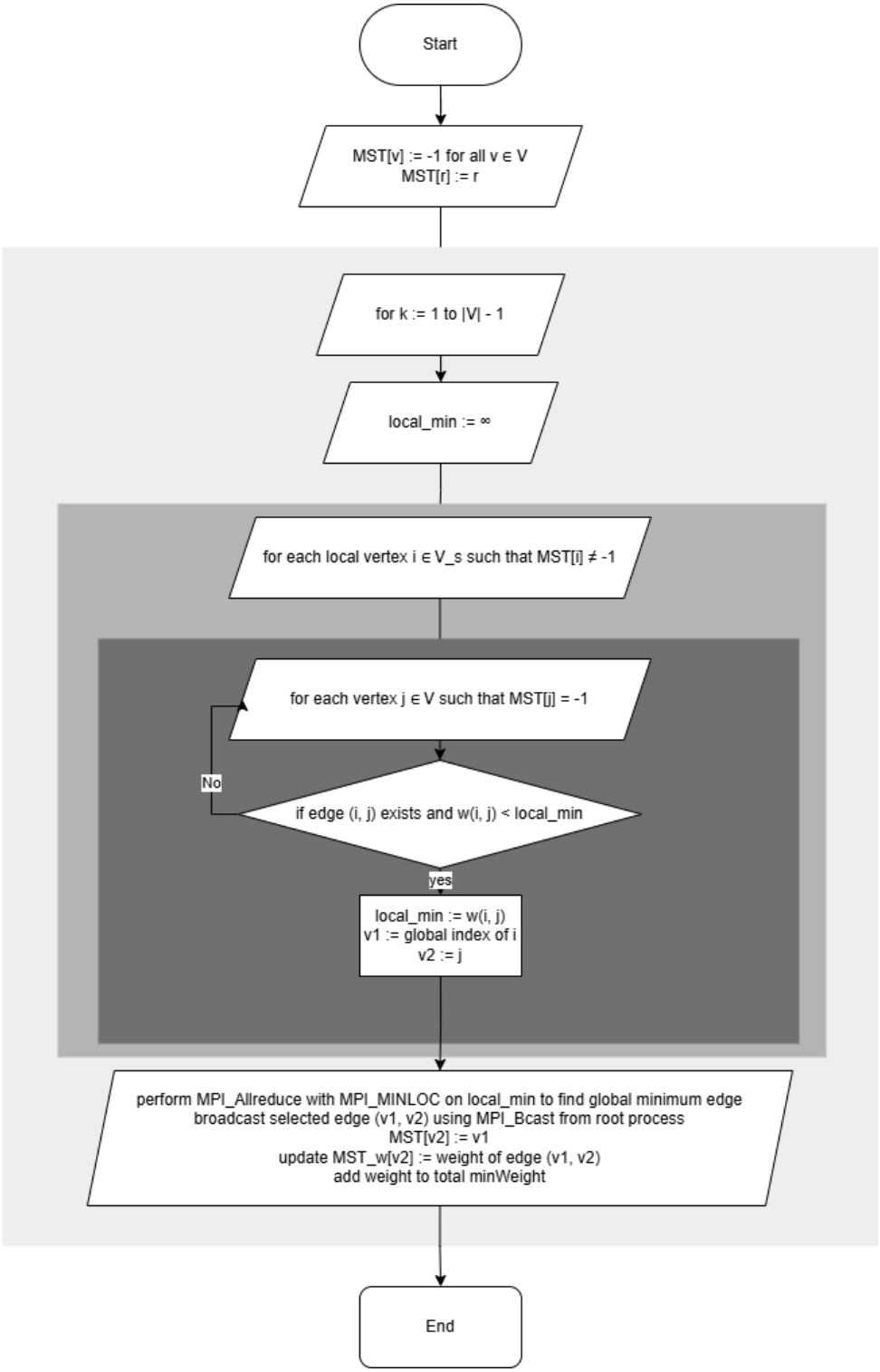
W klasycznej wersji Prima w każdej iteracji szukana jest najmniejsza krawędź wychodząca z już zbudowanego MST i dodawany jest wierzchołek. Algorytm można zrównoleglić dzieląc macierz sąsiedztwa grafu pomiędzy działające procesy. W każdym kroku procesor wybiera lokalnie (w podzielonej macierzy sąsiedztwa, czyli w podzielonych krawędziach grafu) najbliższy węzeł, a następnie przeprowadza globalną redukcję w celu wybrania globalnie najbliższego węzła. Ten węzeł jest wstawiany do MST, a wybór jest rozgłaszany do wszystkich procesorów.

```
1.  procedure PARALLEL_PRIM_MST(V,  E, w)
2.  begin
3.  initialize  MST[v] := -1 for all v ∈ V
4.  initialize  MST[r] := r  // start from root r
5.  for k := 1 to |V| - 1 do
6.    local_min := ∞
7.    for each local vertex i ∈ V_s such that MST[i] ≠ -1 do
8.      for each vertex j ∈ V such that MST[j] = -1 do
9.        if edge (i, j) exists and w(i, j) < local_min then
10.         local_min := w(i, j)
11.         v1 := global index of i
12.         v2 := j
13.   perform MPI_Allreduce with MPI_MINLOC  on local_min to find global minimum  edge
14.   broadcast selected edge (v1, v2) using MPI_Bcast from root process
15.   MST[v2] := v1
16.   update MST_w[v2] := weight of edge (v1, v2)
17.   add weight to total  minWeight
18. end PARALLEL_PRIM_MST
```

Rysunek 3: Algorytm przedstawiający działanie równoległego algorytmu Prima.

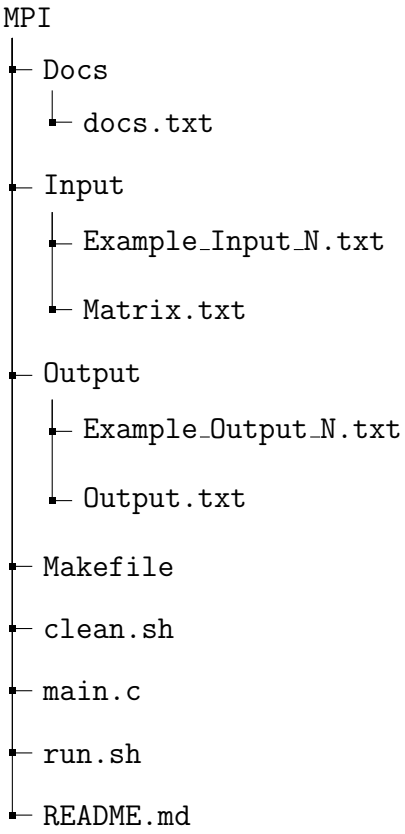
Wyjaśnienie:

- V_s oznacza lokalny zbiór wierzchołków przydzielony do danego procesu.
- $MPI\_Allreduce$ zbiera lokalne minimum z każdego procesu i wybiera globalne minimum.
- MPI_Bcast rozsyła wybraną krawędź do wszystkich procesów.
- $MST[v]$ przechowuje poprzednika wierzchołka v w drzewie MST.
- $MST\_w[v]$ to waga krawędzi łączącej v z jego poprzednikiem w MST.



Rysunek 4: Schemat blokowy równoległego algorytmu Prima

2 Budowa projektu



Na rysunku 2 przedstawiona została struktura projektu.

- Docs/
  - docs.txt - Zawiera dokumentację projektu – opis działania algorytmu Prima, sposób podziału pracy między procesy MPI, format danych wejściowych/wyjściowych oraz założenia projektowe i decyzje implementacyjne.
- Input/
  - Pliki wejściowe (grafy zapisane w formacie macierzy sasiedztwa) potrzebne do działania programu – np. reprezentację grafów (listy sąsiedztwa, macierze wag, itp.), które będą analizowane przez algorytm Prima.
- Output/
  - Pliki wyjściowe z wynikami działania programu takie jak minimalne drzewa rozpinające, czas wykonania.
- Makefile - Umożliwia automatyczną kompilację programu przy użyciu komendy make. Zawiera instrukcje, jak kompilować main.c z odpowiednimi flagami dla MPI.
- clean.sh - Skrypt do czyszczenia projektu. Usuwa pliki wynikowe, pliki tymczasowe, skompilowane pliki binarne itp., przywracając projekt do stanu "czystego".
- main.c - Główny plik źródłowy, w którym znajduje się implementacja algorytmu Prima z wykorzystaniem MPI. Odpowiada za:

  - Wczytanie danych wejściowych,
  - Podział pracy między procesy,
  - Obliczenie MST przy użyciu algorytmu Prima w sposób równoległy,
  - Zapis wyników.

- run.sh - Skrypt do przygotowania środowiska i uruchamiania algorytmu Prima w wersji równoległej MPI.
- README.md - Plik opisujący projekt, jego cel, sposób kompilacji (make), uruchomienia (run.sh), wymagania (np. MPI), oraz dodatkowe informacje dla użytkownika lub osoby przeglądającej repozytorium.

3 Działanie projektu

3.1 Zdefiniowane zmienne

W projekcie zdefiniowane zostały następujące zmienne:

- **int isMatrixRandom** - przchowuje informację, czy macierz sąsiedztwa będzie tworzona za pomocą liczb pseudolosowych (1), czy będzie wczytywana z pliku (0).
- **int size** - liczba procesów.
- **int rank** - numer procesu.
- **int* MatrixChunk** - część macierzy sąsiedztwa dla każdego procesu.
- **int mSize** - rozmiar macierzy sąsiedztwa.
- **int* displs** - pomocnicza tablica do rozdzielania macierzy.
- **int* sendcounts** - pomocnicza tablica do rozdzielania macierzy.
- **typedef struct { int v1; int v2;} Edge** - struktura do przechowywania krawędzi
- **int* MST_w** - wagi drzewa MST.
- **int* MST** - drzewo MST.
- **int minWeight** - łączna waga drzewa MST
- **FILE *f_matrix** - plik do odczytu.
- **FILE *f_output** - plik do zapisu.

3.2 Inicjalizacja MPI, odczytanie danych z pliku

W programie początkowo inicjalizowane jest MPI, w przypadku błędu, jest on obsługiwany, inicjalizowane są liczba procesów oraz ich numery, oraz seed liczb pseudolosowych, co widoczne jest w poniższym kodzie.

```
int errcode = MPI_Init(&argc, &argv);
handle_error(errcode, "MPI_Init");
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
srand(12345);
```

Następnie w zerowym procesie wczytywany jest rozmiar macierzy z pliku (jeśli się da), albo ustawiany domyślny (64). Sprawdzana jest poprawność pliku i w razie problemów macierz będzie generowana losowo.

```
if (rank == 0) {
    // Otwieranie plikow
    f_output = fopen("Output/Output.txt", "a");
    f_matrix = fopen("Input/Matrix.txt", "r");

    // Proba wczytania rozmiaru macierzy z pliku
```

```
    if (f_matrix && fscanf(f_matrix, "%d", &mSize) == 1) {
        // Sprawdzenie czy rozmiar jest prawidlowy i czy matrix
           jest losowy
        ...
    } else {
        mSize = 64;
        isMatrixRandom = 1; // macierz bedzie generowana losowo
    }

    fclose(f_matrix);
}
```

Ostatcznie wszystkie prcesy otrzymują informację o rozmiarze macierzy.

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

3.3 Podział macierzy sąsiedzctwa na węzły, inicjalizacja macierzy sąsiedzctwa

Następnie obliczne są *sendcounts* i *displs*. *Sendcounts[i]* definiuje ile wierszy dostanie proces *i*, a *displs[i]* definiuje indeks, od którego zaczyna się jego fragment.

```
int baseRows = mSize / size;
int extraRows = mSize % size;
for (int i = 0; i < size; ++i) {
    sendcounts[i] = baseRows + (i < extraRows ? 1 : 0);
    displs[i] = (i == 0) ? 0 : displs[i - 1] + sendcounts[i - 1];
}
```

Następnie w procesie 0 inicjalizowna jest macierz sąsiedzctwa. Jeśli *isMatrixRandom == 1* generowany jest losowy graf symetryczny bez pętli. W innym wypadku następuje próba odczytu macierzy z pliku.

```
if (rank == 0) {
    matrix = malloc(...);
    if (isMatrixRandom) {
        // Generowanie losowej symetrycznej macierzy grafu
    } else {
        // Wczytanie z pliku
    }
}
```

Proces 0 drukuje macierz na ekran. Ostatecznie następuje rozdzielanie macierzy między procesy (MPI_Scatterv). Każdy proces otrzymuje odpowiedni fragment (swoje wiersze) macierzy sąsiedztwa.

```
MatrixChunk = malloc(...);

MPI_Datatype matrixString;
MPI_Type_contiguous(mSize, MPI_INT, &matrixString);
MPI_Type_commit(&matrixString);

MPI_Scatterv(matrix, sendcounts, displs, matrixString,
   MatrixChunk, sendcounts[rank], matrixString, 0, MPI_COMM_WORLD)
   ;
```

3.4 Algorytm Prima

Algorytm Prima służy do znajdowania minimalnego drzewa rozpinającego (MST) w grafie ważonym. Poniższy opis dotyczy implementacji równoległej z użyciem MPI:

1. **Inicjalizacja:**
   - Każdy proces posiada swoją część macierzy sąsiedztwa — `MatrixChunk`.
   - Zmienna `MST[]` przechowuje informację, które wierzchołki są już w MST.
   - `sendcounts[]` i `displs[]` określają rozkład danych pomiędzy procesami.
2. **Pętla główna:** dla każdej z $mSize - 1$ krawędzi MST:
   (a) Inicjalizujemy zmienne: `min = INT_MAX`, `v1 = -1`, `v2 = -1`.
   (b) Przeglądamy wszystkie wierzchołki lokalne procesu (`i`) z aktualnie zbudowanego MST:
      - Dla każdego nieodwiedzonego wierzchołka `j` sprawdzamy, czy krawędź (i, j) ma mniejszy koszt niż obecne minimum.
      - Jeśli tak, aktualizujemy `min`, `v1` i `v2`.

   (c) Tworzymy strukturę `row`, zawierającą lokalne minimum i rangę procesu.
   (d) Wykonujemy `MPI_Allreduce` z operatorem `MPI_MINLOC`, aby znaleźć globalne minimum i rangę procesu, który je znalazł.
   (e) Następnie rozgłaszamy (broadcast) znalezioną krawędź do wszystkich procesów przy użyciu `MPI_Bcast`.
   (f) Aktualizujemy tablicę `MST[]` — oznaczamy, że wierzchołek `v2` należy teraz do MST.
   (g) Zapisujemy wagę tej krawędzi w `MST_w[]` i dodajemy ją do całkowitej wagi drzewa `minWeight`.

```
//Perform Prim's algorithm to find MST
for (int k = 0; k < mSize - 1; ++k) {
    int min = INT_MAX, v1 = -1, v2 = -1;
    for (int i = 0; i < sendcounts[rank]; ++i) {
        if (MST[i + displs[rank]] != -1) {
            for (int j = 0; j < mSize; ++j) {
                if (MST[j] == -1 && MatrixChunk[mSize * i + j] <
                   min && MatrixChunk[mSize * i + j] != 0) {
                    min = MatrixChunk[mSize * i + j];
                    v1 = i + displs[rank];
                    v2 = j;
                }
            }
        }
    }
    row.min = min;
    row.rank = rank;
    MPI_Allreduce(&row, &minRow, 1, MPI_2INT, MPI_MINLOC,
       MPI_COMM_WORLD);
```

```
18
19      edge.v1 = v1;
20      edge.v2 = v2;
21      MPI_Bcast(&edge, 1, MPI_2INT, minRow.rank, MPI_COMM_WORLD);
22
23      MST[edge.v2] = edge.v1;
24      MST_w[edge.v2] = minRow.min;
25      minWeight += minRow.min;
26  }
```

3.5 Zapis wyniku

Poniższy fragment kodu odpowiada za zapis wyników obliczeń algorytmu Prima (minimalne drzewo rozpinające, MST) do pliku oraz wyświetlenie wyników na ekranie. Opis wykonania tych operacji jest następujący:

1. **Sprawdzenie ranku procesu:**
   - Tylko proces o `rank == 0` (zwykle proces nadrzędny lub główny) wykonuje zapis wyników.
   - Pozostałe procesy (o innych rangach) nie biorą udziału w zapisie do pliku ani wyświetlaniu wyników.
2. **Otwieranie pliku do zapisu:**
   - Funkcja `fopen` otwiera plik `Output/Output.txt` w trybie `a`, co oznacza, że wyniki będą dopisywane na końcu pliku.
   - Jeśli plik nie może zostać otwarty (np. z powodu braku uprawnień), wyświetlany jest komunikat o błędzie.
3. **Zapis do pliku:**
   - Jeżeli plik został pomyślnie otwarty, zapisujemy do niego następujące dane:
     - `minWeight` — minimalna waga MST.
     - Krawędzie MST, zawierające wierzchołek początkowy `i`, jego rodzica `MST[i]` oraz wagę krawędzi `MST_w[i]`.
     - Dodatkowe informacje: liczba procesorów (`size`), liczba wierzchołków (`mSize`), czas wykonania (`calc_time`) oraz całkowita waga MST.
   - Na końcu zamykamy plik za pomocą funkcji `fclose`.
4. **Wyświetlanie wyników na ekranie:**
   - Następnie, proces o `rank == 0` wyświetla wyniki na ekranie w formie:
     - Krawędzie MST z ich wagami.
     - Całkowitą wagę MST.
     - Czas wykonania algorytmu.

```
if (rank == 0) {
    f_output = fopen("Output/Output.txt", "a");
    if (f_output == NULL) {
        printf("Error in Opening Output File!\n");
    } else {

        fprintf(f_output, "The min weight is %d\n", minWeight);
        for (int i = 0; i < mSize; ++i) fprintf(f_output, "Edge %
           d %d - weight %d \n", i, MST[i], MST_w[i]);
        fprintf(f_output, "\nProcessors: %d\nVertices: %d\
           nExecution Time: %f\nTotal Weight: %d\n\n", size, mSize
           , calc_time, minWeight);


        fclose(f_output);
    }

    printf("\nMST Result (Rank 0):\n");
    for (int i = 0; i < mSize; ++i) {
        printf("Vertex %d -> Parent %d - Weight: %d\n", i, MST[i
           ], MST_w[i]);
    }
    printf("Total Weight: %d\n", minWeight);
    printf("\nExecution Time: %f seconds\n", calc_time);
}
```

4 Instrukcja obsługi projektu

Projekt przygotowany jest w ten sposób aby działał prawidłowo w sali 204 w *stud-204*. Znajdując się w katalogu głównym projektu można skompilować i uruchomić program wykorzystując plik *Makefile*, lub skorzystać z gotowych skryptów *run.sh* i *clean.sh*.

W przypadku uruchomiania projektu ręcznie za pomocą *Makefile* neleży:

1. Wykonać polecenia: *source /opt/nfs/config/source_mpich430.sh* i *source /opt/nfs/config/source_cuda121.sh* w celu zainicjalizowania MPI.
2. Wykonać polecenie: */opt/nfs/config/station204_name_list.sh 1 16 > nodes* zapisujące do pliku *nodes* informacje o dostępnych węzłach.
3. Wykonać polecenie: *make* kompilujące program do plików wykonywalnch.
4. Wpisać w plik *InputMatrix.txt* wyłącznie jedną liczbę całkowitą w pierwszej linijce, definiującą rozmiar macierzy sąsiedztwa.
5. Wykonać polecenie: *make post_build ARG=ilosc_wezlow* odpowiedzialne za uruchomienie programu. Do argumentu ARG należy wpisać ilość procesów na której program będzie się wykonywał.
6. Wynik programu widoczny będzie w pliku *Output/Output.txt*

7. Wykonać polecenie: *make clean* w celu wyczyszczenia projektu z plików wykonywalnych.

W przypadku wykorzystania skryptów textitrun.sh i *clean.sh* należy:

1. Wpisać w plik *InputMatrix.txt* wyłącznie jedną liczbę całkowitą w pierwszej linijce, definiującą rozmiar macierzy sąsiedztwa.

2. Wykonać polecenie *./run.sh*, które zainicjalizuje MPI, zapisze do pliku *nodes* odpowiednie informacje o dostępnych węzłach, skompiluje program oraz wywoła go dla 3, 8 i 16 węzłów.

3. Wynik programu widoczny będzie w pliku *Output/Output.txt*

4. Wykonać polecenie: *./clean.sh* w celu wyczyszczenia projektu z plików wykonywalnych i pliku *Output.txt*.

Poza tym w pliku *InputMatrix.txt* poza wpisaniem rozmiaru macierzy sąsiedztwa, można ją ręcznie zdefiniować w następnych linijkach, przykładowo, w następujący sposób:

0	5	0	9	0	0	3	0
5	0	9	0	8	6	0	7
0	9	0	9	4	0	5	3
9	0	9	0	0	0	8	0
0	8	4	0	0	2	1	0
0	6	0	0	2	0	6	0
3	0	5	8	1	6	0	9
0	7	3	0	0	0	9	0

Macierz powinna być zdefiniowana przez użytkownika w sposób prawidłowy.

5 Podsumowanie

W ramach projektu przeprowadzono implementację zrównoleglonego algorytmu Prima z wykorzystaniem biblioteki MPI. Celem było przyspieszenie wyznaczania minimalnego drzewa rozpinającego (MST) w grafach o dużej liczbie wierzchołków i krawędzi.

Przeprowadzone testy wykazały, że zrównoleglenie algorytmu przyniosło zauważalny wzrost wydajności, szczególnie dla większych grafów. Dla małych instancji różnice czasowe były nieznaczne, co jest zrozumiałe ze względu na narzut komunikacyjny między procesami. Jednak w przypadku grafów o rozmiarze rzędu kilku tysięcy wierzchołków, czas wykonania zmniejszał się wraz ze wzrostem liczby procesów, aż do momentu, gdy dalsze zwiększanie liczby procesów przestawało być opłacalne (ze względu na rosnący koszt synchronizacji i komunikacji).

Efektywność równoległej wersji algorytmu była zależna m.in. od struktury grafu oraz strategii podziału danych. Lepsze rezultaty uzyskano w przypadku grafów gęstych, gdzie koszt obliczeń dominował nad kosztem komunikacji.

Literatura

[1] https://www.cs.purdue.edu/homes/ayg/book/Slides/chap10_slides.pdf