# MPI – Message Passing Interface

1. MPI – Message Passing Interface .... 2
1.1 Wstęp .... 2
1.2 Instalacja i konfiguracja .... 5
1.2.1 Instalacja pakietu lam/mpi .... 5
1.2.2 Sprawdzenie komunikacji wewnątrz klastra .... 5
1.2.3 Konfiguracja usługi rsh lub ssh .... 5
1.2.4 Utworzenie pliku konfiguracyjnego .... 5
1.2.5 Uruchomienie systemu .... 5
1.2.6 Testowanie systemu .... 6
1.2.7 Kompilacja .... 6
1.2.8 Uruchomienie .... 6
1.2.9 Testowanie .... 6
1.3 Podstawy .... 7
1.4 Komunikatory .... 8
1.5 Grupy i operacje na grupach .... 9
1.6 Operacje na komunikatorach .... 12
1.7 Podstawowe typy danych .... 14
1.8 Komunikacja punkt – punkt .... 15
1.8.1 Wysyłanie i odbiór komunikatów .... 15
1.8.2 Przykład – Znajdowanie liczb pierwszych w przedziale .... 18
1.8.3 Synchronizacja w komunikacji punkt - punkt .... 20
1.9 Tworzenie nowych typów danych .... 27
1.10 Operacje grupowe .... 30
1.11 Topologie wirtualne .... 41
1.12 Tworzenie procesów .... 46
1.13 Własności systemów MPI .... 47

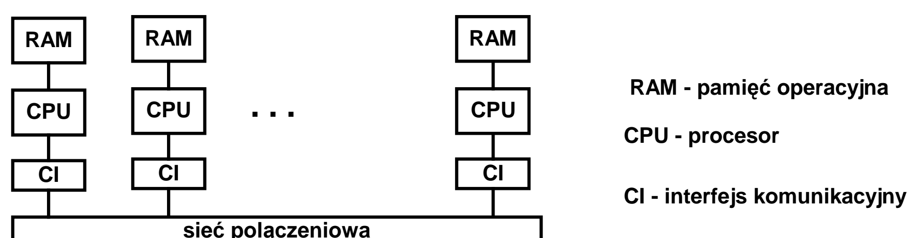


PDF created with pdfFactory Pro trial version www.pdffactory.com

# 1. MPI – Message Passing Interface

## 1.1 Wstęp

MPI (ang. *Message Passing Interface*) jest standardem systemu programowania równoległego dla maszyn składających się z komputerów z pamięcią lokalną. MPI nie jest językiem programowania, ale specyfikacją interfejsu.



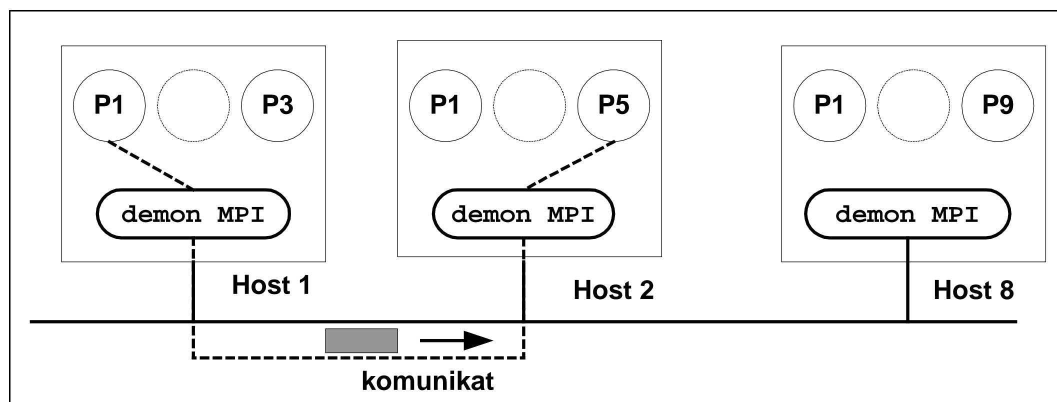

Rysunek 1-1 Multikomputer każdy procesor ma własną pamięć lokalną

Cel:

- Opracowanie jednolitego interfejsu programistycznego (API) dla programowania aplikacji wykorzystujących przekazywanie komunikatów.
- Zapewnienie efektywnego systemu przekazywania komunikatów
- Możliwość implementacji w środowiskach składających się z komputerów różnego typu (heterogenicznych)
- Łączenie z językami C, C++, Fortran 77 i Fortran 95
- Zapewnienie niezawodnej komunikacji, użytkownik zwolniony z obsługi błędów.
- Powinna być możliwość implementacji w różnych systemach bez konieczności dokonywania istotnych modyfikacji ich systemów komunikacji
- Semantyka niezależna od konkretnego języka
- Możliwość programowania wielowątkowego



PDF created with pdfFactory Pro trial version www.pdffactory.com




Rysunek 1-2 Maszyna wirtualna MPI. Zadanie P1 z węzła 1 komunikuje się z zadaniem 5 z węzła 2

Historia:

- 1992 - Pierwsza propozycja standardu, znana jako MPI1 opracowana przez Dongarrę, Hempela, Heya i Walkera,
- 1993 – Pierwsza wersja standardu MPI opracowana przez MPI forum http://www.mpi-forum.org i przedstawiona na konferencji Supercomputing 93
- 1995 – Poprawiona wersja standardu znana jako MPI 1.1
- 2002 – Wersja 2.0
- 2007 – Wersja 2.1

Standard obejmuje:

- Używane typy danych
- Komunikację punkt – punkt
- Operacje grupowe
- Zarządzanie otoczeniem
- Tworzenie procesów i zarządzanie nimi
- Równoległe wejście / wyjście
- Łączenie z językami C, C++, Fortran
- Interfejs do profilowania aplikacji

Liczba funkcji:

- MPI 1.1 – 124
- MPI 2.0 – 124 + 120



PDF created with pdfFactory Pro trial version www.pdffactory.com

Do posługiwania się systemem wystarcza 6 funkcji:

| MPI_Init | Inicjalizacja systemu |
|---|---|
| MPI_Finalize | Wyjście z systemu |
| MPI_Comm_size | Odczyt liczby procesów w komunikatorze |
| MPI_Comm_rank | Pobranie numeru procesu bieżącego |
| MPI_Send | Wysłanie komunikatu |
| MPI_Recv | Odbiór komunikatu |

Wersje:

- MPICH - Argonne National Laboratory - http://www-unix.mcs.anl.gov/mpi/
- LAM-MPI - http://www.lam-mpi.org/
- Open MPI - http://www.open-mpi.org/
- Deino – wersja MPI 2 dla .net - http://mpi.deino.net/index.htm

Implementacje:

- Unix, AIX, FreeBSD, HP-UX, IRIX, LINUX, Solaris, and SunOS
- Windows NT, Windows 2000
- Inne maszyny IBM SP, Intel i860, Delta, Paragon , CRAY T3D, Meiko CS2, Ncube 2
- Microsoft HPC Server 2008 i Windows Compute Cluster Server 2003



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.2 Instalacja i konfiguracja

### 1.2.1 Instalacja pakietu lam/mpi

Należy zainstalować pakiet w wybranym systemie operacyjnym na grupie komputerów połączonych siecią (klastrze). Niech będą to komputery node1, node2, node3, node4. Sprawdzić prawidłowość instalacji za pomocą polecenia:
$laminfo

### 1.2.2 Sprawdzenie komunikacji wewnątrz klastra

Sprawdzić czy komputery klastra komunikują się.

### 1.2.3 Konfiguracja usługi rsh lub ssh

Skonfigurować usługę rsh lub ssh na komputerach klastrach wraz z prawami dostępu tak aby można było uruchamiać programy na zdalnych węzłach bez podawania haseł. Gdy stosujemy rsh należy przeprowadzić edycję pliku ./rhosts w katalogach domowych użytkownika komputerów klastra. Gdy stosujemy ssh należy poustawiać klucze SSH używające ssh.

### 1.2.4 Utworzenie pliku konfiguracyjnego

Utworzyć plik konfiguracyjny np. o nazwie np. hostfile zawierający nazwy lub adresy IP komputerów wchodzących w skład klastra oraz wskazujący ile każdy z komputerów ma procesorów (jest to informacja ile procesów na danym węźle można utworzyć).

```
node1.cluster.pwr.wroc.pl cpu=2
node2.cluster.pwr.wroc.pl cpu=4
node3.cluster.pwr.wroc.pl
node4.cluster.pwr.wroc.pl cpu=2
```

Przykład 1 Plik konfiguracyjny hostfile

### 1.2.5 Uruchomienie systemu

Uruchomić demon systemu poleceniem:

```
$lamboot -v -ssi boot rsh hostfile
lub
$lamboot -v -ssi boot ssh hostfile
```



PDF created with pdfFactory Pro trial version www.pdffactory.com

### 1.2.6 Testowanie systemu

Sprawdzić czy węzły są widoczne poleceniem:

```
l$ lamnodes
n0 node1.cluster.pwr.wroc.pl:2:origin,this node
n1 node2.cluster.pwr.wroc.pl:4:
n2 node3.cluster.pwr.wroc.pl:1:
n3 node4.cluster.pwr.wroc.pl:2:
```

### 1.2.7 Kompilacja

Kompilację wykonuje się poprzez wykonanie polecenia:

```
$mpicc hello.c -o hello
```

### 1.2.8 Uruchomienie

**Program uruchamia się poleceniem:**

```
$mpirun -C hello
```

Opcja -C uruchom jedną kopię na każdym z procesorów

```
$mpirun -np 10 hello
```

Opcja -np N uruchamia N kopii procesu w środowisku MPI. Procesy będą szeregowane zgodnie z podaną liczbą procesorów według algorytmu round robin.

Według specyfikacji MPI 2.1 lepiej używać polecenia mpiexec.

```
$mpiexec -n1 manager:worker
```

Uruchomienie na każdym z komputerów jednej kopii procesu `manager` i jednej `worker`.

### 1.2.9 Testowanie

Wyświetlenie wykonywanych zadań - polecenie:

```
$mpitask
```

Skasowanie wszystkich wykonywanych programów:

```
$lamclean
```

Zatrzymanie systemu:

```
$lamhalt

$lamwipe hostfile
```

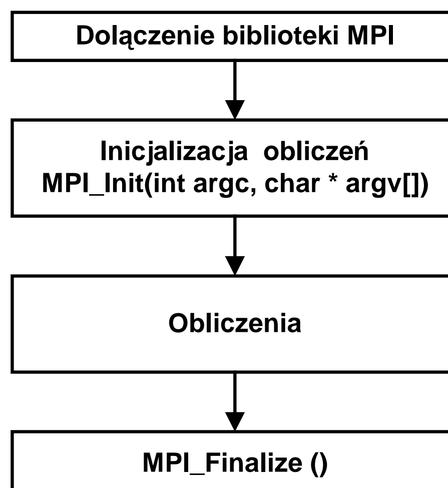


PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.3 Podstawy

*Struktura programu i podstawowe funkcje*

Plik nagłówkowy **#include "mpi.h"**




Rysunek 1-3 Struktura programu w MPI

*Inicjalizacja obliczeń*

Funkcja musi być wywołana na początku obliczeń, przekazywane są argumenty funkcji main.

**MPI_Init** - Inicjalizacja obliczeń

**int MPI_Init( int *argc, char *argv[] )**

**argc** Liczba argumentów programu głównego

**argv** Wskaźnik na tablicę z parametrami

Funkcja może pobrać przeznaczone dla MPI opcje i zwrócić zmienione argumenty, które mogą być wykorzystane w programie.

Funkcja zwraca:

**MPI_SUCCESS** – gdy sukces

Gdy wystąpi błąd jego kod zależy od implementacji

*Zakończenie obliczeń*

```
MPI_Finalize – zakończenie obliczeń
MPI_Finalize (void)
```

Po wywołaniu tej funkcji nie można wywoływać żadnej funkcji MPI



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.4 Komunikatory

MPI pozwala na jednoczesne uruchomienie wielu kopii tego samego programu:

```
$mpiexec -n liczba_procesów program
```

Komunikator tworzy grupa procesów oraz kontekst ich wykonania. Komunikator identyfikowany jest przez uchwyt. Po uruchomieniu aplikacji powstaje komunikator MPI_COMM_WORLD. Jest to grupa procesów, które mogą się ze sobą komunikować.

*Pobranie liczby procesów w komunikatorze*

**`MPI_Comm_size`** – pobranie liczby procesów w komunikatorze

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

| | |
|---|---|
| `Comm` | Komunikator |
| `Size` | Liczba procesów w komunikatorze |

Wykonanie funkcji powoduje umieszczenie w zmiennej size liczby procesów w komunikatorze.

*Identyfikacja procesu*

**`MPI_Comm_rank`** – identyfikacja procesu

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

| | |
|---|---|
| `comm` | Komunikator |
| `rank` | Numer procesu w komunikatorze |

Funkcja wpisuje do zmiennej **`rank`** numer bieżącego procesu w danym komunikatorze.

*Zakończenie wszystkich procesów komunikatora*

**`MPI_Abort`** – zakończenie wszystkich procesów

```
int MPI_Abort( MPI_Comm comm, int errcode)
```

| | |
|---|---|
| `comm` | Komunikator |
| `errcode` | Kod błędu zwracany do procesu który zainicjował aplikację |



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[]) {
   int procesow, rank, rc;
   rc = MPI_Init(&argc,&argv);
   if (rc != 0) {
     printf ("Blad inicjacji\n");
     MPI_Abort(MPI_COMM_WORLD, rc);
   }

   MPI_Comm_size(MPI_COMM_WORLD,&procesow);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   printf ("Il. proc= %d mój numer = %d\n",procesow,rank);
   MPI_Finalize();
}
```

Przykład 2 Program test testujący komunikator

```
$mpirun -np 4 test
$ Il. Proc=4 mój numer = 0
$ Il. Proc=4 mój numer = 1
$ Il. Proc=4 mój numer = 2
$ Il. Proc=4 mój numer = 3
```

## 1.5 Grupy i operacje na grupach

Grupa jest zbiorem procesów. Każdy proces w grupie jest identyfikowany przez indeks.

- Gdy w grupie jest N procesów to ich indeksy są kolejnymi liczbami z przedziału 0 do N-1.
- Grupa jest identyfikowana przez uchwyt MPI_Group.
- Istnieje predefiniowana grupa pusta identyfikowana przez uchwyt MPI_GROUP_EMPTY

`MPI_Com_group` Utworzenie kopii grupy

`int MPI_Com_group(MPI_Comm * comm, MPI_Group *group)`

`comm` Komunikator
`group` Uchwyt grupy



PDF created with pdfFactory Pro trial version www.pdffactory.com

Funkcja tworzy kopie grupy identyfikowanej przez komunikator comm.

`MPI_Group_size` Policzenie elementów grupy

```
int MPI_Group_size(MPI_Group *group, int *size)
```

`Group` Uchwyt grupy
`Size` Liczba elementów grupy

Po wykonaniu funkcji do zmiennej size wpisywana jest liczba elementów grupy.
Na grupach można wykonywać operacje mnogościowe:

- sumy,
- iloczynu,
- różnicy:

```
int MPI_Group_union(MPI_Group g1, MPI_Group g2, MPI_Group *new)
int MPI_Group_intersection(MPI_Group g1, MPI_Group g2, MPI_Group *new)
int MPI_Group_difference(MPI_Group g1, MPI_Group g2, MPI_Group *new)
```

- MPI_Group_union – suma mnogościowa do grupy new należą te elementy które należą do g1 lub g2.
- MPI_Group_ intersection – iloczyn mnogościowy do grupy new należą te elementy które należą do g1 i g2.
- MPI_Group_ difference – różnica mnogościowy do grupy new należą te elementy które należą do g1 ale nie należą do g2.

Do grupy można jawnie włączać procesy, których numery podane są w tablicy.

`MPI_Group_incl` Włączanie procesów do grupy

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group newgroup)
```

`group` Uchwyt grupy źródłowej
`n` Liczba dodawanych elementów
`ranks` Tablica z numerami nowych procesów
`newgroup` Uchwyt grupy wynikowej



PDF created with pdfFactory Pro trial version www.pdffactory.com

Funkcja tworzy nową grupę (ze starej) umieszczając w niej elementy podane w tablicy ranks.

```
#include "mpi.h";
  MPI_Group wszystkie, odd_group, even_group;;
  int i, p, Neven, Nodd, members[8], ierr;
  ...
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_group(MPI_COMM_WORLD, &wszystkie);

  // procesy z MPI_COMM_WORLD są dzielone na parzyste i
  // nieparzyste
  Neven = (p+1)/2;    Nodd = p - Neven;
  for (i=0; i<Neven; i++) {  members[i] = 2*i; };
  MPI_Group_incl(wszystkie, Neven, members, &even_group);
...
```

Przykład 3 Utworzenie grupy procesów parzystych

Usunięcie grupy:

```
int MPI_Group_free(MPI_Group group)
```



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.6 Operacje na komunikatorach

Kopiowanie komunikatora

**MPI_Comm_dup** Tworzenie kopii komunikatora

```
int MPI_Comm_dup( MPI_Comm comm, MPI_Comm comm *newcomm )
```

**comm** Komunikator źródłowy
**newcom** Komunikator wynikowy

Funkcja tworzy kopię komunikatora zawierającą te same procesy co komunikator źródłowy. Nowy komunikator otrzymuje nowy kontekst.

Do utworzenia nowego komunikatora można użyć grupę procesów.

**MPI_Comm_create** Tworzenie nowego komunikatora z grupy procesów

```
int MPI_Comm_create( MPI_Comm comm, MPI_Group
newgroup,MPI_Comm *newcomm )
```

**comm** Komunikator źródłowy
**newgroup** Uchwyt grupy wynikowej
**newcomm** Nowy komunikator (wynikowy)

Nowa grupa powinna być podgrupą grupy komunikatora pierwotnego.

Nowy komunikator może też być otrzymany z podziału starego komunikatora na grupy.

**MPI_Comm_split** Tworzenie nowego komunikatora poprzez podział

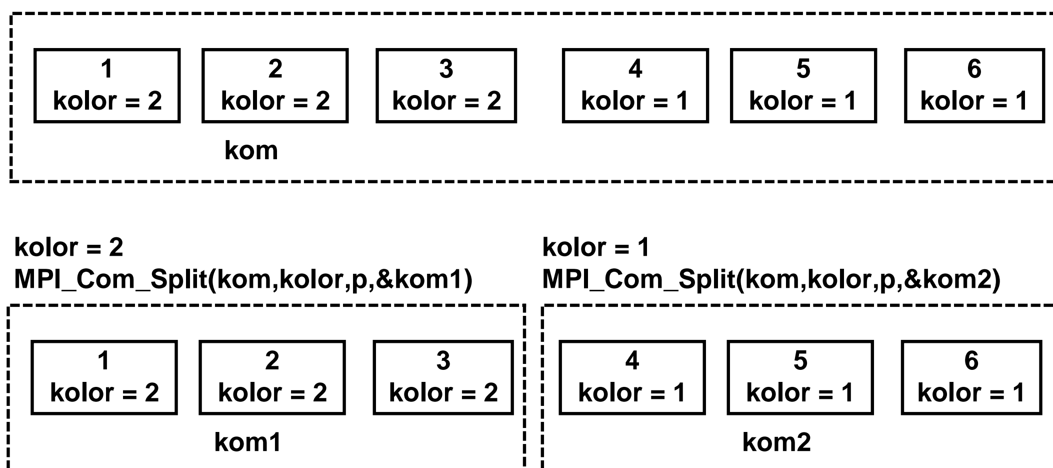
```
int MPI_Comm_split ( MPI_Comm comm, int color, int key
,MPI_Comm *newcomm )
```

**comm** Komunikator źródłowy
**color** Procesy o jednakowej wartości tego parametru będą włączone do nowego komunikatora
**key** Priorytet
**newcomm** Nowy komunikator (wynikowy)

Liczba nowych komunikatorów równa jest liczbie różnych wartości parametru color.



PDF created with pdfFactory Pro trial version www.pdffactory.com




Rysunek 1-4 Podział komunikatora kom na kom1 i kom2

```
int main( int argc, char *argv[] )
{
    MPI_Status status;
    MPI_Comm comm,scomm;
    int rank, color, errs=0;
    MPI_Init( 0, 0 );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    color = rank % 2;
    MPI_Comm_split( MPI_COMM_WORLD, color, rank, &scomm );
  …
}
```

Przykład 4 Podział komunikatora na 2 komunikatory



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.7 Podstawowe typy danych

W funkcjach wysyłania i odbierania komunikatów pojawiają się typy danych. Ich symboliczne oznaczenia pokazuje tabela.

| | |
|---|---|
| **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 bitów** |
| **MPI_PACKED** | **dane spakowane lub rozpakowane z użyciem MPI_Pack()lub MPI_Unpack()** |

Tab. 1-1 Typy danych w systemie MPI



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.8 Komunikacja punkt – punkt

### 1.8.1 Wysyłanie i odbiór komunikatów

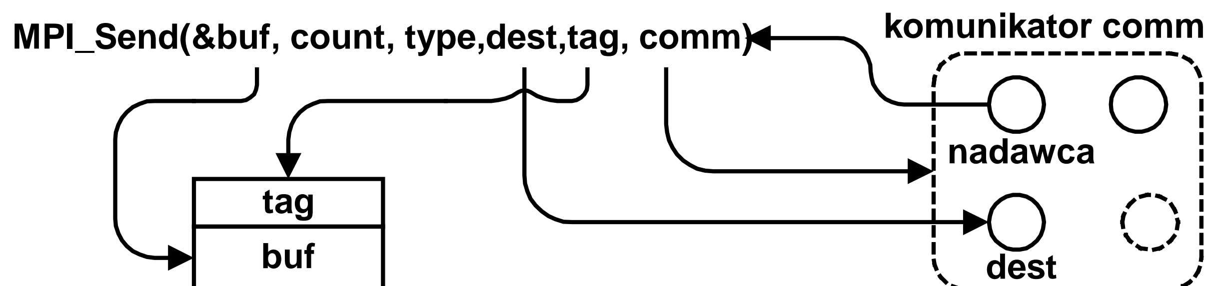
*Synchroniczne wysłanie komunikatu*

**MPI_Send** Synchroniczne wysłanie komunikatu

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

| | |
|---|---|
| **buf** | Bufor danych wysyłanych |
| **count** | Liczba elementów w buforze danych wysyłanych |
| **datatype** | Typ danych |
| **dest** | Numer procesu docelowego |
| **tag** | Etykietka danych z zakresu 0 do MPI_TAG_UB |
| **comm** | Komunikator |

Proces bieżący pozostaje zablokowany do chwili odebrania komunikatu.




Rys. 1-1 Znaczenie parametrów funkcji MPI_Send(...)



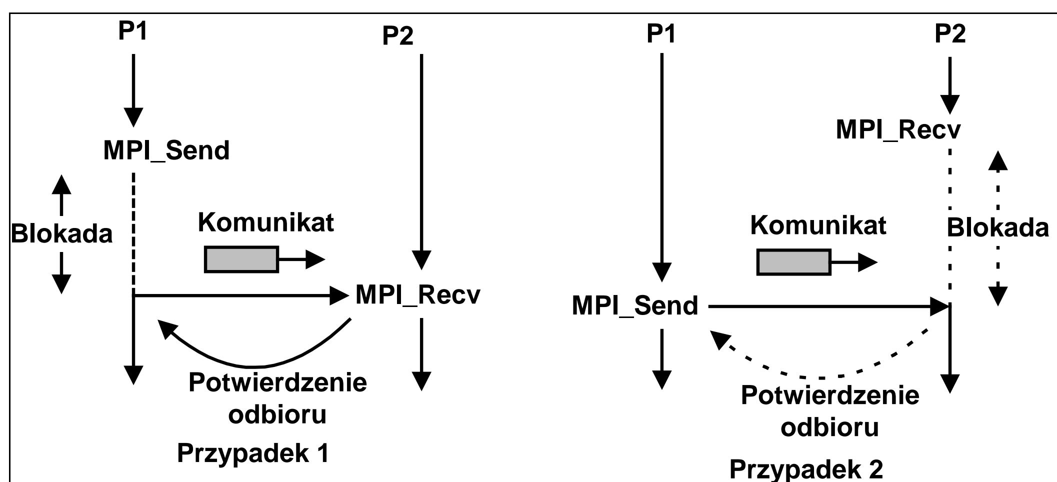
PDF created with pdfFactory Pro trial version www.pdffactory.com

**MPI_Recv** Synchroniczny odbiór komunikatu

```
int MPI_Recv( void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm, MPI_Status
*status )
```

| | |
|---|---|
| `buf` | Bufor danych odebranych |
| `count` | Maksymalna elementów w buforze danych odbieranych |
| `datatype` | Typ danych |
| `source` | Numer procesu wysyłającego, może być MPI_ANY_SOURCE wtedy odbiór z dowolnego procesu. |
| `tag` | Etykietka danych, może być MPI_ANY_TAG wtedy odbiór komunikatu z dowolną etykietką. |
| `comm` | Komunikator |
| `status` | Status |

Funkcja powoduje zablokowanie procesu bieżącego o ile brak procesu wysyłającego komunikat. Funkcja oczekuje na komunikat o etykietce tag chyba że wyspecyfikowano etykietkę MPI_ANY_TAG.




Rys. 1-2 MPI - komunikacja synchroniczna



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[]) {
   char msg[20];
   int myrank, tag = 1;
   MPI_STATUS status;
   rc = MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   if (myrank == 0){
     strcpy(msg, "Hello there");
     MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag,
              MPI_COMM_WORLD);
   }
   if (myrank == 1){
     MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
     &status);
     printf("Odebrano: %s\n",msg);
   }
   MPI_Finalize();
}
```

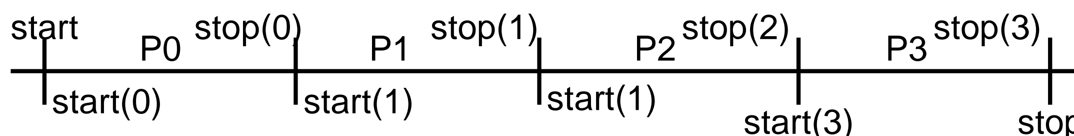
Przykład 5 Komunikacja synchroniczna w MPI



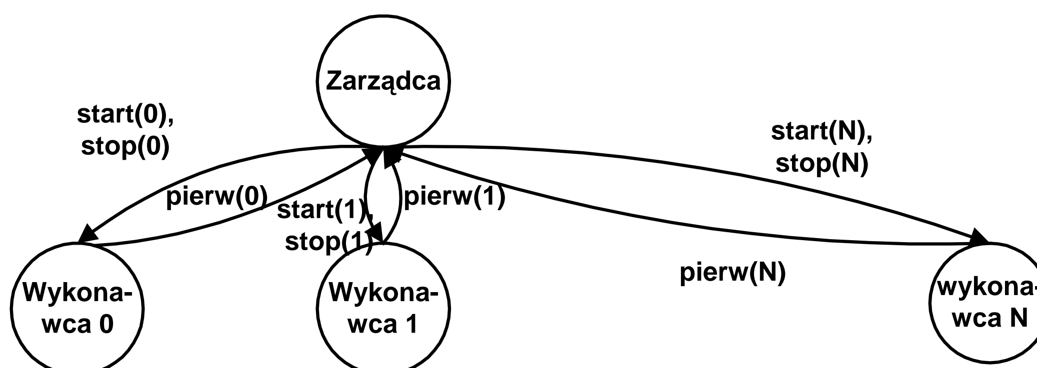
PDF created with pdfFactory Pro trial version www.pdffactory.com

### 1.8.2 Przykład – Znajdowanie liczb pierwszych w przedziale

Należy znaleźć wszystkie liczby pierwsze w przedziale od start do stop przy wykorzystaniu N procesorów.




Rysunek 1-5 Podział odcinka [start, stop] na 4 części




Rysunek 1-6 Podział zadania pomiędzy procesy wykonawcze

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

int pierwsza(int liczba){ // Czy pierwsza ?
  int i;
  if (liczba==1) return 0;
  for (i=2;(i*i<=liczba);i++) {
      if(((liczba%i)==0)) return 0;
  }
  return 1;
};

main(int argc, char *argv[]) {
  int liczba=2;
  int ierr, size, rank, ile, ile_danych, i;
  int start,stop,pierwszych=0;
  int bufor,Suma=0;
  char nazwa[50];
  int start, stop;
```



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  ile  = atoi(argv[1]);  // Ile liczb sprawdzamy
  size = atoi(argv[2]);  // Ile procesów roboczych

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

  if (rank==0)  { // Zarzadca ----------------------
   ile_danych= ile/size;
   start=1;
   stop=ile_danych;

   for(i=1;i<size;i++)  {
      MPI_Send(&start,1,MPI_INT,i,1,MPI_COMM_WORLD);
      MPI_Send(&stop,1,MPI_INT,i,2,MPI_COMM_WORLD);
      start+=ile_danych;
      stop+=ile_danych;
   }

   for(i =0;i<size;i++) {
       MPI_Recv(&bufor,1,MPI_INT,MPI_ANY_SOURCE,3,
              MPI_COMM_WORLD,0);
       Suma=bufor+Suma;
    }
    printf("Liczb pierwszych %d \n",Suma);

  } else  {

     // Worker -----------------------------------
     MPI_Recv(&start,1,MPI_INT,0,1,MPI_COMM_WORLD,0);
     MPI_Recv(&stop,1,MPI_INT,0,2,MPI_COMM_WORLD,0);
     for (int i=start;i<=stop;i++) {
         pierwszych+=pierwsza(i);
     }
     MPI_Send(&pierwszych,1,MPI_INT,0,3,MPI_COMM_WORLD);
  }
 MPI_Finalize();
 return 0;
}
```

Przykład 6 Program znajdowania liczb pierwszych



PDF created with pdfFactory Pro trial version www.pdffactory.com

### 1.8.3 Synchronizacja w komunikacji punkt - punkt

W MPI występuje wiele rodzajów synchronizacji.

- Ze względu na współdziałanie strony nadającej i odbierającej:
  *Komunikacja synchroniczna* – nadawca czeka aż odbiorca przystąpi do odbioru
  *Komunikacja asynchroniczna* – gdy odbiorca nie jest gotowy to nadawca umieszcza komunikat w buforze.

- Ze względu na blokowanie:
  *Komunikacja blokująca* – powrót z funkcji wysyłania / odbioru następuje po zakończeniu działań funkcji
  *Komunikacja nie blokująca* – powrót z funkcji wysyłania / odbioru następuje natychmiast

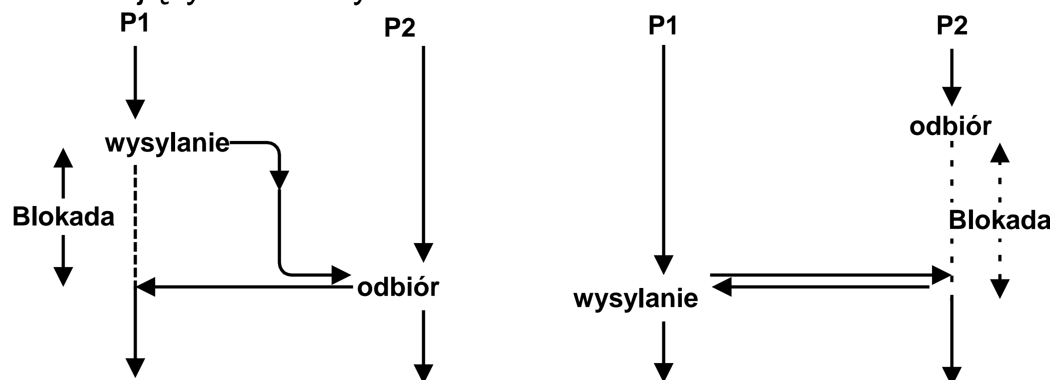
*Rodzaje funkcji wysyłania i odbierania*

W zależności od kombinacji opcji współdziałania i buforowania występuje wiele rodzajów funkcji.

| | | | |
|---|---|---|---|
| B | Buforowany | *Buffered* | Wysyłanie do bufora który musi być wcześniej przygotowany |
| S | Synchroniczny | *Synchronous* | Komunikacja typu spotkaniowego |
| R | Przygotowany | *Ready* | Najpierw musi wystąpić gotowość odbioru |
| I | Nieblokujący | *Immediate* | Funkcja nieblokująca |

Tab. 1 Przedrostki funkcji wysyłania i odbierania komunikatów

*Tryb synchroniczny* – Proces wysyłający blokowany do czasu gdy proces odbierający zakończy odbiór.



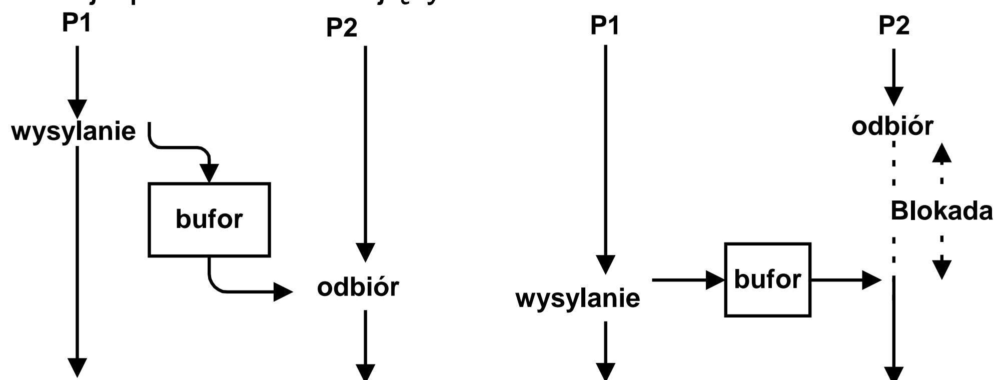

Rys. 1-3 MPI - Tryb synchroniczny



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
MPI_Ssend (buf,count,datatype,dest,tag,comm,ierr)
```

*Tryb buforowany* – proces wysyłający przesyła dane do bufora skąd pobiera je proces odbierający.




Rys. 1-4 MPI - Tryb buforowany

| | |
|---|---|
| **MPI_Ssend** | Wysyłanie synchroniczne blokujące |
| **MPI_Isend** | Standardowe wysyłanie nieblokujące |
| **MPI_Ibsend** | Buforowane wysyłanie nieblokujące |
| **MPI_Irsend** | Przygotowane wysyłanie nieblokujące |
| **MPI_Recv** | Odbieranie blokujące |
| **MPI_Irecv** | Odbieranie nieblokujące |

Tab. 2 Zestawienie funkcji wysyłania i odbioru komunikatów

## *Buforowanie*

W komunikacji asynchronicznej musza być użyte bufory. Do przydziału i zwalniania buforów służą funkcje:

**MPI_Buffer_attach** Przydział bufora

```
int MPI_Buffer_attach( void *buffer, int size)
```

**buffer** Bufor danych
**size** Wielkość bufora

Funkcja tworzy bufor. Jego wielkość size powinna być nie mniejsza niż rozmiar pojedynczego komunikatu.

**MPI_Buffer_dettach** Zwolnienie bufora

```
int MPI_Buffer_dettach( void *buffer, int size)
```



PDF created with pdfFactory Pro trial version www.pdffactory.com

`buffer` Bufor danych
`size` Wielkość bufora

Funkcja zwalnia utworzony wcześniej bufor.

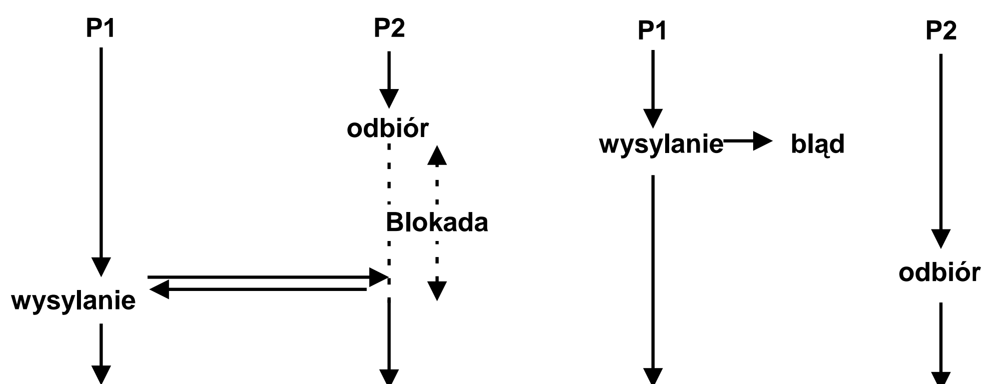
### *Nieblokujące wysyłanie buforowane.*

`MPI_Ibsend` Nieblokujące buforowane wysłanie komunikatu

```
int MPI_Ibsend( void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm com, MPI_Request
*request)
```

| | |
|---|---|
| `buf` | Bufor danych wysyłanych |
| `count` | Liczba elementów w buforze danych wysyłanych |
| `datatype` | Typ danych |
| `dest` | Numer procesu docelowego |
| `tag` | Etykietka danych z zakresu 0 do MPI_TAG_UB |
| `com` | Komunikator |
| `request` | Identyfikator zlecenia (uchwyt) |

*Tryb przygotowany* - komunikat może być wysłany tylko wtedy, gdy wcześnie została już wywołana funkcja odbierająca. Jeśli ten warunek nie jest spełniony, funkcja zwraca błąd.




Rys. 1-5 MPI - Tryb przygotowany

```
MPI_Irsend (*buf,count,datatype,dest,tag,comm)
```



PDF created with pdfFactory Pro trial version www.pdffactory.com

### *Finalizowanie operacji nieblokujących*

Przy wysyłaniu nieblokującym komunikat umieszczany jest w buforze który przez pewien czas nie może być użyty.

Zakończenie operacji wysłania - nadawcy wolno zmieniać zawartość bufora komunikatów. Nie oznacza to jednak że komunikat jest odebrany.

Zakończenie operacji odbioru - bufor odbiorczy zawiera pełny komunikat, a odbiorca może z niego korzystać.

**MPI_Wait** - blokujące testowanie zakończenia operacji

**MPI_Wait(MPI_Request *request, MPI_Status *status)**

| | |
|---|---|
| **request** | Identyfikaror operacji (uchyt) z funkcji **Ibsend** |
| **status** | Status operacji – struktura zawierająca różne pola informacyjne (tag, source, error,...) |

Funkcja blokuje proces wywołujący do czasu gdy operacja (wysyłania lub odbioru) wskazywana poprzez identyfikator request nie zostanie zakończona. Przy wysyłaniu nie oznacza to że proces odebrał informację z bufora a jedynie ze bufor może być znowu użyty. W zmiennej po zakończeniu operacji będzie wartość MPI REQUEST NULL.

Powyższa funkcja jest blokująca. Gdy istnieje potrzeba testowania zakończenia operacji bez możliwości blokady można użyć funkcji MPI_Test.

**MPI_Test** - nieblokujące testowanie zakończenia operacji

**MPI_Test(MPI_Request *request,int * flag, MPI_Status *status)**

| | |
|---|---|
| **request** | Identyfikaror operacji (uchyt) z funkcji **Ibsend** |
| **flag** | TRUE gdy operacja się zakończyła FALSE gdy nie |
| **status** | Status operacji – struktura zawierająca rózne pola informacyjne (tag, source, error,...) |



PDF created with pdfFactory Pro trial version www.pdffactory.com

Przydatnym narzędziem jest możliwość testowania czy zakończyła się jakakolwiek operacja co wykonuje funkcja MPI_Waitany.

```
int MPI_Waitany( int count, MPI_Request
*array_of_requests, int *index, MPI_Status *status )
```

| | |
|---|---|
| **count** | liczba testowanych zleceń |
| **array_of_requests** | tablica zawierająca uchwyty żądań |
| **index** | indeks do zakończonej operacji |

Funkcja blokuje się do zakończenia jakiejkolwiek operacji.

*Anulowanie operacji nieblokującej:*

```
MPI_Cancel (MPI_Request *request)
```

Istnieje możliwość testowania czy operacja się zakończyła co odbywa się za pomocą funkcji MPI_Iprobe.

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

Gdy w wyspecyfikowanym komunikatorze jest określona przez tag wiadomość od procesu source w zmiennej flag znajduje się wartość TRUE.



PDF created with pdfFactory Pro trial version www.pdffactory.com

## *Jednoczesne wysyłanie i odbieranie*

W MPI istnieje funkcja MPI_Sendrecv która umożliwia wysłanie komunikatu i oczekiwanie na odpowiedź.

**MPI_Sendrecv** Wysyłanie i odbiór komunikatu

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

| | |
|---|---|
| **sendbuf** | Bufor danych wysyłanych |
| **sendcount** | Liczba elementów w buforze danych wysyłanych |
| **sendtype** | Typ danych wysyłanych |
| **dest** | Numer procesu docelowego |
| **sendtag** | Etykietka danych z zakresu 0 do MPI_TAG_UB |
| **recvbuf** | Bufor danych odebranych |
| **recvcount** | Maksymalna elementów w buforze danych odbieranych |
| **recvtype** | Typ danych odbieranych |
| **source** | Od kogo chcemy odbierać |
| **recvtag** | Etykietka danych odbieranych |
| **comm** | Komunikator |
| **status** | Status |

W połączeniu z powyższą funkcją używa się standardowych funkcji wysyłania i odbierania.



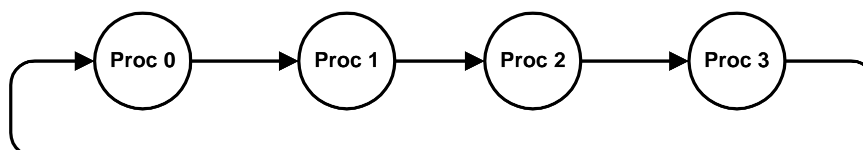
PDF created with pdfFactory Pro trial version www.pdffactory.com

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

int main(int argc, char *argv[]){
    int myid, numprocs, left, right;
    int buffer[50], buffer2[50];
    MPI_Request request;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    right = (myid + 1) % numprocs;
    left = myid - 1;
    if (left < 0) left = numprocs - 1;
    sprintf(buffer,”od procesu %d”,myid);
    MPI_Sendrecv(buffer, 50, MPI_INT, left, 33, buffer2,
    50, MPI_INT, right, 33, MPI_COMM_WORLD, &status);
    printf(„Jestem %d otrzymalem %s\n”,buffer2);
    MPI_Finalize();
    return 0;
}
```

Przykład 7 Cykliczne przekazywanie komunikatów

```
$mpicc sendrecv.c –o sendrecv
$lamboot –v
$mpirun –np 4 ./sendrecv
Jestem 1 otrzymalem od procesu 2
Jestem 2 otrzymalem od procesu 3
Jestem 0 otrzymalem od procesu 1
Jestem 3 otrzymalem od procesu 0
```

Listing 1 Wyniki przykładu cykliczne przekazywanie komunikatów




Rysunek 1-7 Cykliczne przekazywanie komunikatów



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.9 Tworzenie nowych typów danych

W systemie MPI można na podstawie istniejących typów danych tworzyć nowe. W szczególności:

- Tablice
- Wektory
- Struktury

Testowanie długości danych

Do testowania długości danych może być użyta funkcja **MPI_Type_extent**

**MPI_Type_extent** Testowanie długości danych

```
MPI_Type_extent(datatype,*extent)
```

| | |
|---|---|
| **datatype** | Typ danych |
| **extent** | Długość w bajtach |

Definiowanie struktury

Do definiowania struktury może być użyta funkcja **MPI_Type_struct**.

**MPI_Type_struct** Tworzenie struktury danych

```
MPI_Type_struct(count,blocklens[],offsets[], old_types[],
*newtype)
```

| | |
|---|---|
| **count** | Liczba bloków |
| **blocklens[]** | Tablica zawierająca długości bloków |
| **offsets[]** | Tablica zawierająca przesunięcia dla każdego bloku |
| **old_types[]** | Tablica zawierająca typy danych bloku |
| **newtype** | Uchwyt nowego typu danych |

Zatwierdzanie nowego typu danych

Nowo utworzony typ danych musi być zatwierdzony, co wykonywane jest przez funkcję **MPI_Type_commit**.

**MPI_Type_commit** Zatwierdzanie nowego typu danych

```
MPI_Type_commit (datatype)
```

| | |
|---|---|
| **datatype** | Uchwyt nowego typu danych |



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
#include "mpi.h"
#include <stdio.h>
#define NELEM 8
typedef struct {
  float x, y, z;
  float velocity;
  int  n, type;
} Particle;

main(int argc, char *argv[]) {
 int numtasks, rank, source=0, dest, tag=1, i;
 Particle p[NELEM], particles[NELEM];
 MPI_Datatype particletype, oldtypes[2];
 int blockcounts[2];
 MPI_Status stat;

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

 // Definicja 4 pol x, y, z, velocity typu MPI_FLOAT
 offsets[0] = 0;
 oldtypes[0] = MPI_FLOAT;
 blockcounts[0] = 4;

 // Definicja 2 pól n I type typu MPI_INT
 // Testowanie wymiaru typu  MPI_FLOAT
 MPI_Type_extent(MPI_FLOAT, &extent);
 offsets[1] = 4 * extent;
 oldtypes[1] = MPI_INT;
 blockcounts[1] = 2;

 // Definicja struktury particletype
 MPI_Type_struct(2, blockcounts, offsets, oldtypes,
                 &particletype);
 //  Zatwierdzenie struktury
 MPI_Type_commit(&particletype);
```



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
 // Inicjalizacja tablicy particles
 if (rank == 0) {
  for (i=0; i<NELEM; i++) {
     particles[i].x = i * 1.0;
     particles[i].y = i * -1.0;
     particles[i].z = i * 1.0;
     particles[i].velocity = 0.25;
     particles[i].n = i;
     particles[i].type = i % 2;
  }
  // Wyslanie tablicy do wszystkich zadan
  for (i=0; i<numtasks; i++) {
     MPI_Send(particles, NELEM, particletype, i,
              tag, MPI_COMM_WORLD);
  }
 }
 // Odbior struktury
 MPI_Recv(p, NELEM, particletype, source, tag,
           MPI_COMM_WORLD, &stat);
 // …
 MPI_Finalize();
}
```

Przykład 8 Definiowanie struktury



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.10 Operacje grupowe

System MPI umożliwia następujące operacje grupowe:

| | |
|---|---|
| Bariera | MPI_Barrier |
| Wysyłanie od jednego do wszystkich | MPI_Broadcast |
| Rozpraszanie bufora komunikatu na wszystkie procesy grupy | MPI_Scatter |
| Składanie komunikatów od wszystkich procesów w grupie do pojedynczego bufora | MPI_Gather |
| Składanie komunikatów od wszystkich procesów i rozesłanie ich do procesów grupy | MPI_Allgather |
| Redukuje wartości we wszystkich procesach do pojedynczej wartości | MPI_Reduce |

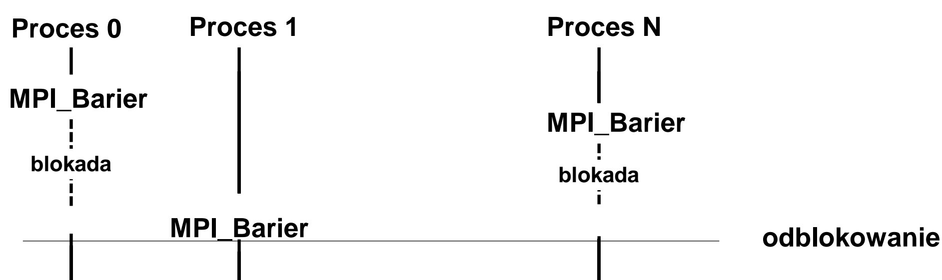
*Bariera*

**`MPI_Barier`** - bariera

**`MPI_Barier(MPI_Comm comm )`**

**`comm`** Identyfikator komunikatora

Funkcja blokuje proces bieżący do czasu aż wszystkie procesy komunikatora nie wywołają tej funkcji.




Rysunek 1-8 Działanie bariery



PDF created with pdfFactory Pro trial version www.pdffactory.com

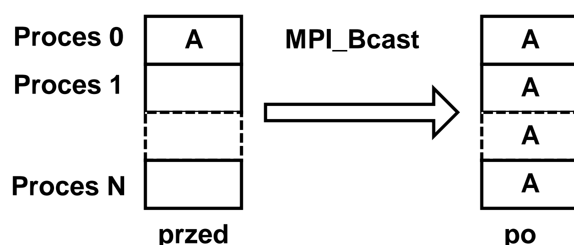
## *Rozsyłanie komunikatu do grupy procesów*

**MPI_Bcast** - rozsyłanie komunikatu do grupy procesów

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

| | |
|---|---|
| **buffer** | Bufor danych wysyłanych |
| **count** | Liczba elementów w buforze danych wysyłanych |
| **datatype** | Typ danych wysyłanych |
| **root** | Numer procesu głównego, który rozsyła dane |
| **comm** | Identyfikator komunikatora |

Funkcja powoduje przesłanie komunikatu umieszczonego w buforze **buffer** od procesu **root** do wszystkich procesów komunikatora. Proces **root** może być dowolnym procesem z komunikatora. Funkcja tak wysyła komunikaty (root) jak i odbiera (wszystkie procesu z wyjątkiem root).




Rysunek 1-9 Działanie mechanizmu rozgłaszania



PDF created with pdfFactory Pro trial version www.pdffactory.com

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

int main(int argc, char *argv[]){
    int myid, numprocs, left, right;
    char buffer[10];
    int root = 0;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if(myid == root) {
      strcpy(buffer,"ladunek");
    } else { strcpy(buffer,"????");
    }
    printf("Przed - jestem %d bufor %s\n",myid,buffer);
    MPI_Bcast(buffer, 10, MPI_CHAR, root, MPI_COMM_WORLD);
    printf("Po - jestem %d bufor %s\n",myid,buffer);
    MPI_Finalize();
    return 0;
}
```

Przykład 9 Ilustracja działania funkcji MPI_Bcast

```
Przed – jestem 0 bufor ladunek
Przed – jestem 1 bufor ????
Przed – jestem 2 bufor ????
Przed – jestem 3 bufor ????
Po – jestem 0 bufor ladunek
Po – jestem 1 bufor ladunek
Po – jestem 2 bufor ladunek
Po – jestem 3 bufor ladunek
```

Listing 2 Wyniki z Przykład 9



PDF created with pdfFactory Pro trial version www.pdffactory.com

## *Wysyłanie danych od jednego procesu w grupie do wszystkich*

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

**MPI_Scatter** - rozsyłanie wektora danych do grupy procesów

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

| | |
|---|---|
| **sendbuffer** | Bufor danych wysyłanych |
| **sendcount** | Długość podwektora z **sendbuf** |
| **sendtype** | Typ danych wysyłanych |
| **recvbuf** | Bufor odbiorczy |
| **recvcount** | Długość podwektora odbieranego w **recvbuf** |
| **recvtype** | Typ danych odbieranych |
| **root** | Numer procesu głównego który rozsyła dane |
| **comm** | Identyfikator komunikatora |

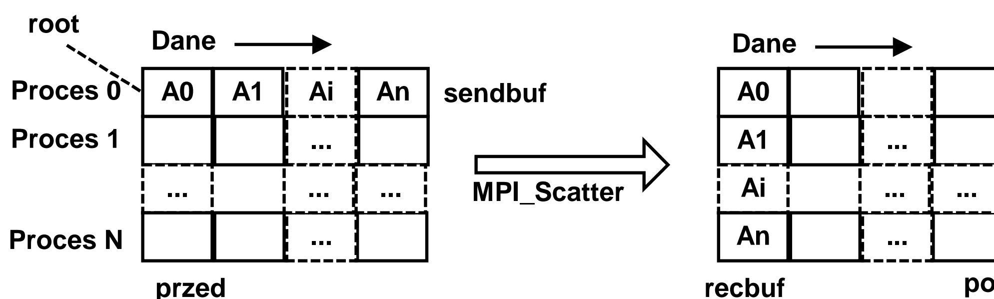
Proces o numerze root dzieli wektor **sendbuf** na **sendcount** elementów i rozsyła te fragmenty do wszystkich procesów komunikatora comm łącznie z sobą samym. Podwektory te są umieszczane w buforach **recbuf**. Wszystkie podwektory mają tę samą długość.

Process root:

```
MPI_Send(sendbuf + i * sizeof(sendtype) * sendcount,
sendcount, sendtype,i,..);
```

Każdy z procesów:

```
MPI_Recv(recvbuf, recvcount,i,...)
```




Rys. 1-6 Działanie operacji MPI_Scatter



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100,
MPI_INT, root, comm);
```

Przykład 10 Użycie funkcji MPI_Scatter – wysyłanie 100 elementowego wektora liczb int do procesów komunikatora.

## *Odbieranie danych od wszystkich procesów w grupie*

**MPI_Gather** - odbieranie wektora danych do grupy procesów

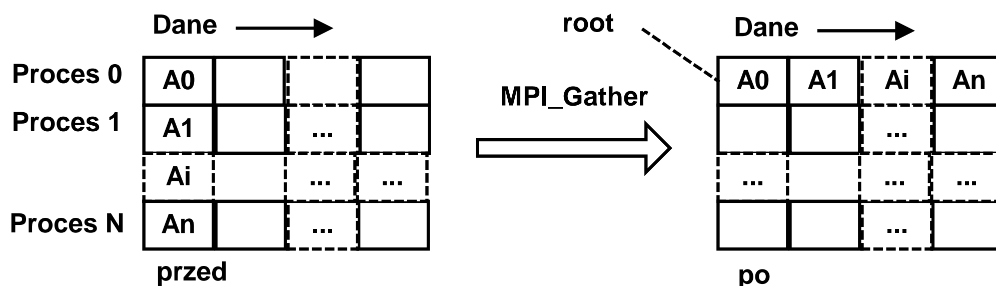
```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

| | |
|---|---|
| **sendbuffer** | Bufor danych wysyłanych |
| **sendcount** | Długość podwektora z sendbuf |
| **sendtype** | Typ danych wysyłanych |
| **recvbuf** | Bufor odbiorczy |
| **recvcount** | Długość podwektora odbieranego w recvbuf |
| **recvtype** | Typ danych odbieranych |
| **root** | Numer procesu głównego który rozsyła dane |
| **comm** | Identyfikator komunikatora |

Każdy z procesów grupy wysyła podwektor danych o długości **sendcount** umieszczony w buforze **sendbuf**. Proces o numerze root zbiera te dane i umieszcza w buforze **recvbuf**. Dane umieszczane są w kolejności zgodnej z numerami nadawców.



PDF created with pdfFactory Pro trial version www.pdffactory.com




Rys. 1-7 Działanie operacji MPI_Gather

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank( comm, &myrank);
if ( myrank == root) {
   MPI_Comm_size( comm, &gsize);
   rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100,
            MPI_INT, root, comm);
```

Przykład 11 Użycie funkcji MPI_Gather

*Odbiór danych od procesów w grupie i rozesłanie ich złożenia*

**MPI_Allgather** - odbieranie danych do grupy procesów i wysłanie do wszystkich ich złożenia

```
int MPI_Allgather(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm)
```

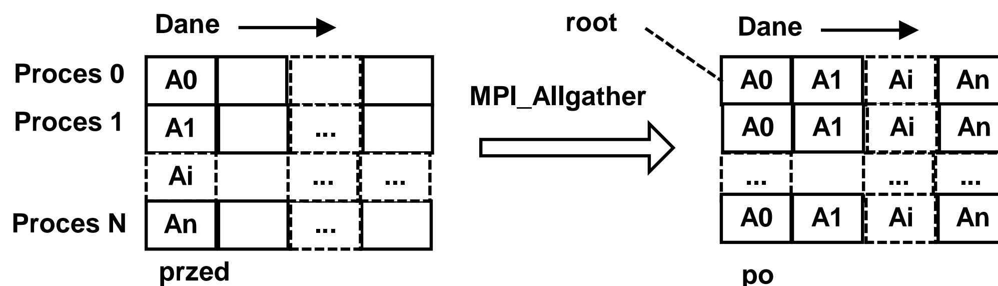
**sendbuffer** Bufor danych wysyłanych
**sendcount** Długość podwektora z sendbuf
**sendtype** Typ danych wysyłanych
**recvbuf** Bufor odbiorczy
**recvcount** Długość podwektora odbieranego w recvbuf
**recvtype** Typ danych odbieranych
**comm** Identyfikator komunikatora

Każdy z procesów grupy wysyła podwektor danych o długości **sendcount** umieszczony w buforze **sendbuf**. Dane te są składane w



PDF created with pdfFactory Pro trial version www.pdffactory.com

jeden wektor i wysyłane do bufora **recvbuf** wszystkich procesów komunikatora.




Rys. 1-8 Działanie operacji MPI_Allgather

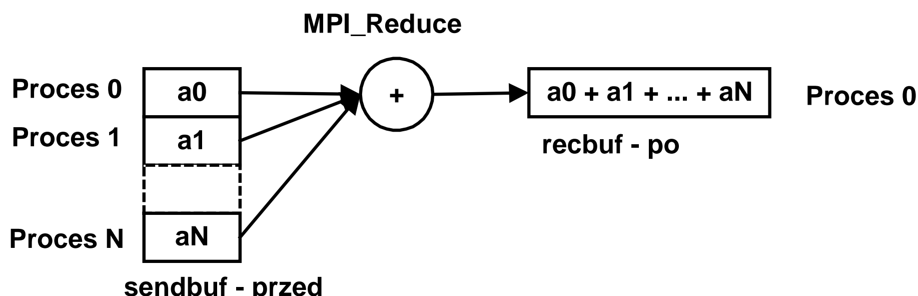
*Redukcja wartości we wszystkich procesach do pojedynczej wartości.*

**MPI_Reduce** Redukcja wartości we wszystkich procesach do pojedynczej wartości

```
int MPI_Reduce(void* sendbuf, void* recbuf , int count,
MPI_Datatype type, int op, int root, MPI_Comm comm)
```

| | |
|---|---|
| **sendbuf** | Bufor danych wysyłanych |
| **recbuf** | Bufor danych zredukowanych |
| **count** | Długość wektora danych |
| **type** | Typ danych |
| **op** | Rodzaj operacji |
| **root** | Numer procesu root |
| **comm** | Identyfikator komunikatora |

Funkcja wykonuje operację zdefiniowaną w zmiennej op na elementach zawartych w buforze **sendbuf** przez każdy z procesów w grupie. Wynik zapisywany jest do bufora **recbuf** w procesie zdefiniowanym jako root. Metoda jest wywoływana przez wszystkich członków grupy z tymi samymi wartościami argumentów.




Rysunek 1-10 Działanie mechanizmu redukcji na przykładzie sumowania



PDF created with pdfFactory Pro trial version www.pdffactory.com

| Oznaczenie | Operacja | Typ |
|---|---|---|
| **MPI_MAX** | maksimum | **Int, float** |
| **MPI_MIN** | minimum | **int, float** |
| **MPI_SUM** | Suma | **int, float** |
| **MPI_PROD** | produkt | **int, float** |
| **MPI_LAND** | Logiczne AND | **int** |
| **MPI_BAND** | Bitowe AND | **int, MPI_BYTE** |
| **MPI_LOR** | Logiczne OR | **int** |
| **MPI_BOR** | Bitowe OR | **int, MPI_BYTE** |
| **MPI_LXOR** | Logiczne XOR | **int** |
| **MPI_BXOR** | Bitowe XOR | **int, MPI_BYTE** |
| **MPI_MAXLOC** | Maksymalna wartość i położenie | **float, double, long double** |
| **MPI_MINLOC** | Minimalna wartość i położenie | **float, double, long double** |

Tabela 1-1 Działania w operacje redukcji funkcji MPI_Reduce



PDF created with pdfFactory Pro trial version www.pdffactory.com

Przykład – Sortowanie bąbelkowe

```
#include <stdio.h>
#define N 10
int main (int argc, char *argv[]) {
 int a[N];
 int i,j,tmp;
 for (i=0; i<N; i++) {
   a[i] = rand(); printf("%d\n",a[i]);
 }
 for (i=0; i<(N-1); i++)
  for(j=(N-1); j>i; j--)
    if (a[j-1] > a[j]) {
       tmp = a[j];a[j] = a[j-1];a[j-1] = tmp;
    }
 printf(„Posortowane\n”);
 for (i=0; i<N; i++) printf("%d\n",a[i]);
}
```

Przykład 12 Program sekwencyjny sortowanie bąbelkowego

Krok 1 – Dekompozycja
Ustalić w jaki sposób podzielić domenę

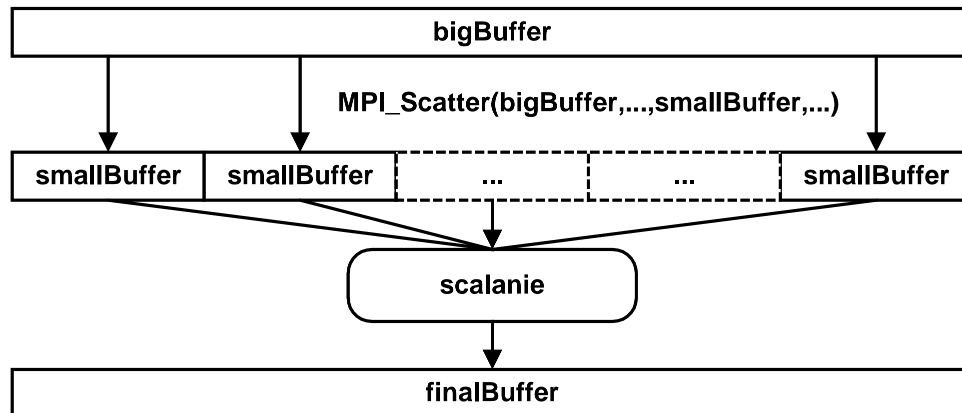
Krok 2 – Komunikacja
Ustalić sposób komunikacji pomiędzy procesami

Krok 3 – Aglomeracja
Zgrupować zadania aby uprościć problem i przyspieszyć obliczenia

Krok 4 – Rozmieszczenie
Przydzielić zadania do procesorów



PDF created with pdfFactory Pro trial version www.pdffactory.com




```
void init_root(char* input_fn, int** smallBuffer,
               int* bufSize, int* cnt){
  // Inicjalizacja danych,
  // pobranie danych z pliku input_fn i umieszczenie
  // ich w buforze bigBuffer dlugosci bufSize

  // Rozgloszenie  dlugosci bufSize malego bufora
  MPI_Bcast(bufSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Rozproszenie danych z bigBuffer do smallBuffer
  MPI_Scatter(bigBuffer, *bufSize, MPI_INT,
              *smallBuffer, *bufSize, MPI_INT, 0,
               MPI_COMM_WORLD);
}

void init_worker(int** smallBuffer, int* bufSize){
  // Pobierz dlugosc mojego segmentu danych
  MPI_Bcast(bufSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
  ...
  // Pobierz dane do obliczen
  MPI_Scatter(bigBuffer, *bufSize, MPI_INT,
  *smallBuffer, *bufSize, MPI_INT, 0, MPI_COMM_WORLD);
}

void merge(int* input, int* output, int bufSize,
           int totalSize){
   // Polaczenie danych z tabl. Input wielk bufSize
   // Do tablicy output wielk totalSize
}
```



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
void main(int argc, char* argv[]){
  // Inicjacja MPI
  // Master inicjalizuje obliczenia
  if(rank==0)
    init_root(input_fn, &inBuf, &bufSize, &totalSize);
  else  // Wykonawca pobiera dane
    init_worker(&inBuf, &bufSize);

  //  Sortowanie
  mysort(inBuf, bufSize);

  // Zlozenie posort. czesci w procesie zarzadzajacym
  MPI_Gather(inBuf, bufSize, MPI_INT, tmpBuf, bufSize,
             MPI_INT, 0, MPI_COMM_WORLD);
  if(rank==0){
    // Polaczenie posortowanych sekwencji w jedna
    merge(tmpBuf, finalBuf, bufSize, totalSize);
    // Zapis wynikow do pliku
  }
  // Zakonczenie obliczen
}
```

Przykład 13 Sortowanie bąbelkowe w systemie MPI



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.11 Topologie wirtualne

W standardowym komunikatorze procesy uporządkowane są liniowo od 0 do N.
Takie uporządkowanie nie zawsze odpowiada strukturze aplikacji. W praktyce często mamy do czynienia z następującymi strukturami procesów w aplikacji:

- Siatka dwuwymiarowa
- Siatka trójwymiarowa
- Graf

System MPI umożliwia tworzenie następujących topologii procesów

- Topologia kartezjańska
- Topologia grafowa

Utworzenie topologii odmiennych od uporządkowania liniowego upraszcza strukturę aplikacji.

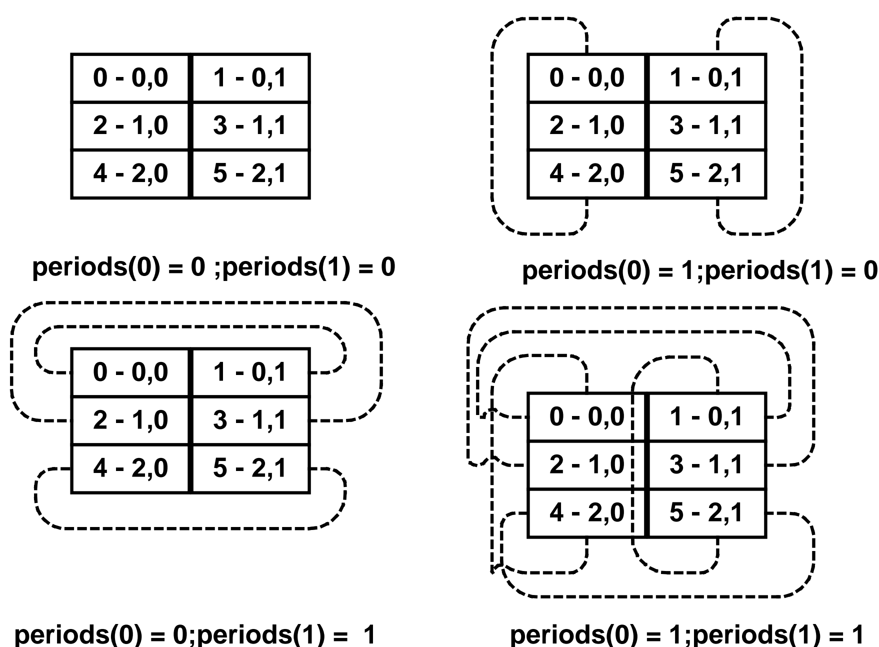
```
int MPI_Cart_create( MPI_Comm oldcomm, int ndims, int
*dims, int *periods, int reorder, MPI_Comm *cart_comm )
```

| | |
|---|---|
| **oldcomm** | Komunikator pierwotny |
| **ndims** | Liczba wymiarów |
| **dims** | Tablica podająca rozmiary w kolejnych wymiarach |
| **periods** | Tablica - gdy element wynosi true stosowane jest wymiarowanie cykliczne (siatki tworzą pierscień) gdy false nie |
| **reorder** | Gdy false indeksy procesów są identyczne jak w komunikatorze wyjściowym, gdy true indeksy mogą być dostosowane do topologii. |
| **cart_comm** | Uchwyt nowego komunikatora |



PDF created with pdfFactory Pro trial version www.pdffactory.com

```
#include "mpi.h";
 MPI_Comm  old_comm, new_comm;
 int ndims, reorder, ierr;
 int dim_size[2], periods[2];

 old_comm = MPI_COMM_WORLD;
 ndims = 2;
 dim_size[0] = 3;
 dim_size[1] = 2;
 periods[0] = 1;
 periods[1] = 0;
 reorder = 1;
 MPI_Cart_create(old_comm, ndims, dim_size, periods,
                  reorder, &new_comm);
```




Rys. 1-9 Topologie kartezjańskie otwarte i cykliczne

Przeliczanie współrzędnych kartezjańskich na indeks procesu

```
int MPI_Cart_rank(MPI_Comm cart_comm, int *coords, int *rank )
```

`coords` – we, tablica zawierająca współrzedne kartezjańskie
`rank` - wy, numer procesu w numeracji liniowej



PDF created with pdfFactory Pro trial version www.pdffactory.com

## Przeliczanie indeksu procesu na współrzędne kartezjańskie

```
int MPI_Cart_coords( MPI_Comm cart_comm, int rank, int ndims,
int *coords )
```

**rank** - we, numer procesu w numeracji liniowej
**ndims** - we, liczba wymiarów
**coords** - wy, tablica zawierająca współrzedne kartezjańskie

## Wymiana komunikatów z sąsiednimi procesami

```
int MPI_Cart_shift( MPI_Comm cart_comm, int direction,
int disp, int *rank_source, int *rank_dest )
```

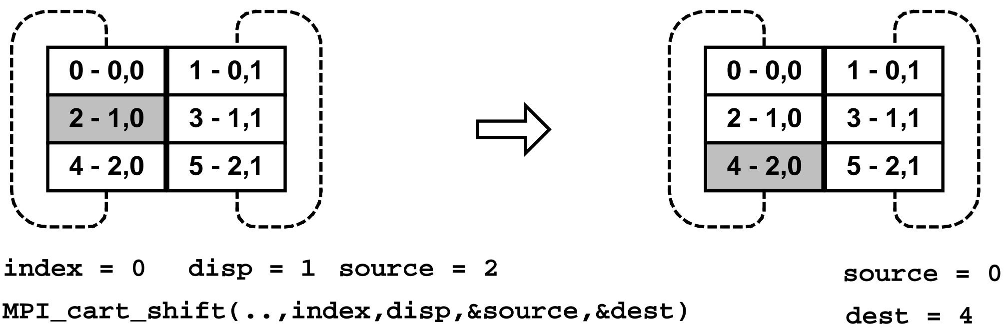
| | |
|---|---|
| **cart_comm** | Komunikator (kartezjański) |
| **direction** | Kierunek przesunięcia – należy podać indeks wymiaru |
| **disp** | Przesunięcie (gdy > 0 w kierunku większych indeksów, gdy < 0 w kierunku mniejszych indeksów) |
| **rank_source** | Numer procesu bieżącego (po wykonaniu przesunięcie w przeciwnym kierunku) |
| **rank_dest** | Numer procesu sąsiedniego |

```
#include "mpi.h";
 MPI_Comm  old_comm, new_comm;
 int ndims, reorder, ierr, source, dest;
 int dim_size[2], periods[2];

 old_comm = MPI_COMM_WORLD;
 ndims = 2;
 dim_size[0] = 3;
 dim_size[1] = 2;
 periods[0] = 1;
 periods[1] = 0;
 reorder = 1;

 MPI_Cart_create(old_comm, ndims, dim_size, periods,
                 reorder, &new_comm);

 source = me;   /* Identyfikator procesu bieżącego */
 index =  0;    /* Przesunięcie wzdłóż pierwszego indeksu  */
 displ =  1;    /* przesunięcie 0  1 */
 MPI_Cart_shift(old_comm, index, displ, &source, &dest);
```

Przykład 14 Przejście do następnego procesu wzdłuż wymiaru 0



PDF created with pdfFactory Pro trial version www.pdffactory.com

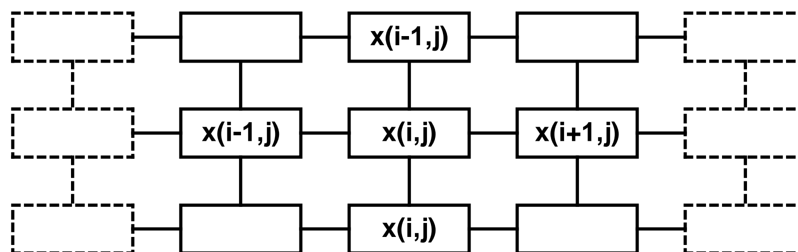



Rys. 1-10 Przejście do następnego procesu wzdłuż wymiaru 0



PDF created with pdfFactory Pro trial version www.pdffactory.com

Zastosowanie – metoda elementów skończonych




Rys. 1-11 Zastosowanie topologii kartezjańskich w metodzie elementów skończonych



PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.12 Tworzenie procesów

Specyfikacja MPI 2.0 wprowadziła możliwość tworzenia procesów.

**MPI_Comm_spawn** - Tworzenie nowych procesów

```
int MPI_Comm_spawn( char *program,char *argv[], int
maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm
*intercomm, int errors[] )
```

| | |
|---|---|
| **program** | Nazwa programu |
| **argv** | Tablica z argumentami |
| **maxprocs** | Liczba kopii procesu |
| **info** | Uchwyt do obiektu z dodatkowymi informacjami |
| **root** | Numer procesu głównego, który rozsyła dane |
| **comm** | Identyfikator komunikatora |
| **intercomm** | Uchwyt komunikatora zewnętrznego |
| **errors** | Tablica z kodami błędów |

Funkcja próbuje utworzyć **maxprocs** identycznych kopii procesów potomnych z pliku wykonywalnego o nazwie **program**.
Funkcja powinna być wykonana grupowo, ale tylko proces o numerze **root** tworzy nowe procesy i tylko w nim wykorzystane będą parametry **program, argv, maxprocs, info.**

Funkcja tworzy nowy komunikator **intercomm**, który zawiera tak stare procesy z komunikatora **comm** (jako grupa lokalna) jak i nowo utworzone procesy (jako grupa zdalna). Nowo utworzone procesy otrzymują własny komunikator **MPI_COMM_WORLD**.

Argument info zawiera uchwyt do obiektu zawierającego pary **nazwa=wartość**.

Aby nowe procesy mogły się komunikować procesy potomne muszą znać komunikator procesów macierzystych. Może on być uzyskany za pomocą funkcji **MPI_Comm_get_parent**.

**MPI_Comm_get_parent - Tworzenie nowych procesów**

**Int MPI_Comm_get_parent( MPI_Comm *comm)**

**comm** Identyfikator komunikatora macierzystego



PDF created with pdfFactory Pro trial version www.pdffactory.com

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

#define NUM_SPAWNS 2

int main( int argc, char *argv[] ) {
    int np = NUM_SPAWNS;
    int errcodes[NUM_SPAWNS];
    MPI_Comm parentcomm, intercomm;

    MPI_Init( &argc, &argv );
    MPI_comm_get_parent( &parentcomm );
    if (parentcomm == MPI_COMM_NULL)  {
       MPI__Com_spawn( "spawn_example", MPI_ARGV_NULL, np,
       MPI_INFO_NULL, 0, MPI_COMM_WORLD,
       &intercomm, errcodes );
       printf("Proces macierzysty\n");
    } else {
        printf("Nowy proces\n");
    }
    fflush(stdout);
    MPI_Finalize();
    return 0;
}
```

Przykład 15 Tworzenie 2 procesów potomnych

## 1.13 Własności systemów MPI

Zalety:

- Efektywność na komputerach z pamięcią lokalną (multikomputery)
- Podjęcie problemu równoległego wejścia wyjścia

Wady:

- Nie można łączyć różnych implementacji
- Nie można łączyć aplikacji pisanych w różnych językach programowania.
- Brak komunikacji pomiędzy niezależnie uruchamianymi procesami



PDF created with pdfFactory Pro trial version www.pdffactory.com