Programowanie w standardzie MPI

Podstawy programowania z przesyłaniem komunikatów

- Model systemu równoległego w postaci p procesów, każdy z nich z własną przestrzenią adresową, nie współdzieloną z innymi procesami. Przestrzeń danych podzielona na p partycji.
  - Każdy element danych musi należeć do jednej z partycji. Dane muszą być jawnie podzielone na p procesów.
  - Wszelkie interakcje (odczyt bądź odczyt-zapis) wymagają jawnej współpracy dwóch procesów: procesu posiadającego dane i procesu chcącego uzyskać dostęp do danych.
- Te dwa ograniczenia utrudniają programowanie. Z drugiej strony sprawiają, że koszty komunikacji są wyjątkowo wyraźnie widoczne dla programisty.

Historia

- Zanim powstał MPI inna biblioteka dla każdego typu komputera (np. CMMD, MPL, NX)
- PVM (parallel virtual machine) – próba standardu; niezbyt udana (niska wydajność, nie najlepiej wyspecyfikowane)
- MPI stworzone przez MPI Forum, organizację reprezentującą przemysł, uczelnie, laboratoria rządowe.
- MPI-1 (1994) kodyfikacja istniejących praktyk w zakresie przesyłania komunikatów.
- MPI-2 (1997) projekt badawczy, nowe elementy.
  - Jednostronna komunikacja.
  - Równoległe wejście – wyjście.
  - Interfejsy do C++ i Fortranu 90.
  - Dynamiczne tworzenie procesów.
- **MPI odniosło sukces i stało się standardem**
  - Należy jednak pamiętać że standard staje się barierą dalszego rozwoju.

MPI: Message Passing Interface

- MPI jest definicją zbioru funkcji bibliotecznych dla C, C++ i Fortranu (bardzo ważny język w obliczeniach naukowo-inżynierskich) pozwalających na pisanie przenośnych programów realizujących model z przekazywaniem komunikatów.
  - ***Specyfikacja interfejsu, nie implementacja !!!***
  - Pozwala na pisanie przenośnych programów.
- Specyfikacja MPI definiuje składnię i semantykę operacji komunikacyjnych.
- Sprzedawcy specjalizowanych komputerów równoległych dostarczają własne implementacje MPI.
- Dodatkowo istnieją darmowe implementacje (MPICH, LAM, OpenMPI) przeznaczone dla systemów połączonych standardowymi sieciami połączeń (TCP/IP over Gigabit Ethernet) jak i bardziej egzotycznymi (Myrinet, Infiniband).
- W pełni funkcjonalny program może być napisany z wykorzystaniem sześciu funkcji MPI.

Model SPMD (ang. single program, multiple data)

- MPI wykorzystuje model programowania SPMD, w którym na każdym procesorze wykonuje się ten sam program.
- O.K., skoro każdy procesor wykonuje ten sam program, nie mamy pożytku z równoległości -> wszystkie procesory otrzymają takie same wyniki.
- Na szczęście każdy proces otrzymuje swój unikalny identyfikator w komunikatorze, instrukcje kontrolne np. If pozwalają na wykonania różnych czynności przez różne procesy.
- Funkcja `MPI_Comm_size` zwraca liczbę procesów w komunikatorze.
- Funkcja `MPI_Comm_rank` zwraca numer – rangę (ang. Rank) procesu w komunikatorze.

Kompilacja i uruchamianie programów

- Dla programów w C oraz C++ należy zastosować dyrektywę

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

- Programy w C kompilujemy i linkujemy skryptem mpicc, a programy w C++ skryptem mpiCC (lub mpicxx). Skrypty te wywołują kompilator, za pomocą którego została skompilowana biblioteka MPI (np. Gcc).
- Program uruchamiamy skryptem mpirun, podając po opcji -np liczbę procesów. Na przykład:

```
mpirun -np 4 myprog arg1 arg2
```

uruchomi 4 procesy z programu myprog, przekazując każdemu procesowi argumenty wiersza poleceń arg1 oraz arg2.

- Liczba procesów jest ustalana na stałe przy uruchomieniu i jest stała (w MPI 2.0 to zmieniono).
- Przypisanie procesów procesorom jest pozostawione implementacji. Oczywiście jeżeli biblioteka MPI została zainstalowana na pojedynczej jednoprocesorowej stacji roboczej, wszystkie procesy zostaną uruchomione na jednym procesorze.

MPI_Init oraz MPI_Finalize

- Podsystem MPI jest inicjalizowany przy pomocy funkcji `MPI_Init`. Należy jej przekazać niezmienione argumenty `argc` oraz `argv` funkcji `main` !!!. Swoje argumenty można przetwarzać dopiero po wywołaniu `MPI_Init`.
  - skrypt mpirun przekazuje programowi dodatkowe argumenty, które są wykorzystywane przez `MPI_Init`
- Przed wywołaniem `MPI_Init` nie wolno wywoływać żadnych funkcji MPI.
- Przed zakończeniem pracy MPI należy wywołać funkcję `MPI_Finalize`. Kończy ona pracę podsystemu MPI.
- Po wywołaniu `MPI_Finalize` nie wolno wywoływać żadnych funkcji MPI.
- Uwaga, przed wywołaniem należy zadbać, aby każdy komunikat wysłany przez operację send został odebrany przez operację receive. ***Niedopuszczalne jest pozostawienie „osieroconych" komunikatów, tzn. wysłanych a nie odebranych !!!***

Kody powrotu z funkcji MPI

- Wszystkie funkcje MPI (za wyjątkiem `MPI_Wtime` oraz `MPI_Wtick` służących do pomiaru czasu) zwracają kod powrotu informujący o wyniku operacji.
- Wartość `MPI_SUCCESS` oznacza operację zakończoną pomyślnie, każda inna wartość oznacza błąd.
- Przy pomocy funkcji `MPI_Error_string(int errorcode,char *string, int stringlen)` możemy przekształcić kod błędu na jego angielski opis tekstowy.
  - `stringlen` jest długością bufora przekazanego przez `string`.

Komunikatory MPI

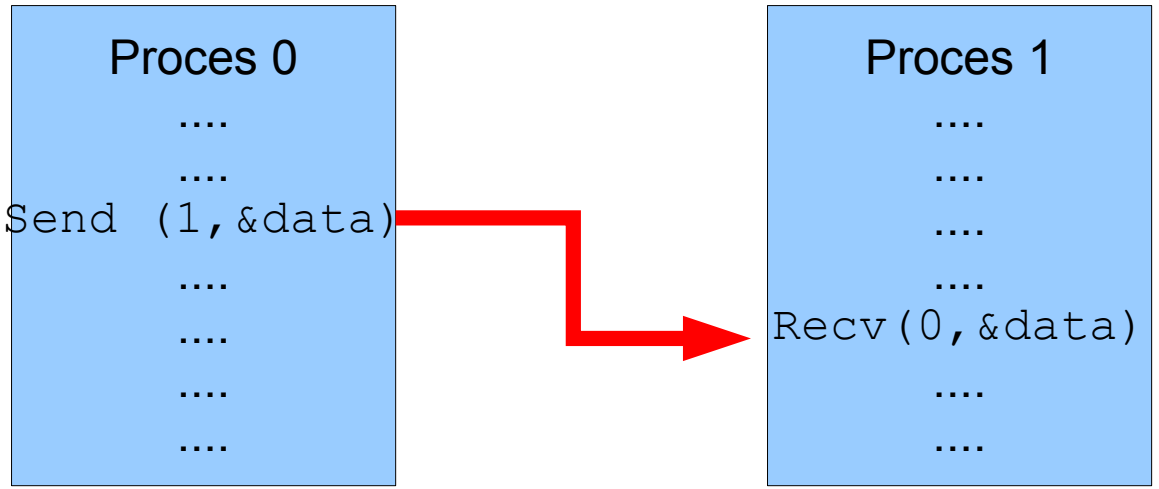
- Komunikator określa zbiór procesów mogących się komunikować ze sobą.
- Komunikator jest zdefiniowany typem MPI_Comm.
- Wszystkie funkcje przesyłające komunikaty wymagają podania komunikatora.
- Każdy proces będący członkiem komunikatora ma swój numer w tym komunikatorze.
- Proces może być członkiem wielu (potencjalnie nakładających się na siebie) komunikatorów.
- Przy starcie zdefiniowany jest komunikator MPI_COMM_WORLD, którego członkiem są wszystkie procesy.
- Motywacja: budowa bibliotek równoległych

Pierwszy program w MPI - Hello World

```
#include <mpi.h>

int main(int argc, char *argv[])
{
   int npes;
   int myrank;

   // Inicjalizacja podsystemu MPI
   MPI_Init(&argc, &argv);

   // Pobierz rozmiar globalnego komunikatora
   MPI_Comm_size(MPI_COMM_WORLD, &npes);

   // Pobierz numer procesu w globalnym komunikatorze
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   printf("Jestem %d procesem z %d\n",myrank, npes);
   MPI_Finalize();
   return 0;
}
```

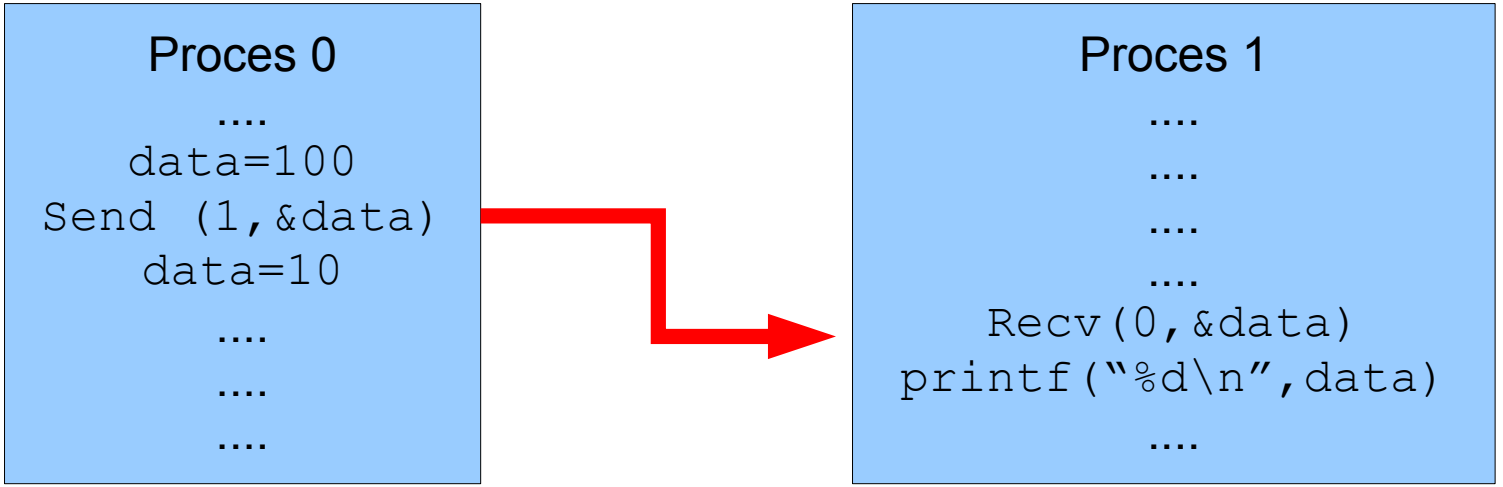
- Użyteczniejszy program podejmowałby różne akcje w zależności od numeru procesu.

Para Send-Receive



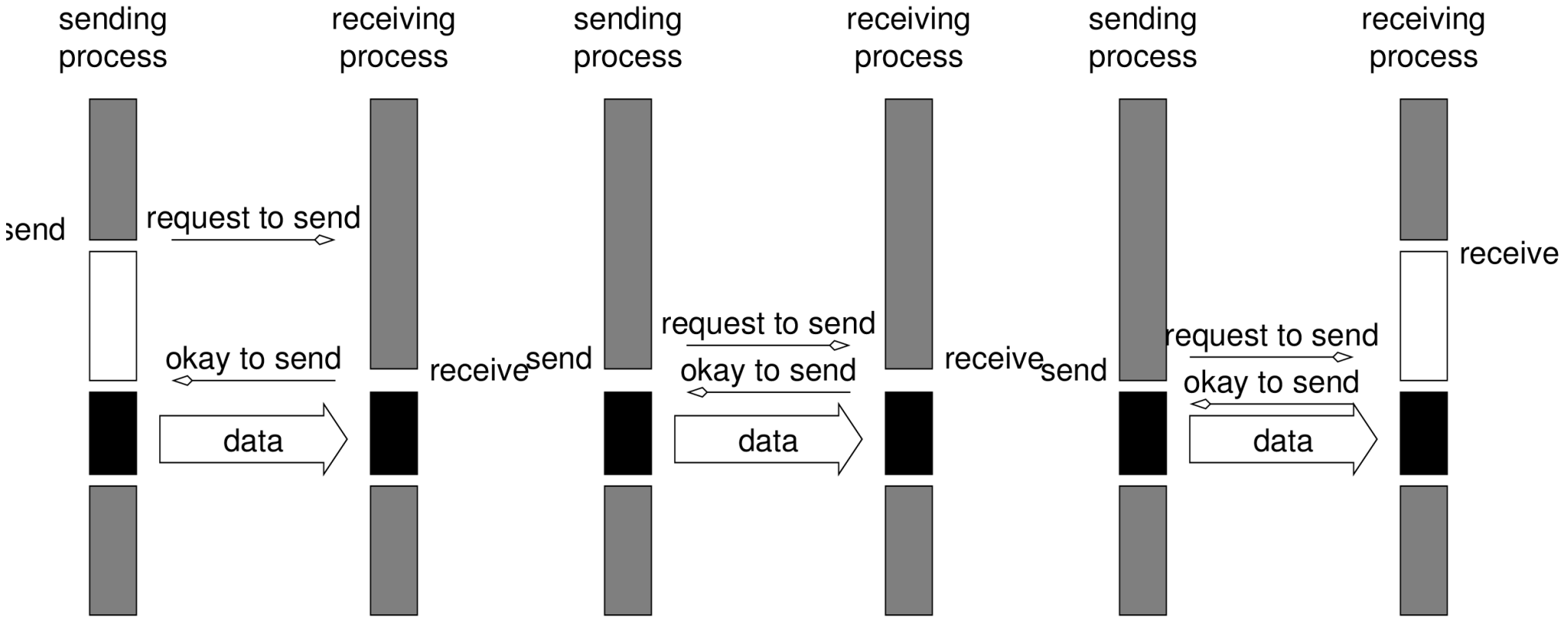
- Potrzebna współpraca nadawcy i odbiorcy.
- Należy określić czy operacje Send/Receive są synchroniczne czy asynchroniczne.
  - Synchroniczna wstrzymuje proces do momentu transferu
  - Asynchroniczna tylko inicjuje transfer i powraca.
- Specyfikacja MPI używa innych pojęć: ***blokujące/nieblokujące***

Semantyka pary operacji blokujących Send/receive



- Musimy mieć pewność, że printf u odbiorcy wypisze 100 (a nie np.. 10 !).
- W przypadku nowoczesnego sprzętu (asynchroniczne transfery przez sieć, DMA) nie jest to takie proste.
- W operacjach blokujących
  - Send powraca dopiero wtedy gdy jest to bezpieczne z punktu widzenia semantyki operacji.
  - Receive powraca po odebraniu danych
- Może (lub nie) być wykorzystane buforowanie

Blokujące send/receive bez buforowania (Grama i wsp., 2003)



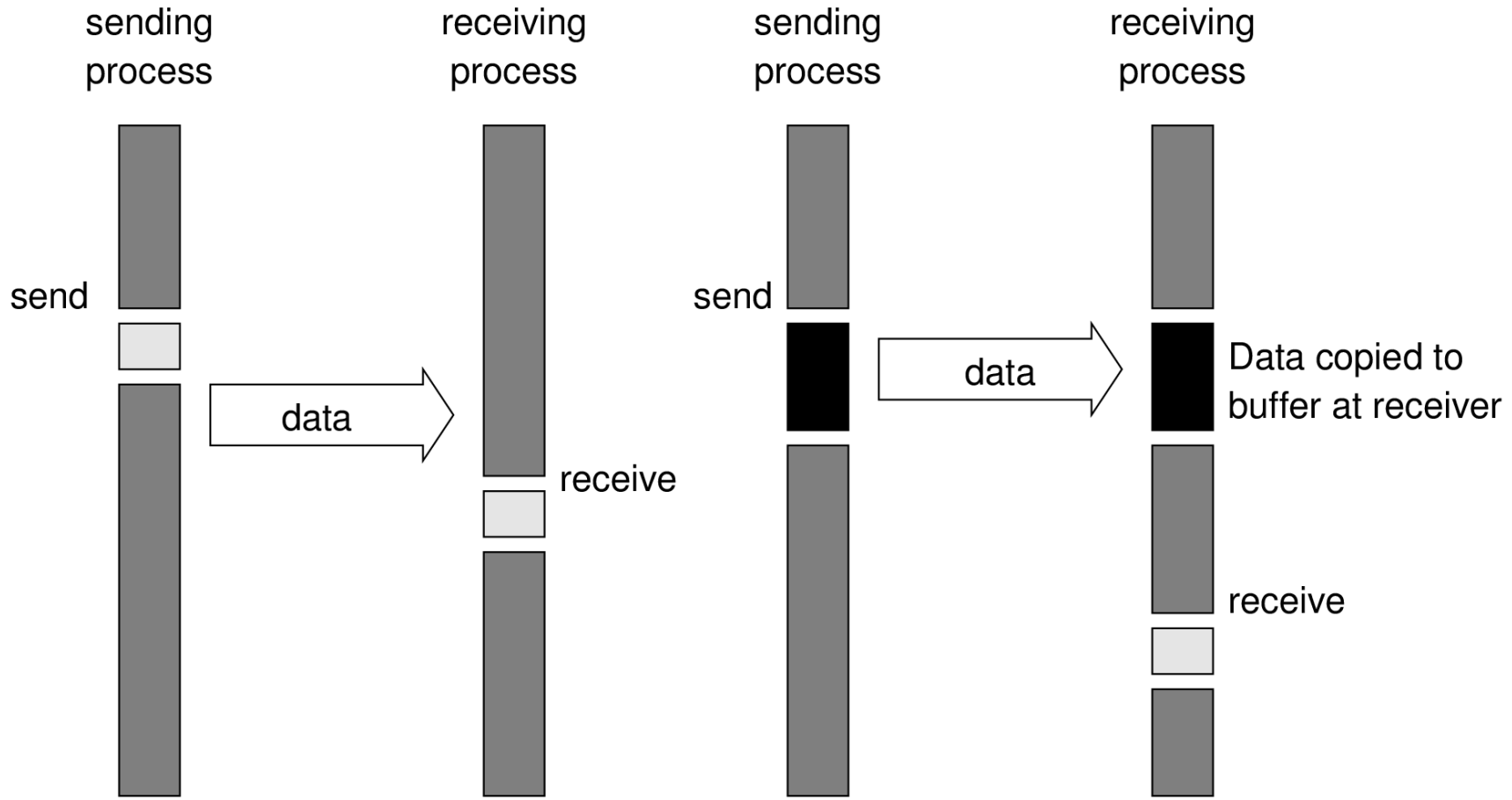
(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

- Protokół z uściskiem dłoni (ang. handshake).
- Niepoprawny program może prowadzić do zakleszczeń (ang. deadlock, o tym później).
- Okresy bezczynności (ang. idling) – buforowanie może je wyeliminować

Blokujące send/receicve z buforowaniem



Buforowanie u nadawcy i odbiorcy
(w przypadku specjalizowanego sprzętu)

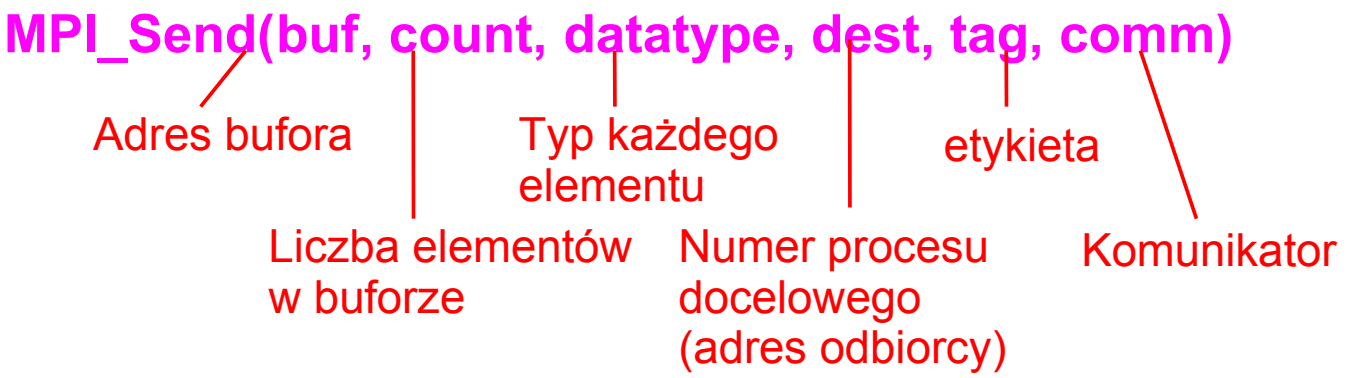
Buforowanie wyłącznie u odbiorcy)

- Czy buforowanie wyeliminuje wszystkie przypadki zakleszczeń ?
  - Bufory mają skończoną pojemność
  - Receive zawsze może wstrzymać proces

Blokujące operacje Send i Receive w MPI

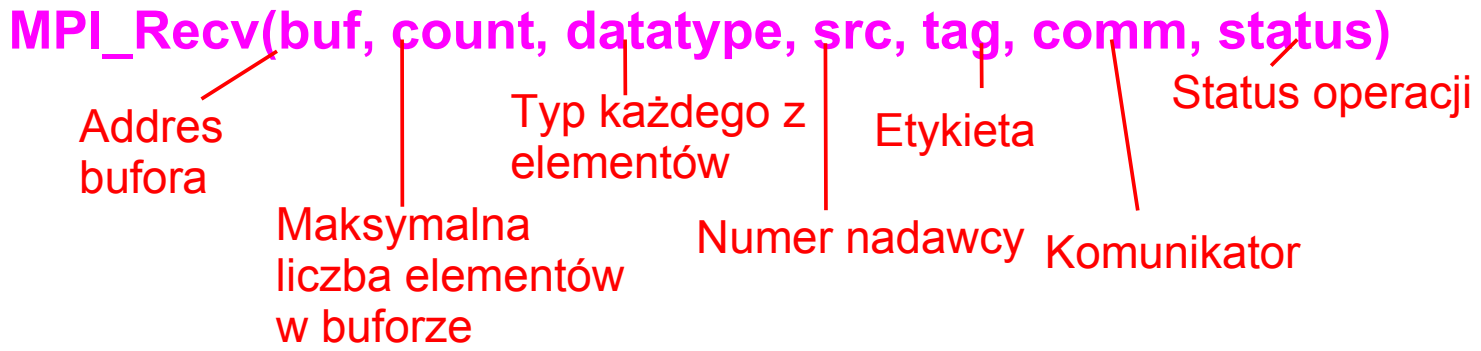
- Wszystkie operacje MPI operują na wektorach danych
- Operacja Send wstrzymuje proces do momentu gdy zostanie „lokalnie skompletowana” (cytat za specyfikacją MPI).
- Powrót z funkcji Send oznacza, że wektor danych możemy ponownie wykorzystać.
- Powrót z operacji Receive oznacza że dane zostały odebrane i są dostępne w wektorze.
- Standard MPI pozwala na swobodę jeżeli chodzi o powrót z operacji Send. Może on nastąpić np.
  - Dopiero po odbiorze danych przez odbiorcę – czekamy na wywołanie Receive przez odbiorcę.
  - Już po skopiowaniu do wewnętrznego bufora (np. w pamięci karty sieciowej) – nie czekamy na Receive.
  - Motywacją do takiej decyzyji jest chęć zapewnienia największej wydajności.
  - Ale może to prowadzić ***do trudnych do wykrycia przypadków blokad (zakleszczeń)***.

Funkcja MPI_Send



- Etykieta – każdy wychodzący komunikat opatrzony jest etykietą będącą liczbą całkowitą. Etykietę wybiera programista.

Funkcja MPI_Recv



- Dopuszczalna jest w sytuacja, w której odbierzemy mniej elementów niż zadeklarowany rozmiar bufora.
- Funkcja czeka na komunikat od nadawcy src o etykiecie tag. (Nie zareaguje na komunikat od innych nadawców lub o innej etykiecie).
- Jako src można użyć MPI_ANY_SOURCE (odbierz komunikat od dowolnego nadawcy), a jako tag MPI_ANY_TAG (o dowolnej etykiecie)

Typy danych MPI

MPI Datatype	C Datatype
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	
MPI_PACKED	

- MPI_BYTE możemy wykorzystać (wraz z operatorem `sizeof`), jeżeli chcemy przesłać typ zdefiniowany przez programistę (np. strukturę składającą się z różnych pól)

Przesłanie zmiennej x z procesu 0 do 1

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0) {
   int x;
   MPI_Send(&x, 1, MPI_INT, 1, 13, MPI_COMM_WORLD);
} else if (myrank == 1) {
   int x;
   MPI_Recv(&x, 1, MPI_INT, 0, 13,MPI_COMM_WORLD,status);
}
```

- Przesyłamy wektor liczb typu int (`MPI_INT`) o rozmiarze jeden
- Jako etykietę wybrano 13.

Status operacji `MPI_recv`

- W języku C/C++ definiowany jako:

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

- `MPI_SOURCE` określa nadawcę operacji
- `MPI_TAG` jest etykietą nadaną przez nadawcę.
- `MPI_ERROR` to kod błędu.
- Dokładną liczbę odebranych elementów poznamy przy pomocy funkcji:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int
*count)
```

Niebezpieczenstwo blokady (1)

- Proces 0 wysyła zmienną x i odbiera y od procesu 1

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0) {
   int x,y;
   MPI_Send(&x, 1, MPI_INT, 1, 13, MPI_COMM_WORLD);
   MPI_Recv(&y, 1, MPI_INT, 1, 13,MPI_COMM_WORLD,status);

} else if (myrank == 1) {
   int x,y;
   MPI_Send(&y, 1, MPI_INT, 0, 13, MPI_COMM_WORLD);
   MPI_Recv(&x, 1, MPI_INT, 0, 13,MPI_COMM_WORLD,status);
}
```

- Taki kod jest określany jako „niebezpieczny” (ang. unsafe).
- Do blokady dojdzie, jeżeli implementacja MPI nie stosuje buforowania. Jeżeli stosuje buforowanie program zadziała.
- Gdyby `MPI_Recv` było przed `MPI_Send` do blokady na pewno dojdzie (dlaczego).
- W praktyce implementacje stosują buforowanie dla komunikatów o długości mniejszej od pewnego progu, ale nie wolno Ci na tym polegać w swoich programach.
- Zmiana kolejności Send i Recv dla jednego z procesów rozwiązuje problem.

Niebezpieczenstwo blokady (2)

- Proces 0 wysyła zmienną x oraz zmienną y do procesu 1

```
nt x,y;
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0) {
   MPI_Send(&x, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(&y, 1, MPI_INT, 1, 2, MPI_COMM_WORLD);

} else if (myrank == 1) {
   int x,y;
   MPI_Recv(&y, 1, MPI_INT, 0,2,MPI_COMM_WORLD,status);
   MPI_Recv(&x, 1, MPI_INT, 0,1,MPI_COMM_WORLD,status);
}
```

- Gdzie tkwi problem ?
- Wniosek: Operacji MPI_Send oraz MPI_Recv nie można „dowolnie mieszać”

Operacje nieblokujące MPI_Isend MPI_Irecv

- **`MPI_Isend(buf,count,datatype,dest,tag,comm,request)`**
- **`MPI_Irecv(buf,count,datatype,source,tag,comm, request)`**
- Operacje te inicjują wysyłanie/odbiór i natychmiast powracają, nie czekając na zakończenie operacji. Rozpoczęta operacja jest identyfikowana przez parametr request (żądanie).
- Na paremetrze request identyfikującym operację “w drodze” możemy wykonać funkcje:
  - `MPI_Wait` czeka aż operacja identyfikowana przez request się zakończy
  - `MPI_Test` sprawdza czy operacja się zakończyła i natychmiast zwraca odpowiedni kod powrotu.
- Możliwe jest dowolne mieszanie operacji blokujących i nieblokujących u nadawcy i odbiorcy. Na przykład komunikat wysłany przez `MPI_Send` mozemy odebrac przez `MPI_Irecv`.

Przykład wykorzystania operacji nieblokujących

- Podobnie jak w poprzednim przykładzie proces 0 przesyła wartość zmiennej x procesowi 1.

```
int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   int x;
   MPI_Request req1;
   MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, &req1);
   compute();
   MPI_Wait(req1, status);
} else if (myrank == 1) {
   int x;
   MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

- Jednak w tym przypadku proces 0 rozpoczyna operację transmisji danych funkcją MPI_Isend, po czym wykonuje obliczenia (funkcja compute()). Po ich zakończeniu wywoływana jest funkcja MPI_Wait czekająca na zakończenie operacji Isend.

- Przykład ten pokazuje bardzo ważna technikę pozwalającą na prowadzenie obliczeń współbieżnie z komunikacją. Pozwala to niekiedy na ukrycie kosztów komunikacji (ang. latency hiding)

Unikniecie blokady - operacje nieblokujące

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0) {
   int x,y;
   MPI_Request reqtab[2];
   MPI_Status stattab[2];
   // rozpocznij wysyłanie
   MPI_Isend(&x, 1, MPI_INT, 1, 13, MPI_COMM_WORLD,&regtab[0]);
   // rozpocznij odbiór
   MPI_Irecv(&y, 1, MPI_INT, 1, 13,MPI_COMM_WORLD,&reqtab[1]);
   // Czekaj na zakończenie obydwu operacji
   MPI_Waitall(2,reqtab,stattab);
} else if (myrank == 1) {
   int x,y;
   MPI_Reuest reqtab[2];
   MPI_Status stattab[2];
   MPI_Isend(&y, 1, MPI_INT, 0, 13, MPI_COMM_WORLD,&regtab[0]);
   MPI_Irecv(&x, 1, MPI_INT, 0, 13,MPI_COMM_WORLD,&reqtab[1]);
   MPI_Waitall(2,reqtab,stattab);

}
```

- `MPI_Waitall` czeka na zakończenie wszystkich rozpoczętych operacji (Są też funkcje `MPI_Waitany/ MPI_Waitsome/ MPI_Testany/ MPI_Testall/ MP_Testsome`)
- Przypominam, że MPI_Request reprezentuje rozpoczętą operację.

Operacje nieblokujące – pułapki

- Pułapki oczywiste.
  - Nie można zmieniać danych (bufora) po wywołaniu `MPI_ISend` a przed wywołaniem (odpowiadajacym) `MPI_Wait`. Wynik nieokreślony
  - Nie można czytać ani modyfikowac danych po wywolaniu `MPI_IRecv` a przed wywołaniem `MPI_Wait`.
  - Nie można mieć rozpoczętych dwóch operacji `MPI_Irecv` do tego samego bufora.
- Pułapki mniej oczywiste
  - Nie można czytać danych po wywolaniu `MPI_ISend` a przed wywołaniem `MPI_Wait`.
  - Nie można mieć dwóch rozpoczętych operacji `MPI_ISend` z tego samego bufora.

Proces MPI_PROC_NULL i żądanie MPI_REQUEST_NULL

- Specjalna wartość `MPI_PROC_NULL` może być użyta w miejsce nadawcy oraz odbiorcy.
- Komunikacja z procesem `MPI_PROC_NULL` jest operacją pustą.
- Operacja Send z argumentem odbiorcy równym `MPI_PROC_NULL` powraca natychmiast.
- Operacja Receive z argumentem nadawcy równym `MPI_PROC_NULL` powraca natychmiast nie modyfikując bufora odbiorczego.

- Specjalna wartość `MPI_REQUEST_NULL` może być zastosowana jako argument operacji z rodziny `MPI_Wait` oraz `MPI_Test`.
- Wywołanie operacji z rodziny `MPI_Wait` z argumentem `MPI_REQUEST_NULL` powraca natychmiast.
- Wywołanie operacji z rodziny `MPI_Test` z argumentem `MPI_REQUEST_NULL` powraca natychmiast ustawiając wartość `flag` na true.

Funkcja MPI_Sendrecv

- Poprzedni wzorzec komunikacji występuje bardzo często w programach MPI. Dla jego obsłużenia zaprojektowano funkcję `MPI_Sendrecv`, która jednocześnie wysyła i odbiera komunikat. Jej parametry są połączeniem parametrów `MPI_Send` oraz `MPI_Recv`.

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
senddatatype, int dest, int      sendtag, void *recvbuf, int
recvcount,     MPI_Datatype recvdatatype, int source, int
recvtag,    MPI_Comm comm, MPI_Status *status)
```

- Odbiorca (`dest`) i nadawca (`source`) komunikatu mogą ale nie muszą być identyczni. Wysyłane i odbierane dane muszą być tego samego typu.
- Komunikat wysłany/odebrany przez `MPI_Sendrecv` może być odebrany/wysłany innymi funkcjami MPI (np. `MPI_Recv` i `MPI_Send`)
- Sendbuf oraz recvbuf muszą wskazywać na rozłączne obszary pamięci. Jeżeli chcemy użyć jednego bufora, to należy posłużyć się funkcją `MPI_Sendrecv_replace`:

```
int MPI_Sendrecv_replace(void *buf, int count,MPI_Datatype
datatype, int dest, int sendtag, int source, int recvtag,
MPI_Comm comm,MPI_Status *status
```

Funkcje MPI_Probe oraz MPI_Iprobe

```
int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status
*status)
```

- Czeka (***wywołanie blokujące***) na nadejście komunikatu ze źródła source o etykiecie tag. Nie odbiera komunikatu. Pod adresem `status` umieszcza dane o komunikacie.

```
int MPI_Iprobe(int source,int tag,MPI_Comm comm,MPI_Status
*status)
```

- Sprawdza (***wywołanie nieblokujące***) czy nadszedł (jest gotowy) komunikat ze źródła source o etykiecie tag; jeżeli tak pod adresem `flag` wpisuje 1 a pod adresem `status` umieszcza dane o komunikacie.
- Po powrocie z `MPI_Probe` lub jeżeli test `MPI_Iprobe` da wynik pozytywny należy jeszcze odebrać komunikat np. przez `MPI_Recv`.
- Jako source możemy zastosować `MPI_ANY_SOURCE` a jako tag `MPI_ANY_TAG`.
- Przykład zastosowania, chcemy odebrać komunikat, ale nie znamy jego długości i nie wiemy jak duży bufora zaalokować (`MPI_Probe /MPI_Get_count /MPI_Recv`).

MPI - pomiar czasu

```
double t1=MPI_Wtime();
...
...
... wykonaj obliczenia .....
...
...
double t2=MPI_Wtime();
printf(”Czas obliczeń: %f sekund\n”,t2-t1);
```

- Funkcja `MPI_Wtime` zwraca ilość sekund, jakie upłynęły od pewnej ustalonej chwili w przeszłości
  - zmienna double, rozdzielczość sekundach (czas pomiędzy dwoma tyknięciami zegara) zwracana funkcją `MPI_Wtick()` .
- Zegary nie są zsynchronizowane pomiędzy procesami, chyba że stała `MPI_WTIME_IS_GLOBAL` jest ustawiona na 1.