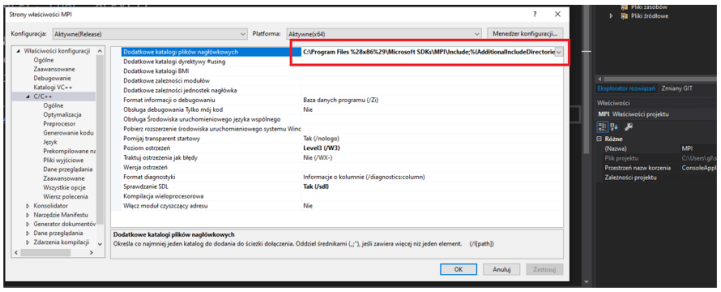
MPI na komputerze z systemem Windows

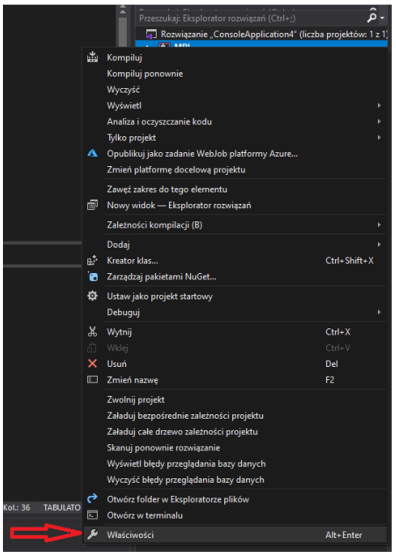
W celu uruchomienia programów stworzonych dla MS MPI należy doinstalować odpowiednie SDK : https://www.microsoft.com/en-us/download/details.aspx?id=100593 Instalujemy oba pliki z linku.

TWORZENIE PROJEKTU MPI

- Dodanie plików nagłówkowych i bibliotek do projektu - VisualStudio

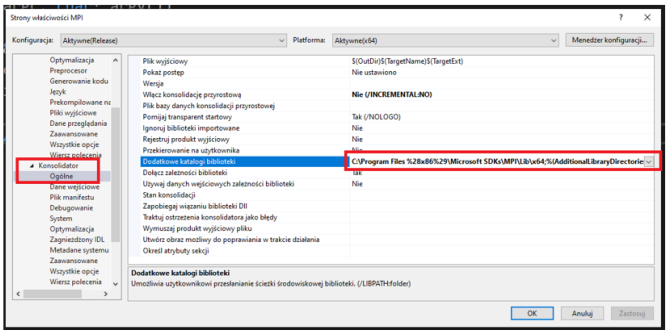


- Podajemy ścieżką do plików nagłówkowych MS-MPI:
  - W tym przypadku: C:\Program Files (x86)\Microsoft SDKs\MPI\Include



TWORZENIE PROJEKTU MPI

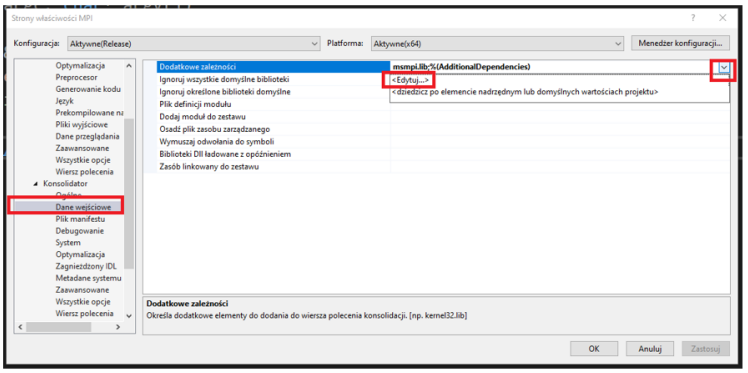
- Dodajemy biblioteki



- Podajemy ścieżką do bibliotek MS-MPI:
  - W tym przypadku: C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64

TWORZENIE PROJEKTU MPI

- Podajemy dodatkowe zależności dla danych wyjściowych w opcji „Konsolidator” umieszczając wpis: msmpi.lib



W razie wystąpienia błędu “vcruntime140d.dll” - rozwiązanie : https://www.youtube.com/watch?v=YYxMuidkR8I

Pierwszy program “Hello, world!”

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

int main(int argc, char* argv[])
{
        int id, count;
        MPI_Init(&argc, &argv);

        //pobranie id oraz ilości procesów do zmiennych
```

```
	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &count);

	//kod procesu komunikującego się z użytkownikiem
	if (id == 0) {
		printf("Hello, world from main process %u/%u!", id, count);
	}
	//kod procesów wykonujące jakieś działania
	else {
		printf("Hello, world from worker process %u/%u!", id, count);
	}

	MPI_Finalize();
	return 0;
}
```

Aby uruchomić nasz program kompilujemy go a następnie uruchamiamy w CMD:
mpiexec -n 4 NazwaNaszegoPliku.exe parametr1 parametr2

gdzie parametr1 i parametr2 to standardowe parametry programu C/C++ przypisywane do argv.
a 4 to liczba procesów.

Wstęp teoretyczny

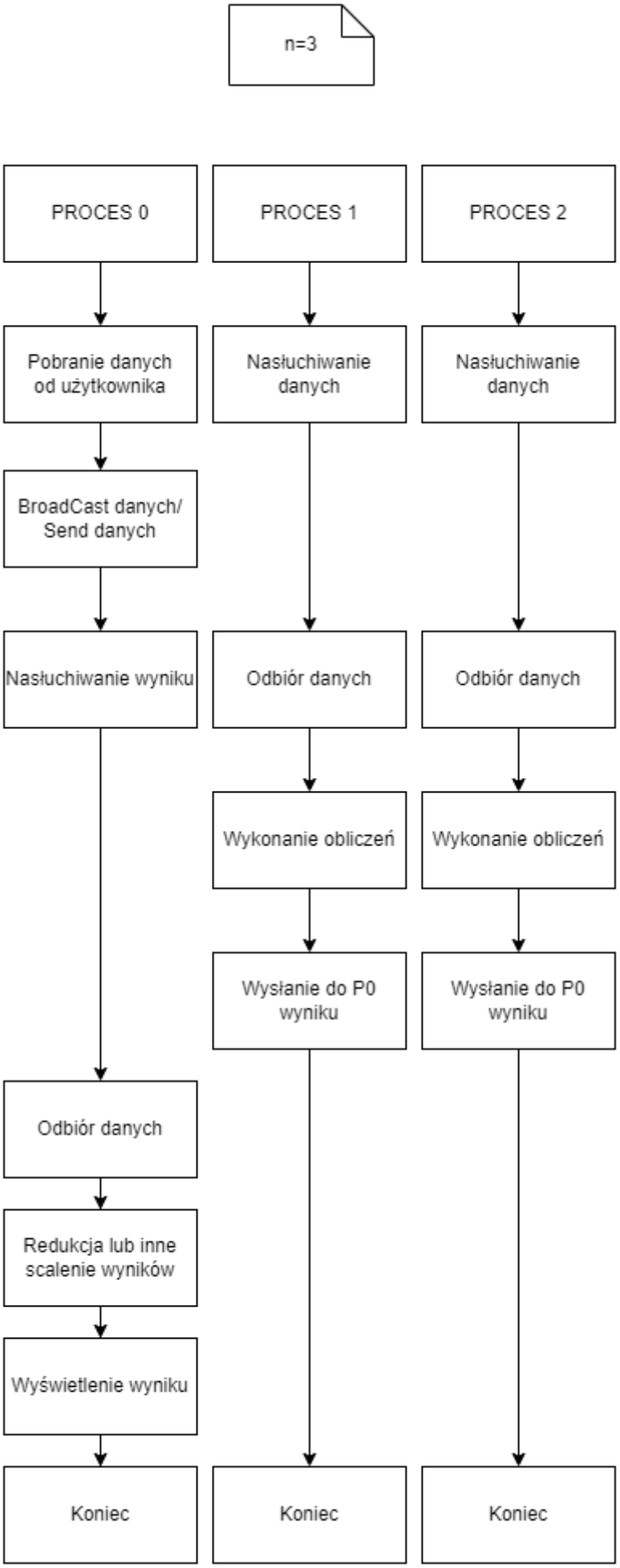
Programowanie z pomocą Microsoft MPI na jednym urządzeniu z systemem Windows przypomina programowanie wielowątkowe. Różnica polega na tym iż w przypadku wielowątkowości uruchamiamy nasz kod na N wątkach a w przypadku MPI uruchamiamy całą aplikację N razy.

W przypadku wielowątkowości bardzo często stosuje się mechanizm, który uzależnia wykonanie od ID wątku. Jeżeli ID = 0 wykonaj taki kod lub ten fragment pętli a jeżeli ID=1 ten fragment itd. W MPI schemat ten jest wykorzystywany cały czas gdyż kod, który napiszemy (cała aplikacja) zostanie uruchomiony N razy (N procesów). Programista w aplikacji musi zawrzeć informacje, który fragment kodu musi wykonać się w procesie o ID=0 a który gdy proces ma ID=1 itd.

Zazwyczaj do problemu podchodzi się tak, iż jeden proces o np ID=0 jest głównym procesem (komunikuje się z użytkownikiem) a pozostałe wykonują przydzielone im zadania.

UWAGA:
Aby użyć rand() w wielu procesach należy seed uzależnić od id procesu:

srand(time(NULL) + id);


n=3
PROCES 0
PROCES 1
PROCES 2
Pobranie danych od użytkownika
Nasłuchiwanie danych
Nasłuchiwanie danych
BroadCast danych/ Send danych
Nasłuchiwanie wyniku
Odbiór danych
Odbiór danych
Wykonanie obliczeń
Wykonanie obliczeń
Wysłanie do P0 wyniku
Wysłanie do P0 wyniku
Odbiór danych
Redukcja lub inne scalenie wyników
Wyświetlenie wyniku
Koniec
Koniec
Koniec

Powyższy przebieg ilustruje poniższą implementacje:

```cpp
#include "mpi.h"
#include <iostream>

int main(int argc, char* argv[])
{
    int id, count;
    MPI_Init(&argc, &argv);

    //pobranie id oraz ilości procesów do zmiennych
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &count);

    //kod procesu komunikującego się z użytkownikiem
    if (id == 0) {
        int iteration = 0;
        std::cout << "Podaj ilosc iteracji: " << std::endl;
        std::cin >> iteration;

        MPI_Bcast(&iteration, 1, MPI_INT, 0, MPI_COMM_WORLD); //proces 0 jest nadawcą

        int* results = new int[count - 1];
        MPI_Request* requests = new MPI_Request[count - 1];
        MPI_Status* statuses = new MPI_Status[count - 1];

        //wywołujemy nasłuch od każdego procesu oprócz samego siebie
        //asynchroniczne wywołanie pozwoli uruchomić nasłuch dla każdego
        //procesu bez potrzeby czekania na dane od poprzednika
        for (int32_t i = 0; i < count - 1; i++) {
            MPI_Irecv(&results[i], 1, MPI_DOUBLE, i + 1, 0, MPI_COMM_WORLD, &requests[i]);
        }
        MPI_Waitall(count - 1, requests, statuses);//oczekuje

        //tu można by było coś zrobić z tymi danymi
        //ale to tylko przykład

        printf("Koniec\r\n");

        delete[] results;
        delete[] requests;
        delete[] statuses;
    }
    //kod procesów wynokujących jakieś działania
    else {
        int iteration = 0;

        MPI_Bcast(&iteration, 1, MPI_INT, 0, MPI_COMM_WORLD); //pozostałe procesy czekają tu aż P0 nada wartość

        //jakieś operacje
        for (int i = 0; i < iteration; i++) {
            //cos tam
        }

        int dane = 1;
        MPI_Send(&dane, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

MPI API

Pobranie ID oraz liczby procesów

int id, count;
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &count);

MPI_Init, MPI_Finalize

- Przed użyciem metody z biblioteki MPI należy ja zainicjalizować za pomocą metody MPI_Init – należy do niej przekazać niezmodyfikowane parametry otrzymane w funkcji main albo NULL

- mpiexec przekazuje w ten sposób funkcji MPI_Init dodatkowe parametry

- Po zakończeniu obliczeń – zazwyczaj w końcowej części programu – należy uruchomić funkcję MPI_Finalize, która kończy działanie podsystemu MPI, m. in. zwalniając pamięć stosowana do buforowania przesyłanych komunikatów

- Przed wywołaniem MPI_Finalize należy zadbać o to, aby wszystkie wysyłane komunikaty zostały odebrane

MPI_Send

Składnia operacji nadania wiadomości jest następująca:

- **MPI_Send(buffer, count, datatype, dest, tag, comm)** gdzie:
- **buffer** – wskaźnik bufora z danymi do wysłania
- **count** – rozmiar danych do wysłania (ilość elementów typu datatype)
- **datatype** – typ danych umieszczonych w buforze
- **dest** – identyfikator (ranga) odbiorcy
- **tag** – znacznik rodzaju wiadomości
- **comm** – grupa (komunikator), do której należy odbiorca
- Jest to operacja blokująca – po jej zakończeniu można nadpisywać dane umieszczone w buforze **buffer**

MPI_Recv

Składnia operacji odbierania wiadomości jest następująca:

- **MPI_Recv(buffer, count, datatype, source, tag, comm, status)**
- Znaczenie parametrów jest analogiczne do operacji MPI Send, ponieważ wywołania te powinny być ze sobą sparowane
- Dodatkowy parametr status jest wskaźnikiem do struktury z identyfikatorem nadawcy oraz znacznikiem wiadomości, dodatkowo można pobrać z niej również rozmiar odebranego komunikatu
- Odebranie mniejszej liczby danych, niż określona parametrem count zaliczana jest jako poprawne wykonanie metody, natomiast błędem zakończy się próba odebrania wiadomości o większym rozmiarze, niż podany
- Podobnie jak MPI_Send jest to operacja blokująca – proces oczekuje na nadanie wiadomości przez nadawcę – potencjalne źródło zakleszczenia
- Status operacji :
- Operacja **MPI_Recv** i pokrewne zwracają dodatkowe informacje o wykonanej operacji, które zapisywane są w strukturze MPI Status

```
typedef struct {
int MPI_SOURCE;
int MPI_TAG;
int MPI_ERROR;
// dodatkowe pola
}
```

- **MPI_SOURCE** - ranga nadawcy wiadomości
- **MPI_TAG -** znacznik wiadomości
- **MPI_ERROR -** kod ewentualnego błędu
- Liczbę odebranych jednostek danych można sprawdzić za pomocą:

  int MPI_Get_count(MPI Status status, MPI Datatype zawartości, int count)

Typy danych MPI

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

Asynchroniczne wysyłanie komunikatów

- Składnia operacji nadania wiadomości jest następująca:

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

- Parametr request stanowi odnośnik do obiektu umożliwiającego sprawdzenie, czy operacja zakończyła się oraz ewentualne oczekiwanie na jej zakończenie
- Do oczekiwania na zakończenie wysyłania wiadomości służy funkcja MPI_Wait
- Do sprawdzenia, czy operacja zakończyła się służy funkcja MPI_Test
- Do czasu rzeczywistego zakończenia przekazywania wiadomości nie wolno modyfikować zawartości bufora wskazywanego przez buffer

Asynchroniczne odbieranie komunikatów

- Składnia operacji nadania wiadomości jest następująca:

MPI_Irecv(void * buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

- Parametr **request** pełni analogiczna role, jak w operacji **MPI_Isend**
- Do czasu rzeczywistego zakończenia przekazywania wiadomości nie wolno modyfikować zawartości bufora wskazywanego przez buffer

Przykład asynchronicznej wymiany komunikatów

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Status status;
const int TagLiczby = 1;
const int N = 10;
if (rank == 0) { // wysyłanie tablicy liczb
int* buf = new int[N];
for (int i = 0; i < N; ++i) {
buf[i] = 1 << i;
}
MPI_Request request;
MPI_Isend(buf, N, MPI_INT, 1, TagLiczby, MPI_COMM_WORLD, &request);
obliczenia();
if (MPI_Wait(&request, &status) == MPI_SUCCESS)
delete[] buf; // teraz mozna zwolnic pamiec dla bufora
}
else if (rank == 1) { // proc. 1. odbiera tablice
int* buf = new int[N];
MPI_Recv(buf, N, MPI_INT, 0, TagLiczby, MPI_COMM_WORLD, &status);
...
delete[] buf;
}
```

MPI_Bcast

int MPI_Bcast (void* buffer, int count, MPI_Datatype datatype, int rank, MPI_Comm comm)

- rank – ID nadawcy
- brak tag-u wiadomości
- wysyła również do siebie
- nie wymaga Recv (wywołanie = rozesłanie + odebranie)

Przykład:

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

Zadanie

Zaimplementuj rozproszone obliczenie przybliżenia liczby Pi. Skrypt wraz z dokumentacją opisującą go umieść w repozytorium w odpowiednim folderze.