

Marek, Kubicki

Programowanie równoległe.

Sprawozdanie z laboratorium 11,12,13 – MPI.

1. Celem tych zajęć było zapoznanie się z działaniem MPI(ang. *Message Passing Interface*).

MPI to protokół komunikacyjny odpowiedzialny za komunikację pomiędzy procesami, kompatybilny np. z językiem C.

Aby używać MPI (poza odpowiednimi poleceniami w kodzie), należy skompilować program używając **mpicc** i uruchomić używając **mpiexec**.

Aby poprawie używać MPI należy dzielić zadania pomiędzy procesy. Aby poprawnie rozdzielić zadania należy użyć „rank” procesów. Po wywołaniu polecenia **MPI_Comm_rank(MPI_COMM_WORLD, INT*)** ranga każdego procesu będzie zapisana do zmiennej podanej w drugim argumencie. Ranga to numer od 0 do ilości procesów, dzięki któremu można rozdzielić zadania np. poprzez utworzenie polecenia **for()** iterującego po różnych wartościach zależnie od rangi.

```
for(i=max_liczba_wyrazow*rank; i<max_liczba_wyrazow*rank +max_liczba_wyrazow; i++)
```

Innymi poleceniami nie przeznaczonymi (bezpośrednio) do komunikacji pomiędzy procesami są:

```
MPI_Comm_size( MPI_COMM_WORLD, int* )
```

Wykozystywane do otrzymania liczby procesów. Oraz:

```
MPI_Init( &argc, &argv );
```

```
MPI_Finalize();
```

Służące do inicjalizacji i zakończenia programu.

Po wykonaniu obliczeń trzeba połączyć wyniki, w tym celu można wykorzystać polecenia dostarczone przez MPI przeznaczone do komunikacji pomiędzy procesami.

```
int MPI_Send(void* buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Status *stat)
```

Blokujące wysyłanie dwupunktowe, polecenie **send** wysyła zmienną przekazaną w **buf** do procesu o randze **dest**. Polecenie **recv** odbiera wystaną wiadomość i zapisuje ją w zmiennej przekazanej w **buf**, można odbierać wiadomości od konkretnych procesów podając ich rangę do **src** lub od jakiegokolwiek procesu przekazując **MPI_ANY_SOURCE**.

```
int MPI_Pack( void* buf_dane, int count, MPI_Datatype typ, void* buf_send, int buf_send_size, int* pozycja, MPI_Comm comm )
```

```
int MPI_Unpack( void* buf_recv, int buf_recv_size, int* pozycja, void* buf_dane, int count, MPI_Datatype typ, MPI_Comm comm )
```

Polecenia **Pack** i **Unpack** przeznaczone są do zapakowywania i odpakowywania tablic (do typu **MPI_PACKED**) które następnie można wysłać za pomocą np. **MPI_Send**.

Powyższe polecenia były przeznaczone do komunikacji pomiędzy dwoma procesami, zarządzania danymi lub inicjalizacji programu. Polecenia podane poniżej wykorzystywane są do komunikacji pomiędzy wszystkimi procesami (w grupie). Polecenia te są blokujące.

int MPI_Barrier(MPI_Comm comm)

bariera zaimplementowana przez MPI

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

Wysyła wiadomość z jednego procesu do wszystkich. Proces o randze **root** wysyła wartość zmiennej w **buff** do reszty procesów, które przyjmują tę wartość do swoich zmiennych, można przekazywać tablice za pomocą **count**.

int MPI_Gather(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm)

Zbiera elementy ze wszystkich procesów do tablicy dla procesu **root**.

int MPI_Allgather(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, MPI_Comm comm)

Analogicznie do **gather** ale dla wszystkich procesów.

int MPI_Scatter(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm)

Podobne do **Bcast** ale rozsyła tablice i rozsyła jej elementy zależnie od rangi.

int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

int MPI_AllReduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Wykonuje operację podaną za pomocą **op** na przekazanej zmiennej i wysyła wynik do **root** (lub do wszystkich procesów w przypadku **Allreduce**).

Typy operacji:

MPI_MAX – maksimum

MPI_MIN – minimum

MPI_SUM – suma

MPI_PROD – iloczyn

int MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *pop);

Wykorzystywane do definiowania operacji przez użytkownika za pomocą procedury.

Istnieją także procedury do komunikacji nieblokującej:

int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *req)

int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *req)

Odpowiedniki **send** i **recv** dla komunikacji nieblokującej. Zmienna **req** jest wykorzystywana do sprawdzania połączenia i oczekiwania.

int MPI_Wait(MPI_Request *preq, MPI_Status *pstat)

int MPI_Test(MPI_Request *preq, int *pflag, MPI_Status *pstat)

z odpowiednimi wariantami **any** – jakikolwiek proces, oraz **all** – wszystkie procesy.

int MPI_Probe(int src, int tag, MPI_Comm comm, MPI_Status *stat)

int MPI_Lprobe(int src, int tag, MPI_Comm comm, int* flag, MPI_Status *stat)

Testowanie otrzymania komunikatu, odpowiednio blokujące i nieblokujące.

Komunikacja nieblokująca może być wykorzystywana kiedy wykonywane są obliczenia niw wymagające danych odebranych(**irecv**) lub obliczenia danych nie przeznaczonych do wysyłania(**Isend**).

Typy komunikacji:

buforowany (MPI_Bsend, MPI_lbsend) – dane są przechowywane w buforze.

synchroniczny (MPI_Ssend, MPI_issend) – wysyłane jest potwierdzenia.

gotowości (MPI_Rsend, MPI_rsend) – System przygotowuje daną procedurę na odbiór/wysył danych.

Typy danych:

Można używać predefiniowanych typów (takich jak np. **MPI_INT**, **MPI_DOUBLE**) lub zdefiniować nowe typy za pomocą:

int MPI_Type_commit(MPI_Datatype *newtype)

dostarczanie typu

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Oba polecenia używane do definiowania typów, pierwszy używa ilości **count** zmiennych do utworzenia nowego typu. Drugi używa ilości **count** bloków zmiennych o długości **stride** do utworzenia nowego typu.

int MPI_Type_indexed(int count, int* tablica_długości_bloków, int*tablica_odstępów, MPI_Datatype oldtype, MPI_Datatype *newtype)

Podobne do poprzedniego polecenia, można zdefiniować długości i miejsce w pamięci za pomocą **tablica_długości_bloków** i **tablica_odstępów**.

```
int MPI_Type_create_struct(int count, int* tablica_długości_bloków, MPI_Aint* tablica_odstępów, MPI_Datatype* tablica_typów, MPI_Datatype *newtype)
```

Podobne do poprzedniego ale dodatkowo można zdefiniować pojedyncze typy zmiennych za pomocą **tablica_typów**.

Zmienne wykorzystywane przy tworzeniu nowych typów:

```
int MPI_Get_address( void* location, MPI_Aint* address )
```

Zwraca adresy zmiennych

```
int MPI_Type_get_extent( MPI_Datatype datatype, MPI_Aint* lb, MPI_Aint* extent )
```

Zwraca zasięg zmiennej danego typu

```
int MPI_Type_size( MPI_Datatype datatype, int* size )
```

Zwraca rozmiar zmiennej.

```
int MPI_Type_create_resized( MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype* newtype )
```

„rozszerzająca definicję typu o możliwe wyrównanie w pamięci”

laboratorium 11

Na laboratorium 11 zadaniem było zapoznanie się z podstawowymi funkcjonalnościami MPI, poprawne skompilowanie i uzupełnienie pliku. Celem ćwiczenia było stworzenie programu w którym proces **rank=0** odbierał by dane z tablicy od reszty procesów. W mojej implementacji programu wykorzystywane były 3 polecenia **send** i 3 polecenia **recv** (w pętli). Proces **rank=0** oczekiwał na wiadomość z rangą nadawcy odbierany od jakiegokolwiek procesu, następnie odbierał wyłącznie od tego procesu. Inne procesy wysyłały w kolejności: **rank**, rozmiar tablicy, i zawartość tablicy

Wysyłanie i odbieranie:

```
if( rank != 0 ){ dest=0; tag=0;
    MPI_Send( &rank, 1, MPI_INT, dest, tag, MPI_COMM_WORLD );
    MPI_Send( &h_size, 1, MPI_INT, dest, tag, MPI_COMM_WORLD );
    MPI_Send( (void*)&h_name, h_size, MPI_CHAR, dest, tag, MPI_COMM_WORLD );
} else {

    for( i=1; i<size; i++ ){
        MPI_Recv( &ranksent, 1, MPI_INT, MPI_ANY_SOURCE,
            MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        MPI_Recv( &h_size, 1, MPI_INT, ranksent,
            MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        MPI_Recv( &h_name, h_size, MPI_CHAR, ranksent,
            MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        printf("Dane od procesu o randze (status.MPI_SOURCE ->) %d: %d\n",
            (i==%d)\nHostname: %s\n",
            status.MPI_SOURCE, ranksent, i, h_name );
    }
}
```

laboratorium 12

To laboratorium polegało na podziale pracy pomiędzy procesy w celu obliczenia PI. Proces **rank=0** odbierał z konsoli liczbę grup wyrazów do obliczania przybliżenia PI (grupa = **size**), a następnie rozgłaszał tą liczbę do wszystkich procesów za pomocą **Bcast**. Po otrzymaniu tej wiadomości procesy wykonywały obliczenia zależne od ich rangi. Po wykonaniu obliczeń, wyniki były sumowane za pomocą **reduce** i **Allreduce** (wystarczyło by jedno z tych poleceń, wykorzystane były oba w celu ćwiczenia).

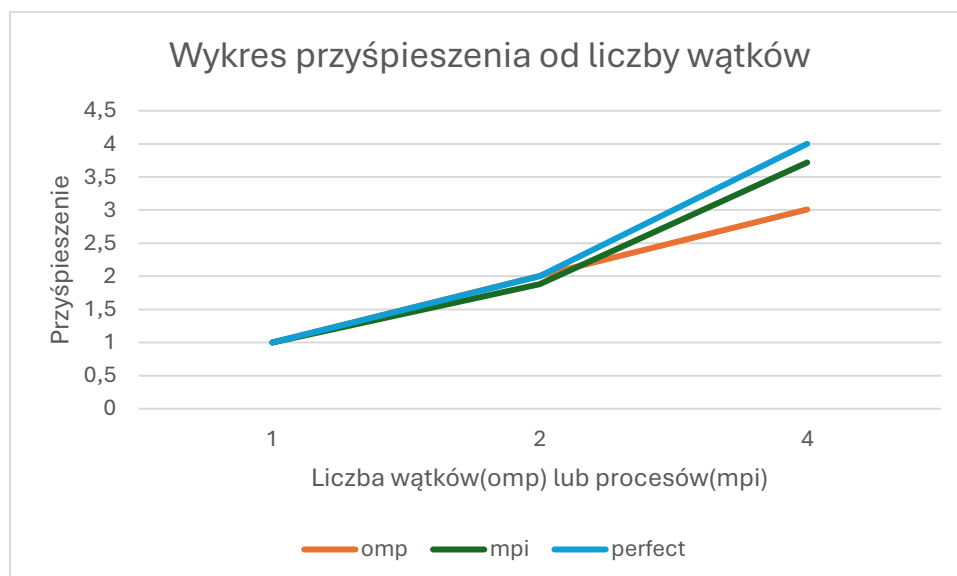
Ważna część kodu programu:

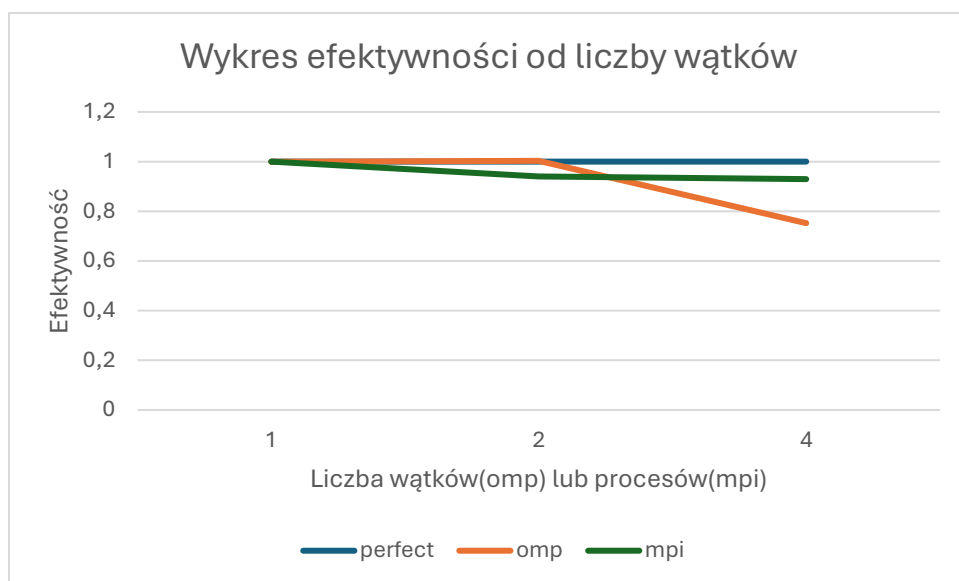
```
for(i=max_liczba_wyrazow*rank; i<max_liczba_wyrazow*rank +max_liczba_wyrazow; i++){
    int j = 1 + 4*i;
    suma_plus += 1.0/j;
    suma_minus -= 1.0/(j+2.0);
}
MPI_Reduce( &suma_plus, &sumap_glob, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
MPI_Reduce( &suma_minus, &sumam_glob, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
MPI_Allreduce( &suma_plus, &allsumap_glob, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
MPI_Allreduce( &suma_minus, &allsumam_glob, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
```

laboratorium 13

Celem tych ćwiczeń było zmierzenie czasu wykonania programów wykorzystujących **OMP** i **MPI**, oraz obliczenie przyspieszenia i efektywności. Kody programów były dostarczone i nie wymagały modyfikacji (poza ewentualnym ustaleniem liczby wątków w **OMP**).

Wyniki w formie wykresów:





Perfect reprezentuje teoretycznie perfekcyjne przyspieszenie i wydajność. Dla przyspieszenia perfect jest równy liczbie wątków/procesów, a dla efektywności jest zawsze równy jeden.

Przyspieszenie opisuje jak szybszy jest program przy coraz większym zrównolegleniu(), a efektywność wyraża jak dobrze wykorzystywane są zasoby przeznaczone do programów (w porównaniu do braku zrównoleglenia).

W tym przypadku **OMP** wykazało zaskakująco dobre parametry dla 2 wątków, może być to spowodowane nieprzewidzianym przyspieszeniem (lub brakiem spowolnienia) programu przy wykonywaniu pomiarów.