

Instalacja MS-MPI

1. Wchodzimy na stronę: <https://www.microsoft.com/en-us/download/details.aspx?id=100593&fbclid=IwAR0jEVTBjzVB27rUUbGRkadJDy3GX576R9cY6UEBNMVbfZ3PWWVRREnpXNk>
2. Pobieramy i instalujemy oba pakiety:

Choose the download you want

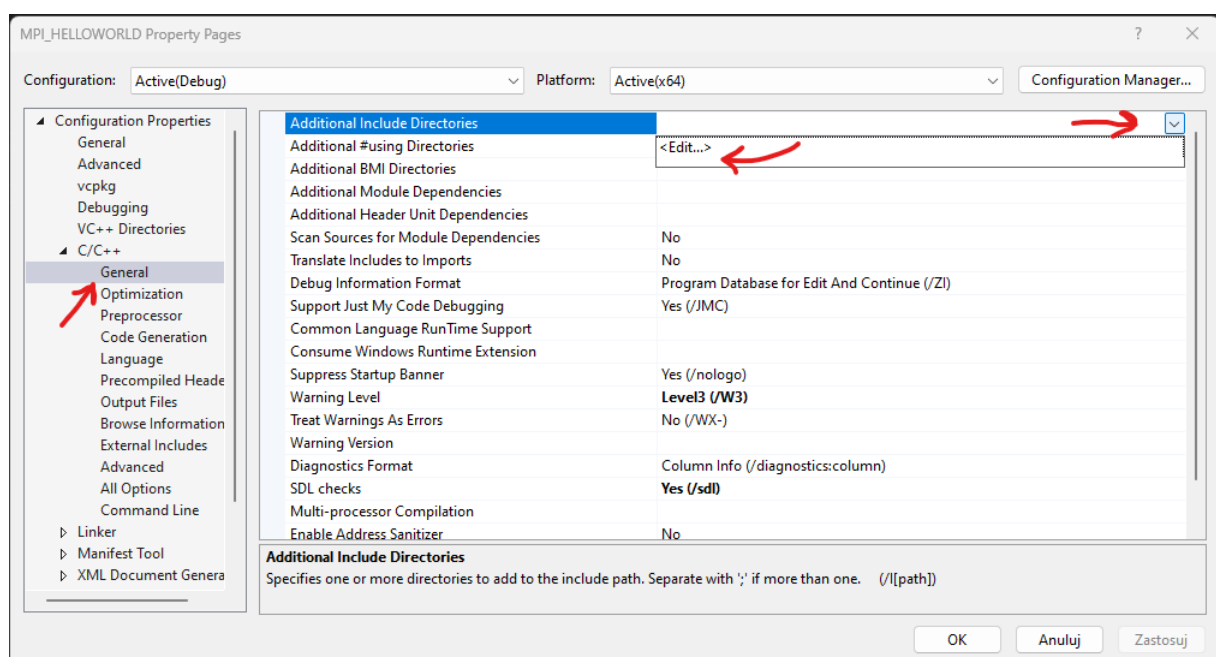
<input type="checkbox"/> File Name	Size
<input type="checkbox"/> msmtpisetup.exe	7.5 MB
<input type="checkbox"/> msmtpisdk.msi	2.2 MB

Narzędzia MS-MPI powinny zainstalować się w katalogu:

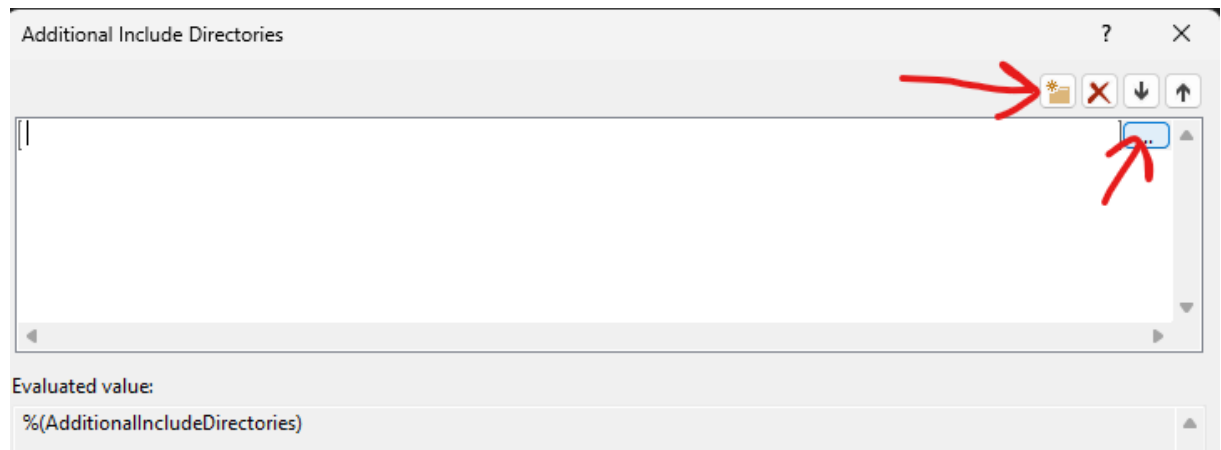
C:\Program Files (x86)\Microsoft SDKs\MPI

Przygotowanie projektu

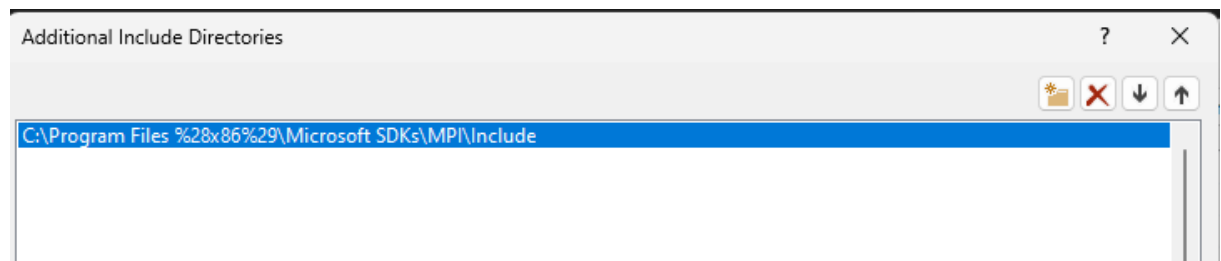
1. W Visual Studio tworzymy nowy projekt konsolowy C++
2. Po utworzeniu klikamy prawym przyciskiem myszy na nawie projektu
3. Wskazujemy programowi Visual Studio miejsce, z którego ma pobierać pliki nagłówkowe MPI:



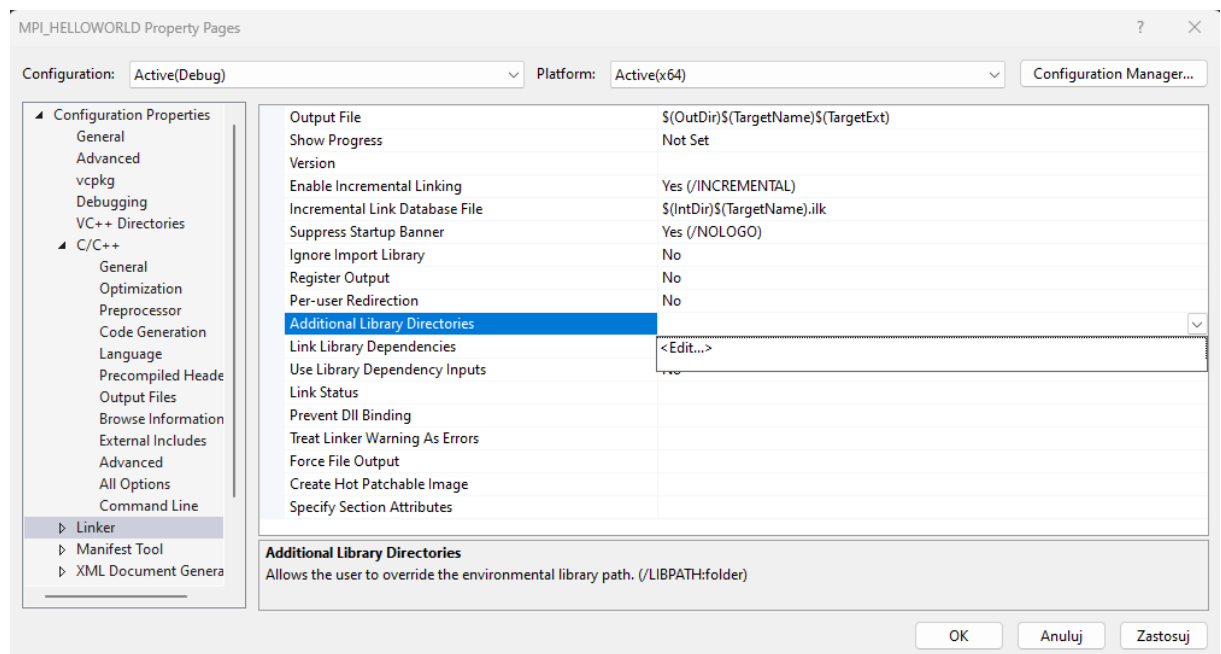
4. Wskazujemy **KATALOG** C:\Program Files (x86)\Microsoft SDKs\MPI\Include



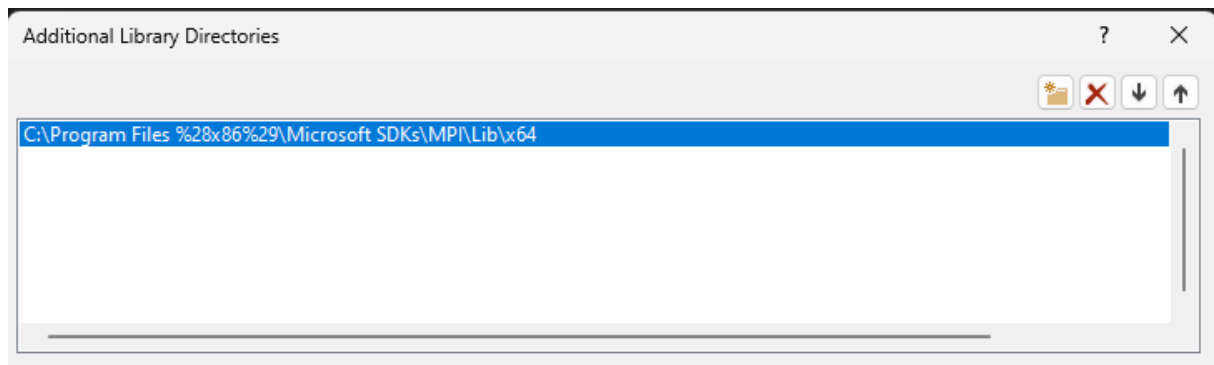
Po dodaniu:



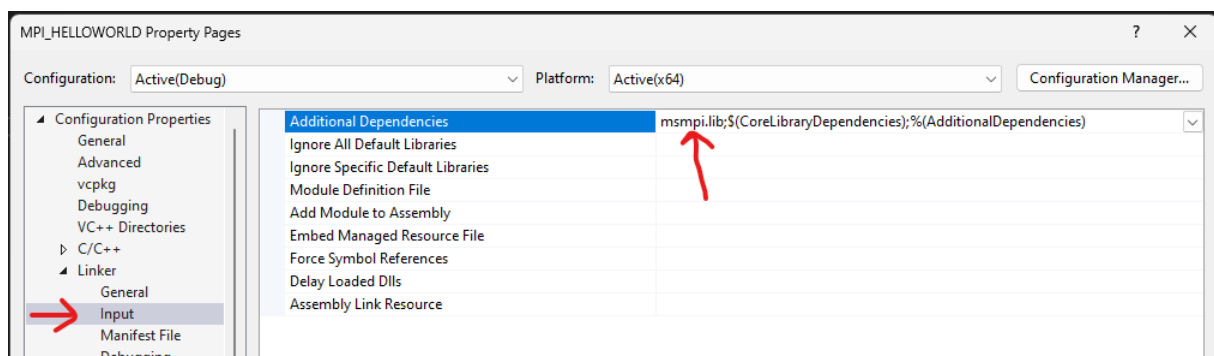
5. Teraz w analogiczny sposób wskazujemy katalog z bibliotekami, które będą dołączane do aplikacji w momencie linkowania:



6. Wskazujemy **KATALOG** C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64



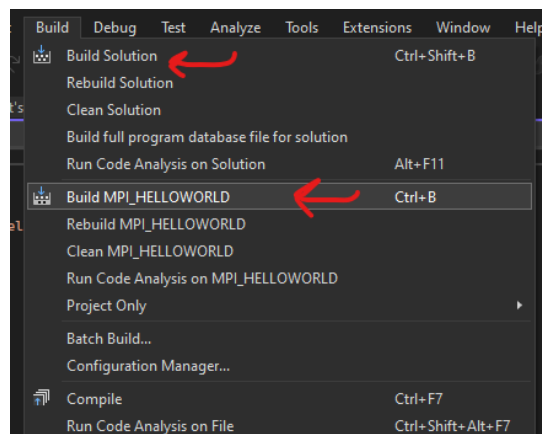
7. Do danych wejściowych linkera dopisujemy **msmpi.lib**;



Tworząc nowy projekt, musimy za każdym razem wskazać te dwa katalogi programowi Visual Studio.

Kompilacja i uruchomienie:

Projekt kompilujemy za pomocą jeden z tych opcji:




Projekt uruchamiamy **ZAWSZE** z widoku konsoli:

1. Przechodzimy do katalogu z plikiem *.exe
2. Wprowadzamy polecenie:

`mpiexec -n ilość_procesów nazwaprogramu.exe`

```
...MPI_HELLOWORLD\x64\Debug>mpiexec -n 5 MPI_HELLOWORLD.exe
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
...MPI_HELLOWORLD\x64\Debug>
```



Tutaj znajduje się
plik *.sln

Hello World MPI

MPI to technologia, która pozwala wykonywać program na wielu maszynach. Jako, że nie jesteśmy w stanie sobie na to pozwolić będziemy operować na procesach. Każdy proces będzie udawał osobną maszynę/komputer.

Flaga -n służy do wskazywania ile procesów będzie brało udział w obliczeniach. Jak widać na powyższym obrazku uruchomione zostało 5 procesów. Każdy z nich wykonał prosty Hello World zgodnie z poniższym kodem źródłowym:

```
#include "mpi.h"
#include <stdio>

int main()
{
    MPI_Init(&argc, &argv); //początek sekcji równoległej

    printf(_Format: "Hello, world!");

    MPI_Finalize(); //koniec sekcji równoległej

    return 0;
}
```

Funkcja `MPI_Init()` przyjmuje dwa parametry: wskaźnik na `argc` i `argv` z nagłówka funkcji `main`. Jeżeli chcielibyśmy przekazać parametry do programu z poziomu polecenia wywołującego tak jak ma to miejsce w tradycyjnym C/C++ musielibyśmy te parametry wskazać.

Ilość procesów oraz ich ID

Pisząc kod wielowątkowy np. za pomocą `std::thread` każdy wątek wykonywał pewien fragment całości zadania. Bardzo często wątki określały przypisaną im porcję zadania na bazie swojego ID (które to w tamtym przypadku nadawaliśmy ręcznie) oraz samej ilości otwartych wątków.

MS-MPI dostarcza API, które pozwala odczytać ID procesu i ilość procesów:

```
Hello, world from process 0/5!  
Hello, world from process 2/5!  
Hello, world from process 3/5!  
Hello, world from process 1/5!  
Hello, world from process 4/5!
```

Ilość to oczywiście wartość wskazana za pomocą przełącznika `-n`

ID procesu to liczba od 0 do ilość-1

```
int main()  
{  
    int PID, PCOUNT;  
  
    MPI_Init(&argc, &argv);  
  
    MPI_Comm_rank(MPI_COMM_WORLD, &PID);  
    MPI_Comm_size(MPI_COMM_WORLD, &PCOUNT);  
  
    printf(_Format: "Hello, world from process %d/%d!", PID, PCOUNT);  
  
    MPI_Finalize();  
  
    return 0;  
}
```

Pobranie obu wartości realizowane jest za pomocą funkcji:

MPI_Comm_rank() oraz **MPI_Comm_size()**

Pierwszy parametr obu funkcji to tak zwany **world** czyli grupa w jakiej procesy się znajdują. Jest to po prostu liczba identyfikująca tę grupę. My jeszcze grup nie tworzyliśmy, więc każdy proces znajduje się w tym samym domyślnym świecie.

MPI_COMM_WORLD to makro na liczbę `0x44000000` (autorzy tak oznaczyli domyślny świat). Tworząc grupy możemy je później identyfikować dowolnie np. 0

ZAUWAŻMY CO SIĘ STANIE GDY URUCHOMIMY PROGRAM WIĘCEJ NIŻ RAZ

Najważniejsze funkcje MS-MPI:

- `int MPI_Comm_rank(MPI_Comm world, int* rank);`
 - `world` – identyfikator świata/grupy, domyślny to `MPI_COMM_WORLD`
 - `rank` – wskaźnik na zmienną, do której zostanie zapisane id
- `int MPI_Comm_size(MPI_Comm world, int* size);`
 - `world` – identyfikator świata/grupy
 - `size` – wskaźnik na zmienną, do której zostanie zapisana ilość procesów
- `int MPI_Send(void* buffer, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);`
 - `buffer` – wskaźnik na zmienną/tablicę do wysłania
 - `count` – ilość elementów do wysłania (ilość elementów typu wskazanego w nst. parametrze)
 - `type` – typ danych do wysłania
 - `dest` – id celu
 - `tag` – identyfikator podgrupy
 - `comm` – identyfikator świata
- `int MPI_Recv(void* buffer, int count, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Status* status);`
 - `buffer` – wskaźnik na zmienną/tablicę do wysłania
 - `count` – ilość elementów do wysłania (ilość elementów typu wskazanego w nst. parametrze)
 - `type` – typ danych do wysłania
 - `dest` – id celu
 - `tag` – identyfikator podgrupy
 - `comm` – identyfikator świata
 - `status` – wskaźnik na obiekt struktury `MPI_Status` gdzie zapisane zostaną informacje o statusie transmisji
- `int MPI_Isend(void* buffer, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request* request);` //wysyłanie asynchroniczne
 - `buffer` – wskaźnik na zmienną/tablicę do wysłania
 - `count` – ilość elementów do wysłania (ilość elementów typu wskazanego w nst. parametrze)
 - `type` – typ danych do wysłania
 - `dest` – id celu
 - `tag` – identyfikator podgrupy
 - `comm` – identyfikator świata
 - `request` – wskaźnik na strukturę `MPI_Request`, w której zapisane zostaną informacje o transmisji asynchronicznej
- `int MPI_Irecv(void* buffer, int count, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Status* status, MPI_Request* request);` //odbiór asynchroniczny
 - `buffer` – wskaźnik na zmienną/tablicę do wysłania

- count – ilość elementów do wysłania (ilość elementów typu wskazanego w nst. parametrze)
 - type – typ danych do wysłania
 - dest – id celu
 - tag – identyfikator podgrupy
 - comm – identyfikator świata
 - request – wskaźnik na strukturę MPI_Request, w której zapisane zostaną informacje o transmisji asynchronicznej
- int MPI_Waitall(int count, MPI_Request* requests, MPI_Status* statuses); //wymusza poczekanie na zdarzenie asynchroniczne
 - count – ilość zdarzeń
 - requests – wskaźnik na tablicę requestów (requesty te otrzymujemy z funkcji asynchronicznych)
 - statuses – wskaźnik na tablicę statusów
 - int MPI_Bcast(void* buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);
//transmisja broadcast (UWAGA! Również do samego siebie)
 - buffer – wskaźnik na dane
 - count - ilość danych typu wskazanego dalej
 - type - typ danych
 - root – id procesu źródła
 - comm – świat

Operacja synchroniczna – operacja, której wywołanie blokuje wątek/proces do momentu jej zakończenia

Operacja asynchroniczna – operacja, która nie blokuje wątku, ponieważ jej wywołanie powoduje jedynie zlecenie tej operacji

Jak rozumieć operację asynchroniczną?

Wyobraźmy sobie, że chcemy odebrać wynik od 10 procesów. Wywołujemy funkcję odbierającą synchronicznie od procesu 1. Gdy odbiór się zakończy wykonujemy synchroniczny odbiór od procesu 2 itd. Podejście to jest bardzo złe. Po pierwsze nie mamy pojęcia, który proces zakończył działanie pierwszy (dobrze było by wykonać odbiór w takiej samej kolejności jak procesy ukończą swoje zadanie). Po drugie marnujemy tonę czasu, bo jeżeli proces 5 zakończy działanie przed 1 to będzie musiał czekać na 1, potem 2, potem 3 itp. Co oczywiście jest marnowaniem czasu.

Asynchroniczny odbiór jest lepszy, ponieważ zleca on tylko odbiór. Nie czeka na jego zakończenie. Gdy zlecimy odbiór z procesu 1 możemy od razu uruchomić nasłuch danych od procesu 2 nawet jeżeli dane z 1 jeszcze nie dotarły.

Typy danych

Nazwa	Odpowiednik j. C/C++
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits
MPI_PACKED	dane spakowane / rozpakowane za pomocą MPI_Pack()/ MPI_Unpack

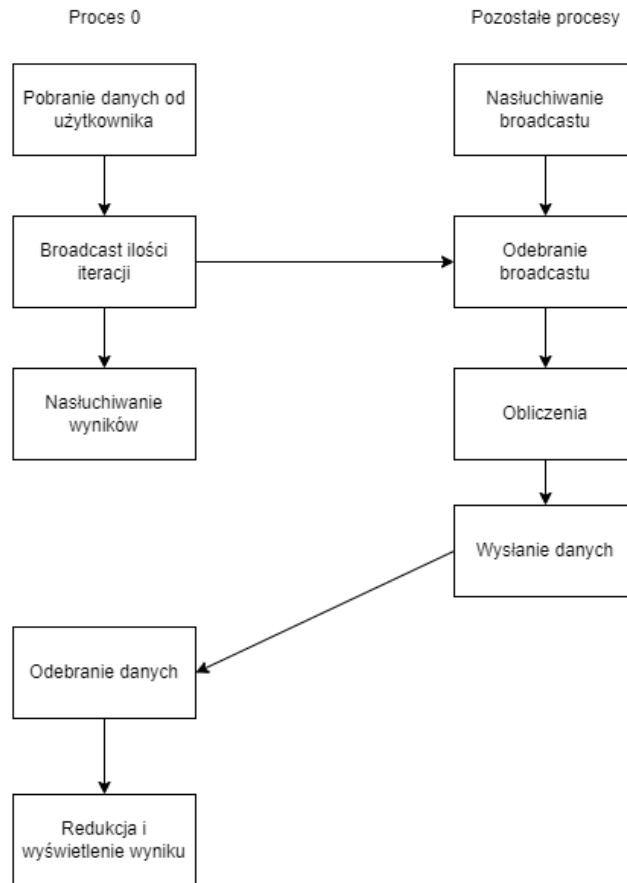
Tak jak wskazano to w opisie funkcji, metody MPI oczekują od programisty by ten podał ilość elementów do wysłania/odebrania (zazwyczaj parametr po wskaźniku na bufor) typu (zazwyczaj kolejny parametr po ilości).

Programując w C/C++ korzystając z różnych API najczęściej spotykamy się z sytuacją gdzie funkcja wymaga od nas wskaźnika na bufor oraz rozmiaru tego bufora w bajtach.

W przypadku tej implementacji MPI sytuacja jest inna. API wymaga od nas wskaźnika na bufor, ilości elementów oraz typu elementu. Mając ilość elementów i typ elementu funkcje są w stanie określić ilość bajtów do wysłania.

Architektura aplikacji MPI

Pisząc aplikację MPI (w naszym przypadku aplikację konsolową na procesach) by uzyskać zysk ze zrównoleglenia aplikację należy zaprojektować w odpowiedni sposób:



Pisząc aplikację wielowątkową mieliśmy do dyspozycji wątek główny oraz n wątków do obliczeń. Wątek główny zbierał wszystko w całość oraz komunikował się z użytkownikiem. W wątku głównym stawiane były punkty synchronizacji np. `join()` tak by wątek ten poczekał na wszystkie pozostałe.

W MPI sytuacja jest podobna, ale tutaj operujemy na procesach. Wybrany przez nas proces np. ten z $ID = 0$ jest procesem głównym. On komunikuje się z użytkownikiem (wypisuje dane, pobiera je od użytkownika).

Obliczenia wykonywane są na pozostałych procesach, ale też nie ma przeciwwskazań by ten główny także wykonał swoją część.

Na rysunku powyżej widnieje przykład architektury aplikacji:

- proces główny (id 0) zbiera informacje potrzebne do wykonania zadania
- pozostałe procesy nasłuchują danych
- proces główny wykonuje broadcast informacji do wszystkich procesów i uruchamia nasłuch wyników
- pozostałe procesy odbierają dane
- po wykonaniu obliczeń dane są odsyłane do procesu głównego

- proces główny wykonuje sumowanie (lub cokolwiek – zależne od problemu) i wyświetla wynik

Jak to wygląda w aplikacji:

Jako przykład posłużymy nam aplikacja, która sumuje dwa wektory.

```
void mainProcess() {  
    ...  
}  
  
void workerProcess(int id) {  
    ...  
}  
  
int main()  
{  
    int PID, PCOUNT;  
  
    MPI_Init(&argc, &argv);  
  
    MPI_Comm_rank(MPI_COMM_WORLD, &PID);  
    MPI_Comm_size(MPI_COMM_WORLD, &PCOUNT);  
  
    if (PID == 0) { //jestem procesem głównym  
        mainProcess();  
    }  
    else { //jestem procesem roboczym  
        workerProcess(PID);  
    }  
  
    MPI_Finalize();  
  
    return 0;  
}
```

W funkcji main umieszczono instrukcję warunkową, która dzieli kod na dwie ścieżki: ścieżkę dla proces głównego oraz ścieżkę dla procesów „roboczych”.

Sterowanie -> proces główny:

Alokujemy wektory i wypełniamy je losowymi danymi:

```
//alokujemy wektory o rozmiarze(5*(ilosc procesów-1))
unsigned int* va = new unsigned int[5 * (size - 1)];
unsigned int* vb = new unsigned int[5 * (size - 1)];
unsigned int* vc = new unsigned int[5 * (size - 1)];

//wypełniamy a i b losowymi danymi, a vc zerujemy
for (unsigned int i = 0; i < 5 * (size - 1); i++) {
    va[i] = rand() % 10;
    vb[i] = rand() % 10;
    vc[i] = 0;
}
```

Sterowanie -> procesy robocze

Alokujemy odpowiednie bufory w procesach roboczych:

```
//alokujemy bufor na moją część zadania
unsigned int* v = new unsigned int[5];

//alokujemy miejsce na wektor a oraz b
unsigned int* va = new unsigned int[5 * (size - 1)];
unsigned int* vb = new unsigned int[5 * (size - 1)];
```

Sterowanie -> proces główny

Wykonujemy broadcast całego wektora a do pozostałych procesów:

```
//broadcastujemy wektor a do pozostałych procesów
MPI_Bcast(buffer: va, count: 5 * (size - 1), datatype: MPI_UNSIGNED, root: 0, comm: MPI_COMM_WORLD);
```

Sterowanie -> procesy robocze:

W procesach roboczych odbieramy wektor a:

```
//nasłuchujemy bcasta wektora a
MPI_Bcast(buffer: va, count: 5 * (size - 1), datatype: MPI_UNSIGNED, root: 0, comm: MPI_COMM_WORLD);
```

Sterowanie -> proces główny:

Nadajemy wektor b:

```
//broadcastujemy wektor b do pozostałych procesów
MPI_Bcast(buffer: vb, count: 5 * (size - 1), datatype: MPI_UNSIGNED, root: 0, comm: MPI_COMM_WORLD);
```

Sterowanie -> procesy robocze:

Odbieramy wektor b:

```
//nasłuchujemy bcasta wektora b
MPI_Bcast(buffer: vb, count: 5 * (size - 1), datatype: MPI_UNSIGNED, root: 0, comm: MPI_COMM_WORLD);
```

Wykonujemy sumowanie:

```
//liczymy sumę
for (unsigned int i = 0; i < 5; i++) {
    v[i] = va[(id - 1) * 5 + i] + vb[(id - 1) * 5 + i];
}
```

Sterowanie -> proces główny:

Uruchamiamy asynchroniczne nasłuchiwanie rezultatów:

```
//odpalamy nasłuch
MPI_Request* requests = new MPI_Request[size - 1];
MPI_Status* statuses = new MPI_Status[size - 1];
for (unsigned int i = 0; i < size-1; i++) {
    MPI_Irecv(buf: vc + i * 5, count: 5, datatype: MPI_UNSIGNED, source: i + 1, tag: 0, comm: MPI_COMM_WORLD, request: &requests[i]);
}
MPI_Waitall(count: size - 1, array_of_requests: requests, array_of_statuses: statuses);
```

Sterowanie -> procesy robocze:

Wysyłamy wyniki do procesu głównego:

```
//odsyłamy wynik
MPI_Send(buf: v, count: 5, datatype: MPI_UNSIGNED, dest: 0, tag: 0, comm: MPI_COMM_WORLD);
```

Sterowanie -> proces główny:

Wypisujemy otrzymane dane oraz zwalniaamy pamięć:

```
//wypisujemy wyniki
for (unsigned int i = 0; i < (5 * (size - 1)); i++) printf(_Format: "%d\t", va[i]);
printf(_Format: "\r\n");
for (unsigned int i = 0; i < (5 * (size - 1)); i++) printf(_Format: "%d\t", vb[i]);
printf(_Format: "\r\n");
for (unsigned int i = 0; i < (5 * (size - 1)); i++) printf(_Format: "%d\t", vc[i]);
printf(_Format: "\r\n");

//zwalniamy pamięć
delete[] va;
delete[] vb;
delete[] vc;
delete[] requests;
delete[] statuses;
```

Sterowanie -> proces roboczy:

Zwalniamy pamięć:

```
//zwalniamy pamięć  
delete[] v;  
delete[] va;  
delete[] vb;
```

Koniec programu:

```
\\Mpcn... \MPI_HELLOWORLD\x64\Debug>mpiexec -n 3 MPI_HELLOWORLD.exe  
1      8      9      3      4      5      6      9      9      6  
8      4      9      8      2      3      3      5      4      4  
9      12     18     11     6      8      9      14     13     10
```

Program wypisuje wektora a, pod nim wektor b, a na samym dole wektor c.

Aby program miał sens, musimy uruchomić program z ilością wątków 2 lub więcej.

Podsumowanie:

- o programowanie rozproszone wymaga by programista rozumiał schematy znane z programowania wielowątkowego
- o istotne jest zrozumienie kiedy program się zatrzymuje np. w celu otrzymania danych
- o istotne jest zrozumienie koncepcji programowania asynchronicznego
- o **wszystkie funkcje MPI zwracają inta, jeżeli jest to 0 funkcja wykonała się z sukcesem (mała porada, na wypadek szukania czemu i gdzie coś nie działa)**

Zadania:

1. Przeanalizuj załączony kod.
2. Zmodyfikuj załączony kod tak by na jeden proces przypadało więcej elementów do zsumowania (obecnie jest 5)
3. Zmodyfikuj załączony program tak, by rozmiar wektorów był pobierany od użytkownika. Program musi wyliczyć jaka część danych przypada na każdy proces, oraz przekazać każdemu z nich tę informację.