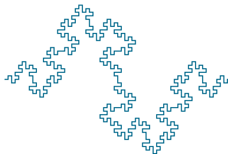
Programowanie współbieżne
Wykład 11
Wprowdzenie do MPI

Rafał Skinderowicz



Programowanie dla modelu z rozproszoną pamiecią

- Niezależne strumienie instrukcji wykonywane na (być może) oddalonych fizycznie komputerach
- Każdy proces ma odrębną przestrzeń adresową – brak bezpośredniego dostępu do zmiennych w pamięciach innych procesów
- Jedyna możliwość wymiany danych między procesami polega na przesyłaniu wiadomości (ang. message passing)
- Obliczenia asynchroniczne poza momentami komunikacji

Dlaczego model z rozproszoną pamięcią?

- Bariery technologiczne uniemożliwiają dalsze istotne zwiększanie taktowania procesorów
- Prawo Moore'a obowiązuje nadal dzięki zwiększaniu liczby rdzeni
- Model ze współdzieloną pamięcią sprawdza się (skaluje się) jedynie dla ograniczonej liczby procesorów (rzędu kilkuset) ze względu na koszt obliczeniowy i opóźnienia związane z protokołami służącymi do zachowania spójnego, globalnego stanu pamięci dzielonej przez procesory
- Jedyna możliwość wykorzystania pełnej mocy komputerów równoległych, to tworzenie oprogramowania w modelu z pamięcią rozproszoną
  - Obecnie najszybszy superkomputer, K-computer, ma 705024 ($\sim$ 700 tyś.) rdzeni obliczeniowych

Dlaczego model z rozproszoną pamięcią?

Autor: H. Stutter, http://www.gotw.ca/publications/concurrency-ddj.htm

Wstęp do MPI

- MPI jest standardem przesyłania komunikatów między procesami w programach równoległych dla modelu z rozproszoną pamięcią.
- MPI jest obecnie dominującym sposobem tworzenia aplikacji równoległych na potrzeby HPC (ang. high performance computing)
- Piersza wersja (MPI-1) powstała w 1994 r. Obecnie najnowsza wersja to MPI-2 (1997 r.)
- MPI zawiera specyfikację funkcji bibliotecznych (API), które muszą zostać dostarczone przez konkretną implementację biblioteki
- Dostępne wersje dla C/C++, Fortran, Ada, ale również pośrednio przez możliwość uruchamiania natywnego kodu dla innych języków, np. Java, Python

MPI – zalety

- Umożliwia tworzenie wydajnych programów równoległych korzystających jednocześnie z tysięcy procesorów
- Można określić logiczną strukturę połączeń między węzłami, np. pierścień, siatka 3D
- Ustandaryzowane – dostępnych kilka implementacji, m.in. MPICH, OpenMPI – przenośność
- Komunikacja zarówno synchroniczna, jak i asynchroniczna
- Bogata lista funkcji (ponad 500)
- Dobra dokumentacja, książki, kursy, strony internetowe itp.

MPI – WADY

- Stosunkowo niskopoziomowa, brak automatycznego zrównoleglania, np. pętli
- Wymagana konfiguracja komputerów (węzłów) wchodzących w skład klastra zależna od konkretnej implementacji
- Statyczna konfiguracja jednostek przetwarzających
- Brak obsługi wielowątkowości w standardzie – można korzystać z wielowątkowości dzięki dodatkowym bibliotekom

Ciekawostka

Udział procentowy systemów operacyjnych w 500 najszybszych superkomputerach na świecie, stan na listopad 2011 r.

Operating System	Count	System Share (%)	Cores
Linux	414	82.8	7013833
AIX	28	5.6	158288
CNK/SLES 9	11	2.2	1093632
SLES10 + SGI ProPack 5	9	1.8	105856
CNL	6	1.2	102050
CentOS	5	1	95264
Cray Linux Environment	4	0.8	287040
RHEL 6.2	4	0.8	82832
bullx SUperCOmputer Suite A.E.2.1	3	0.6	46720
Redhat Linux	3	0.6	42144
RedHat Enterprise 5	2	0.4	17088
SUSE Linux Enterprise Server 10	2	0.4	18944
SUSE Linux	2	0.4	28864
SUSE Linux Enterprise Server 11	2	0.4	13728
RHEL 6.1	1	0.2	37056
Windows HPC 2008	1	0.2	30720
Super-UX	1	0.2	1280
Open Solaris	1	0.2	12032
Cell OS	1	0.2	5472

MPI – STRUKTURA PROGRAMU

Plik nagłówkowy
`#include <mpi.h>`

Deklaracje zmiennych globalnych, funkcji itp.

Część sekwencyjna

Inicjalizacja MPI

Część równoległa

```
int moj_id = MPI_Comm_rank(...);

if (moj_id == 0) {
     funkcja1();
} else {
     funkcja2();
}

obliczenia(porcja(moj_id));
```

Zamknięcie MPI

Część sekwencyjna

MPI – pierwszy program

```
#include "mpi.h"
#include <stdio.h>
int main( intargc, char * argv[] )
{
   MPI_Init( &argc, &argv);
   printf("Hello, world!");
   MPI_Finalize();
   return 0;
}
```

MPI podstawy

- Wszystkie funkcje biblioteczne dostępne są przez nagłówek mpi.h
- Konwencja nazewnicza dla języka C:
  `MPI_Xxxxx(parametr1, parametr2, ...)`
- Wszystkie funkcje MPI zwracają kody błędów lub MPI_SUCCESS jeżeli wykonanie zakończyło się pomyślnie
- Wysyłanie i odbieranie komunikatów wymaga podania wskaźników do buforów z danymi do wysłania / odebrania

MPI – kompilacja i uruchamianie

- Kompilacja zależy od stosowanej implementacji, np. w języku C dla implementacji MPICH: `mpicc program.c -o program.exe` lub mpicxx dla C++ – skrypty te uruchamiają kompilator, za pomocą którego skompilowano bibliotekę MPI, np. GNU gcc
- Sposób uruchomienia programu zależny jest od stosowanej implementacji, w przypadku uruchamiania na lokalnym komputrze może to wyglądać:

```
// dla 16 procesorów
mpiexec -n 16 mojprogram.exe parametr1 parametr2
```

- Zazwyczaj środowisko wykonania konfigurowane jest przed uruchomieniem programu – statycznie

MPI_Init oraz MPI_Finalize

- Przed użyciem jakiejkolwiek metody z biblioteki MPI należy ją zainicjalizować za pomocą metody `MPI_Init` – należy do niej przekazać niezmodyfikowane parametry otrzymane w funkcji `main`
- Skrypt `mpirun` lub `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 stosowaną do buforowania przesyłanych komunikatów
- Przed wywołaniem `MPI_Finalize` należy zadbać o to, aby wszystkie wysłane komunikaty zostały odebrane

Pobieranie informacji o środowisku uruchomieniowym

- Dwie podstawowe kwestie dotyczące środowiska (komputera równoległego), w którym wykonywany jest program to:
  - Liczba procesów wykonujących program
  - Identyfikator bieżącego procesu – od 0 do $n-1$, gdzie n oznacza liczbę procesów
- MPI udostępnia funkcje odpowiadające na podane kwestie:
  - `MPI_Comm_size` zwraca liczbę procesów
  - `MPI_Comm_rank` zwraca rangę procesu, czyli jego identyfikator

MPI – DRUGI PROGRAM

```
#include "mpi.h"
#include <stdio.h>
int main( intargc, char * argv[] )
{
   MPI_Init( &argc, &argv);
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   printf( "Jestem %d. procesem z %d", rank, size );
   MPI_Finalize();
   return 0;
}
```

Przykładowy wynik dla 4 procesów:

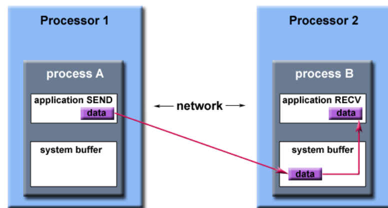
```
Jestem 0. procesem z 4
Jestem 2. procesem z 4
Jestem 3. procesem z 4
Jestem 1. procesem z 4
```

MPI – komunikacja

- Wymiana komunikatów (wiadomości) między procesami wymaga ich kooperacji
- Dane są jawnie wysyłane przez proces-nadawcę i jawnie odbierane przez proces-odbiorcę, jawnie, czyli przez wykonanie określonej funkcji MPI
- Nadawca wywołuje funkcję A, np. MPI_Send, a odbiorca wywołuje odpowiadającą jej funkcję B, np. MPI_Recv
- Komunikacja może być:
  - synchroniczna – proces jest wstrzymywany do momentu zakończenia danej operacji komunikacji
  - asynchroniczna – odbiorca i nadawca wykonują odpowiednie metody komunikacyjne w różnych momentach – tylko inicjacja transferu i powrót
- W specyfikacji MPI używane są pojęcia komunikacji blokującej (ang. blocking) i nieblokującej (ang. non-blocking)
- Nieodpowiednia kolejność wykonywania operacji komunikacji może powodować zakleszczenie programu

Buforowanie

- Wywołanie operacji wysłania wiadomości rzadko zbiega się idealnie w czasie z wywołaniem operacji odbioru tej wiadomości, stąd konieczność buforowania
- Implementacje MPI przechowują odebrane wiadomości w buforze systemowym (niejawnym)
- Buforowanie stosowane jest, aby poprawić wydajność programu



Path of a message buffered at the receiving process

Źródło: https://computing.llnl.gov/tutorials/mpi/

Komunikacja blokująca i nieblokująca

- Większość operacji komunikacji między parą procesów występuje w dwóch odmianach blokującej i nieblokującej
- Operacje blokujące:
  - blokująca operacja zakończy się powodzeniem, wtedy gdy bezpiecznie można zmodyfikować bufor, w którym mieściła się nadawana wiadomość
  - niekoniecznie oznacza, to że wiadomość została odebrana przez odbiorcę, być może została ona tylko umieszczona w buforze systemowym
- Operacje nieblokujące:
  - zwracają wartość niemal natychmiast bez czekania na zakończenie komunikacji
  - o tym kiedy operacja zostanie rzeczywiście wykonana decyduje dana implementacja MPI
  - dopóki operacja blokująca nie zostanie zakończona, nie można używać buforu z danymi nadawanymi lub odbieranymi – istnieją odpowiednie metody "Wait" do sprawdzania statusu operacji

Kolejność i uczciwość w operacjach komunikacji

- Kolejność
  - MPI gwarantuje, że wiadomości nie będą się „nakładać” w trakcie przesyłania
  - Jeżeli proces nadawca wysłał do procesu odbiorcy wiadomości w określonej kolejności, to w takiej również zostaną odebrane
  - Jeżeli więcej, niż jeden wątek uczestniczy w komunikacji, to nie ma gwarancji zachowania kolejności operacji komunikacji
- Uczciwość
  - MPI nie gwarantuje uczciwości operacji komunikacji
  - Przykładowo, jeżeli proces 1. wysyłał wiadomość do procesu 3. a w tym samym czasie proces 2. wysłał analogiczną wiadomość do procesu 3., to tylko jedna z tych operacji zostanie zakończona pomyślnie, nie ma gwarancji, że będzie to operacja procesu o najniższym identyfikatorze
  - Problemu nie ma, gdy odbiorca odbiera wszystkie wysłane do niego wiadomości lub wie ile wiadomości powinien otrzymać od poszczególnych nadawców

Operacja 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
- 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`

Operacja MPI_Recv

Składnia operacji odebrania wiadomości jest następująca:

`MPI_Recv(buffer, count, datatype, dest, 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

MPI – typy danych

- W celu zapewnienia przenośności między różnymi architekturami komputerów i systemami operacyjnymi standard MPI definiuje typy danych, które są używane w operacjach dotyczących MPI
- Dodatkowo dostępne są metody do definiowania własnych, złożonych typów danych, takich jak wektory, czy struktury

MPI – TYPY DANYCH

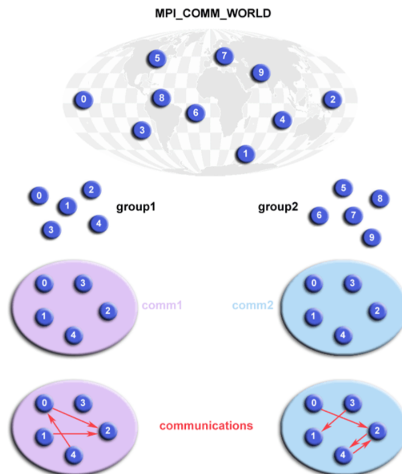
Nazwa	Odpowiednik j. C/C++
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE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

Identyfikatory (rangi)

- Identyfikatory nadawcy oraz odbiorcy, to dane typu `MPI_INT` z zakresu od 0 do $n-1$, gdzie n oznacza rozmiar komunikatora, do którego należą procesy nadawcy i odbiorcy
- Dla operacji `MPI_Recv` można podać stałą `MPI_ANY_SOURCE` w miejscu identyfikatora nadawcy, co umożliwia odebranie wiadomości od dowolnego nadawcy – rzeczywisty identyfikator nadawcy dostępny jest przez parametr status

Grupy i komunikatory

- MPI umożliwia grupowanie procesów w uporządkowane zbiory, w których każdy proces ma unikalny identyfikator
- Dostęp do grupy odbywa się przez jej „uchwyt” udostępniany przez MPI
- Komunikator obejmuje grupę procesów, które mogą się komunikować między sobą
- Z punktu widzenia programisty komunikatory i grupy są tożsame
- Domyślnie wszystkie procesy należą do komunikatora globalnego: `MPI_COMM_WORLD`



Źródło: https://computing.llnl.gov/tutorials/mpi

Grupy i komunikatory c.d.

Cele tworzenia komunikatorów:

- Umożliwiają grupowanie procesów w jednostki zgodnie z ich funkcją w programie równoległym
- Umożliwiają tworzenie wirtualnych topologii
- Mogą być tworzone i usuwane dynamicznie w trakcie działania programu
- Umożliwia ograniczenie komunikacji tylko do wybranej grupy procesów, które powinny się ze sobą komunikować – prosty mechanizm bezpieczeństwa

Znaczniki

- Dla każdej wiadomości należy określić znacznik identyfikujący jej kontekst – dwie identyczne pod względem struktury binarnej wiadomości mogą mieć różne znaczenie
  - W programie można łatwo zdefiniować własne znaczniki dla przesyłanych wiadomości
  - Znacznik to po prostu liczba całkowita bez znaku
  - Dostępne są znaczniki oznaczające dowolną wiadomość (ang. wild-cards)
- W operacji `MPI_Recv` można podać stałą `MPI_ANY_TAG`, co umożliwia odebranie wiadomości o dowolnej wartości znacznika

Przykład

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

int main(int argc, char *argv[]) {
int rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) { // proces 0. najpierw wysyła
   dest = 1;
   source = 1;
   rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}
else if (rank == 1) { // proces 1. najpierw odbiera
   dest = 0;
   source = 0;
   rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
   rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
// wyświetlenie informacji o odebranym komunikacie
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d",
      rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

Status operacji MPI_Recv

- 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_Status;
```

- MPI_SOURCE to ranga nadawcy wiadomości
- MPI_TAG to znacznik wiadomości
- MPI_ERROR to kod ewentualnego błędu
- Liczbę odebranych jednostek danych można sprawdzić za pomocą: `int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int count)`

Niebezpieczeństwo zakleszczeń

- Użycie blokującej komunikacji może prowadzić do zakleszczeń
- Przykład:

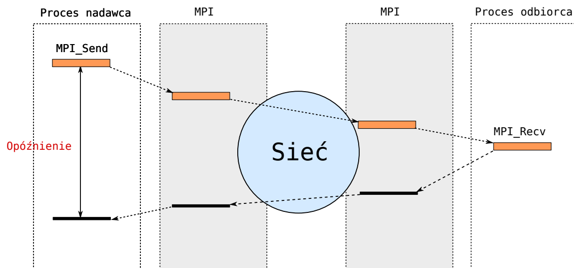
```
int tag1 = 1;
int tag2 = 2;

if (rank == 0) {
   dest = 1;
   rc = MPI_Send(&a, 1, MPI_CHAR, dest, tag1, MPI_COMM_WORLD);
   rc = MPI_Send(&b, 1, MPI_CHAR, dest, tag2, MPI_COMM_WORLD);
}
else if (rank == 1) {
   source = 0;
   rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag2,
                 MPI_COMM_WORLD, &Stat);
   rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag1,
                 MPI_COMM_WORLD, &Stat);
}
```

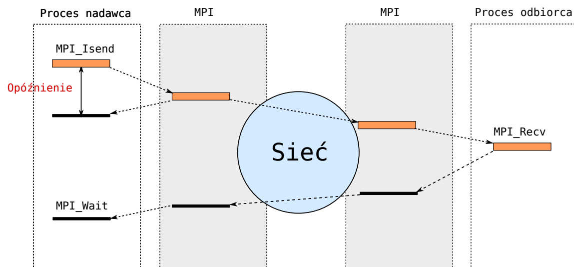
- Kilka możliwych rozwiązań, najprostsze – odpowiednia kolejność operacji komunikacji

Asynchroniczna komunikacja

- MPI definiuje zestaw funkcji do asynchronicznego (nieblokującego) przesyłania komunikatów
- Nazewnictwo: `MPI_Ixxxx` zamiast `MPI_xxxx`
- Wykonanie funkcji nieblokującej kończy się niemal natychmiast, więc proces może kontynuować obliczenia bez oczekiwania na jej zakończenie
- Przesyłanie i odbieranie komunikatów realizowane jest „w tle" – nie można modyfikować zawartości buforów dopóki nie będzie pewności, że operacja zakończyła się poprawnie

Asynchroniczna komunikacja





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 analogiczną rolę, jak w operacji `MPI_Isend`
- Do czasu rzeczywistego zakończenia przekazywania wiadomości nie wolno modyfikować zawartości bufora wskazywanego przez `buffer`

Asynchroniczna wymiana komunikatów – przykład

```
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 można zwolnić pamięć dla bufora
} else if (rank == 1) { // proc. 1. odbiera tablicę
   int *buf = new int[N];
   MPI_Recv(buf, N, MPI_INT, 0, TagLiczby, MPI_COMM_WORLD, &status);
   ...
   delete [] buf;
}
```

Asynchroniczna wymiana komunikatów – przykład 2.

```
int numtasks, rank, next, prev, buf[2];
const int tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[2];

MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// komunikacja wg topologii pierścienia
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
obliczenia();
MPI_Waitall(4, reqs, stats); // czekaj na zakończenie wszystkich
                             // operacji
```

Komunikacja asynchroniczna – pułapki

- Nie można modyfikować bufora z danymi po wywołaniu `MPI_ISend`, a przed wywołaniem odpowiadającego jej `MPI_Wait`
- Nie można czytać, ani modyfikować danych po wywołaniu `MPI_Recv`, a przed wywołaniem `MPI_Wait`
- Nie można mieć rozpoczętych dwóch operacji `MPI_Irecv` z tym samym buforze

Mniej oczywiste

- Nie można czytać danych po wywołaniu `MPI_Isend` a przed wywołaniem `MPI_Wait`
- Nie można mieć dwóch rozpoczętych operacji `MPI_Isend` z tego samego bufora

MPI_Probe oraz MPI_Iprobe

MPI udostępnia funkcje umożliwiające sprawdzenie, czy gotowy jest do odebrania komunikat

- `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - wersja *blokująca* – oczekuje na nadesłanie komunikatu o znaczniku `tag` od procesu `source`, ale go nie odczytuje
  - umieszcza dane o gotowym do odebrania komunikacie w zmiennej `status`
- `int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)`
  - wersja *nieblokująca* – sprawdza, czy został nadesłany komunikat i od razu kończy działanie
  - jeżeli komunikat jest dostępny, to zmienna `flag` ustawiana jest na wartość 1, a w zmiennej `status` umieszczane są informacje o komunikacie
- Funkcje przydatne, gdy potrzeba odebrać komunikat, ale nie jest znany rozmiar bufora dla odbieranych danych

MPI – pomiar czasu

- MPI udostępnia funkcję `MPI_Wtime`, która zwraca czas w sekundach, który upłynął od pewnego ustalonego momentu w przeszłości
- Rozdzielczość zegara zwraca funkcja `MPI_Wtick`
- Zegary poszczególnych procesów nie są zsynchronizowane, chyba że stała `MPI_WTIME_IS_GLOBAL` ma wartość 1

Przykład

```
double start = MPI_Wtime();
...
wykonaj obliczenia
...
double koniec = MPI_Wtime();
double czas_obliczen = koniec - start;
printf("Czas obliczeń: %.1lf sekund", czas_obliczen);
```

Podsumowanie

- Mimo dużej liczby funkcji oferowanej przez MPI do implementacji w pełni funkcjonalnego programu wystarczy zaledwie kilka:
  - `MPI_Init` – inicjalizacja
  - `MPI_Comm_size` – pobranie liczby procesów
  - `MPI_Comm_rank` – pobranie identyfikatora procesu
  - `MPI_Send` – wysłanie komunikatu
  - `MPI_Recv` – odebranie komunikatu
  - `MPI_Finalize` – zakończenie
- Wszystkie procesy należą do komunikatora `MPI_COMM_WORLD`
- Należy pamiętać o wykonywaniu operacji komunikacji w takiej kolejności, aby nie doszło do zakleszczenia
- Wiele z zaawansowanych funkcji MPI ma na celu optymalizację i należy się do nich odwoływać dopiero, gdy program działa poprawnie