Kolejne funkcje MPI

Przypomnienie: sieci typu mesh (Grama i wsp.)

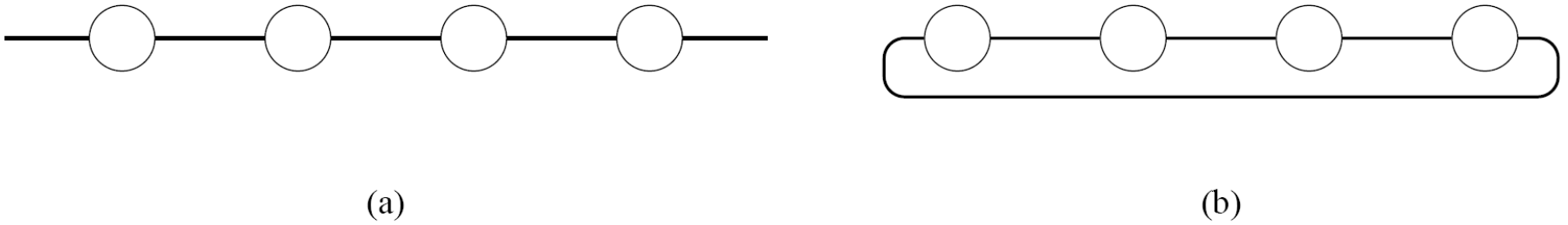


Figure 2.15 Linear arrays: (a) with no wraparound links; (b) with wraparound link.

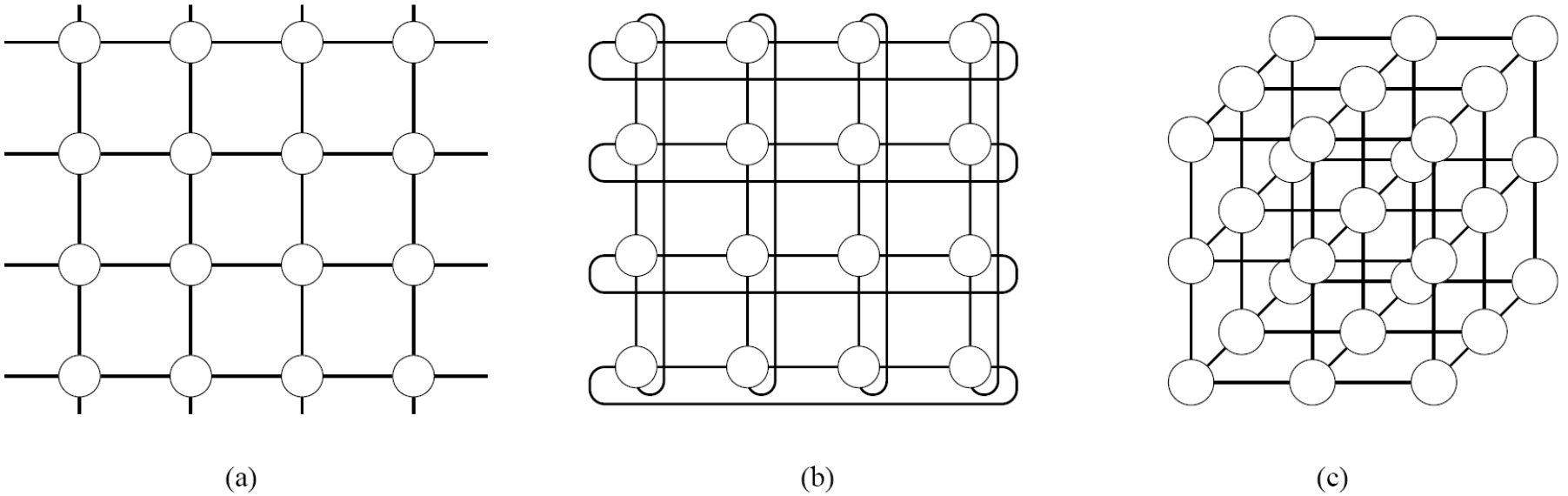


Figure 2.16 Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

Topologie MPI

- Standard MPI abstrahuje od topologii systemu wieloprocesorowego. Sposób uruchamiania aplikacji (mpirun) nie jest częścią standardu. Mamy niewielką kontrolę nad tym gdzie proces zostanie uruchomiony.
- Załóżmy, że system równoległy z pamięcią rozproszoną ma architekturę typu mesh. Pożądane jest, aby procesy często komunikujące się ze sobą były umieszczone w sąsiednich procesorach. Niestety w standardzie MPI brak funkcji „umieść proces w procesorze”.
- MPI dostarcza w zamian mechanim pozwalający zorganizować grupę procesów w komunikator o logicznej topologii mesh. Przyjęto założenie, że implementacja dopasowana do specyficznej architektury zorganizuje procesy tak, ***aby procesy uruchomione na sąsiednich procesorach były sąsiadami w logicznej topologii***
- Warto dodać że struktura procesów tworząca grafy typu mesh występuje w wielu zastosowaniach, zwłaszcza w symulacji.
- Ponadto niektóre architektury sprzętowe (np. Cray) posiadają również strukturę typu mesh
- Topologie typu mesh nazywane są topologiami kartezjańskimi lub siatkami.

Funkcja `MPI_Cart_create`

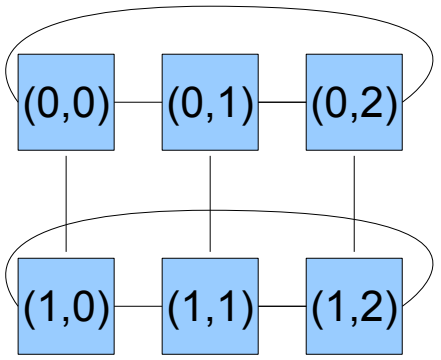
```
int MPI_Cart_create ( MPI_Comm comm_old, int ndims, int *dims, int
*periods, int reorder, MPI_Comm *comm_cart )
```

- Utworzenie nowego komunikatora (zapamiętywany w parametrze `comm_cart`) o topologii kartezjańskiej.
- `ndims` jest wymiarem topologii kartezjańskiej (1 – linia; 2 – torus lub mesh; 3 - siatka trójwymiarowa).
- `dims` tablica o długości równej wymiarowi specyfikująca liczbę procesów w każdym wymiarze.
- `periods` tablica o długości równej wymiarowi specyfikująca czy siatka zawiera „zawinięcia” w każdym wymiarze.
- `reorder` pozwolenie na zmianę rang w nowym komunikatorze.

Przykład

```
MPI_Comm cart;
// trzy procesy w pierwszym wymierze dwa w drugim
int dims[2]={3,2};
// zawinięcie tylko w pierwszym wymiarze
int periods[2]={1,0};
MPI_Cart_create (MPI_COMM_WORLD,2,dims,periods,1,&cart);
```

- Zostanie stworzona następująca logiczna topologia procesów (pierwszy wymiar to oś x)



- W nawiasach współrzędne procesów w topologii.
- Cały trik polega na tym, że w nowym komunikatorze (`cart`) procesy leżące blisko siebie powinny być zlokalizowane w sąsiednich procesorach (ale to zależy od implementacji MPI)
- Dalsze obliczenia należy prowadzić przy pomocy komunikatora `cart` a nie `MPI_COMM_WORLD`.

Jak zlokalizować sąsiadów ? (1)

```
int MPI_Cart_coords ( MPI_Comm comm, int rank, int maxdims, int
   *coords )
```

- `rank` jest numerem procesu (uzyskiwany przez `MPI_Comm_rank`).
- `maxdims` jest rozmiarem tablicy `coords`.
- Funkcja zapisuje do tablicy coords współrzędne procesu o numerze rank.

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

- Funkcja zapisuje pod adresem `rank` numer procesu o współrzędnych określonych w tablicy `coords`.
- Bardzo często chcemy znaleźć odpowiedź na inne pytanie:

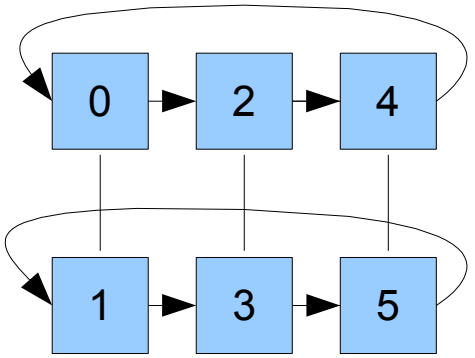
Jaki jest numer sąsiada z lewej, prawej, góry, dołu ?

Jak zlokalizować sąsiadów ? (2)

```
int MPI_Cart_shift ( MPI_Comm comm, int direction, int displ, int
   *source, int *dest )
```

- Funkcja oblicza numer sąsiada numer procesu źródłowego przesuniącia (zapisany pod adresem `source`) oraz procesu docelowego przesunięcia (zapisany pod adresem `dest`).
- Przesuniącie wykonywane jest względem bieżącego procesu, w wymiarze o numerze `direction` o wielkość `displ`.
- Wielkość `displ>0` oznacza przesunięcie w stronę rosnących współrzędnych a wartość `displ<0` w stronę malejących współrzędnych.

Przykład z `MPI_Cart_shift`



- Cyfry oznaczają przykładowe numery (rangi) procesów.
- Wywołanie funkcji jest następujące:

```
int source,dest;
MPI_Cart_shift ( MPI_COMM_WORLD,0,1,&source,&dest);
```

- Przesunięcie po zerowym wymiarze (oś x) w stronę rosnących współrzędnych o 1.
- Dla procesu o numerze 3 source=1 dest=5.
- Dla procesu o numerze 4 source=2 dest=0.

Pakowanie danych i definiowanie typów

- Do tej pory wysyłaliśmy i odbieraliśmy wektory identycznych typów. Problem pojawia się, gdy komunikat do wysłania nie ma takiej jednorodnej struktury.
- Na przykład chcemy wysłać komunikat składający się z dwóch liczb typu int i jednej liczby typu long i stu liczb typu double.
- MPI dostarcza dwa mechanizmy do osiągnięcia tego celu.
- Możemy upakować dane do bufora zdefiniowanego przez użytkownika a następnie wysłać spakowany bufor używając typu dannych `MPI_PACKED`.
  - rozwiązanie to jest w miarę wygodne, ale prowadzi do dodatkowego kopiowania danych do bufora, co obniża wydajność.
- Alternatywnie możemy zdefiniować nowy typ danych (w tym przypadku strukturę) i posługiwać się dalej tym typem jako argumentem funkcji wysyłających i odbierających komunikaty..

Funkcja MPI_Pack

```
int MPI_Pack ( void *inbuf, int incount, MPI_Datatype datatype, void
   *outbuf, int outcount, int *position, MPI_Comm comm )
```

- Funkcja pozwala na inkrementalne dodawanie danych do bufora zdefiniowanego przez użytkownika.
- Pakuje do bufora `outbuf`, (o rozmiarze `outcount`) poczynając od pozycji przechowywanej w zmiennej pod adresem `position`, `incount` elementów typu `datatype` przechowywanych pod adresem `inbuf`.
- Zmienna o adresie `position` jest odpowiednio przesuwana w wyniku dodania elementów do bufora, tak aby wskazywała "pierwsze wolne miejsce".
- `position` jest jednocześnie argumentem wyjściowym i wejściowym.
- Po spakowaniu, przy wysyłaniu bufora należy użyć typu danych `MPI_Packed`.

Funkcja MPI_Pack_size

- Stosując pakowanie danych chcemy znaleźć odpowiedź na pytanie: Jak dużego bufora potrzebuję. Należy to robić przy pomocy funkcji

```
int MPI_Pack_size (int incount, MPI_Datatype datatype, MPI_Comm
   comm,int *size )
```

- Funkcja ta zapisuje pod adresem size liczbę bajtów niezbędną do upakowania incount elementów typu datatype.
- Zauważmy, że zarówno `MPI_Pack` jak i `MPI_Pack_size` wymagają podania komunikatora (parametr comm), pomimo że nie wykonują żadnej komunikacji.
  - Jest to uzasadnione ponieważ w środowisku heterogenicznym (np. klaster zbudowany z maszyn na procesorach Intela i Motoroli) możliwa różna reprezentacja danych (np. int 32-bitowy, 64-bitowy) na różnych komunikatorach

Przykład - MPI_Pack

```
struct Data {
   int x,y;
   double tab[100];
};

void PackAndSend(Data *pD) {
   int Size1,Size2,Size;
   // Oblicz rozmiar bufora
   MPI_Pack_size(2,MPI_INT,MPI_COMM_WORLD,&Size1);
   MPI_Pack_size(100,MPI_DOUBLE,MPI_COMM_WORLD,&Size2);
   Size=Size1+Size2;
   char *pBuf=new char [Size];
   int position=0;
   // Spakuj dane. Dwie zmienne int trzeba pakować osobno !!! (dlaczego ?)
   MPI_Pack(&(pD->x),1,MPI_INT,pBuf,Size,&position,MPI_COMM_WORLD);
   MPI_Pack(&(pD->y),1,MPI_INT,pBuf,Size,&position,MPI_COMM_WORLD);
   // Teraz 100 liczb typu double
   MPI_Pack(&(pD->tab),100,MPI_DOUBLE,pBuf,Size,&position,MPI_COMM_WORLD);
   // Wyślij spakowane dane.
   MPI_Send(pBuf,position,MPI_PACKED,0,0,MPI_COMM_WORLD);
```

- x oraz y pakujemy osobno ponieważ kompilator może wstawić wolne miejsce pomiędzy polami we strukturze.

Funkcja MPI_Unpack

```
int MPI_Unpack ( void *inbuf, int insize, int *position,void *outbuf,
   int outcount, MPI_Datatype datatype,MPI_Comm comm )
```

- Jest odwrotnością funkcji MPI_Pack
- Funkcja rozpakowuje z bufora `inbuf` o długości insize od pozycji `position` outcount elementów typu datatype umieszczając je w buforze outbuf.
- Zmienna o adresie `position` jest odpowiednio przesuwana w wyniku wypakowania elementów z bufora inbuf, tak aby wskazywała kolejny element.
- `position` jest jednocześnie argumentem wyjściowym i wejściowym.
- Bufor inbuf musi być wcześniej odebrany przy pomocy typu `MPI_Packed`.

Przykład - MPI_Unpack

```
struct Data {
   int x,y;
   double tab[100];
};

void ReceiveandUnpack(Data *pD) {
   int Size1,Size2,Size;
   // Oblicz rozmiar bufora
   MPI_Pack_size(2,MPI_INT,MPI_COMM_WORLD,&Size1);
   MPI_Pack_size(100,MPI_DOUBLE,MPI_COMM_WORLD,&Size2);
   Size=Size1+Size2;
   char *pBuf=new char [Size];
   // Odbierz spakowane dane
   MPI_Recv(pBuf,Size,MPI_PACKED,1,0,MPI_COMM_WORLD);
   int position=0;
   // Rozpakuj dane
   MPI_Unpack(pBuf,Size,&position,&(pD->x),1,MPI_INT,MPI_COMM_WORLD);
   MPI_Unpack(pBuf,Size,&position,&(pD->y),1,MPI_INT,MPI_COMM_WORLD);
   MPI_Unpack(pBuf,Size,&position,&(pD->tab),100,MPI_DOUBLE,
              MPI_COMM_WORLD);
   // Można korzystać z danych pod adresem pD
```

- Pakowanie/rozpakowywanie prowadzi do straty wydajności z powodu dodatkowego kopiowania danych ale wydajność i tak jest znacznie lepsza niż gdybyśmy musieli przesłać 3 komunikaty.

Definiowanie typów

- Standard MPI posiada bardzo złożony aparat do definiowania typów. Na wykładzie zostanie szczegółowo opisany tylko jego niewielki fragment służący do definiowania typu struktury. Służy do tego funkcja:

```
int MPI_Type_struct(int count,int blocklens[], MPI_Aint indices[],
   MPI_Datatype old_types[], MPI_Datatype *newtype )
```

- Tablice `blocklens`, `indices`, `old_types` powinny mieć długość równą liczbie pól w strukturze.
  - `blocklens[i]` jest liczbę elementów typu i-tego pola (większa od jeden gdy jest to tablica)
  - `indices[i]` jest przesunięciem pola i od początku struktury
  - `old_types[i]` jest typem pola i.
- Nowo stworzony typ jest zapamiętywany pod adresem newtype

MPI_Type_commit/MPI_Type_free oraz MPI_Address

- Przed wykorzystaniem w operacjach send/receive nowo stworzony typ rejestrujemy funkcją:

```
int MPI_Type_commit( MPI_Datatype *datatype )
```

- Gdy typ jest już nam zbędny, należy go wyrejestrować funkcją:

```
int MPI_Type_free( MPI_Datatype *datatype )
```

- Adress argumentu `location` oblicza funkcja:

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

  i zapisuje go pod adresem `address`. Funkcja ta istnieje w standardzie MPI ze względu na język Fortran, który nie posiada adresów. W C/C++ może być zastąpiona operatorem &.

Definiowanie typów: przykład (1)

```
struct Data {
   int x,y;
   double tab[100];
};
// Długości trzech pól
int blocklens[3]={1,1,100};
// Typy trzech pól
int types[3]={MPI_INT,MPI_INT,MPI_DOUBLE};
MPI_Aint indices[3]
MPI_DATATYPE newtype;

void DefineType(Data *pD) {
   // wypełnij tabelę indices najpierw adresami
   MPI_Address(&(pD->x),indices)
   MPI_Address(&(pD->y),indices+1)
   MPI_Address(&(pD->tab),indices+2)
   // Przelicz adresy na przesunięcia od początku struktury.
   // Uwaga na kierunek pętli !!!
   for(int i=2;i>=0;i--)
      indices[i]-=indices[0];
   // Stwórz i zarejestruj typ
   MPI_Type_struct(3,blocklens,indices,types,&newtype)
   MPI_Type_commit(&newtype);
}
```

Definiowanie typów: przykład (2)

```
void SendStruct(Data *pD) {
   MPI_Send(&(pD->x),1,newtype,0,0,MPI_COMM_WORLD)
}
```

- Jako typ do operacji Send podaję zmienną `newtype` przechowującą nowo zdefiniowany typ MPI.
- Jako adres bufora podany został adres pierwszego elementu,a nie adres struktury. Jest to bardzo ważne ponieważ przesunięcia były liczone od pierwszego elementu (pętla for), a pierwszy element może być przesunięty względem początku struktury (kompilator stosuje tzw. padding).
- Możemy użyć dowolnych operacji MPI dowolną liczbę razy z nowym typem.
- Jeżeli typ stanie się zbędny, powinniśmy go wyrejestrować przy pomocy `MPI_Type_free`.

Trwałe żądania MPI_Request

- Przypomnienie: operacje nieblokujące MPI_Isend/MPI_Irecv rozpoczynają operację transmisji danych i tworzą obiekt żądania typu MPI_Request. Na tym obiekcie wykonywana jest operacji MPI_Wait czekająca na skompletowanie operacji lub operacja MPI_Test testująca skompletowanie.
- W wielu zastosowaniach takich operacji są tysiące a nawet miliony. Jeżeli za każdym razem bufor, typ danych, nadawca i odbiorca są takie same (np. funkcja jest wywoływana w pętli), to niepotrzebnie ponosimy koszty związane z tworzeniem (MPI_Isend/MPI_Irecv) oraz ze zwolnieniem (np. MPI_Wait) obiektu typu MPI_Request.
- Bardzo zaawansowana implementacja MPI powinna być w stanie skorzystać z faktu, że takie same operacje wykonywane są wielokrotnie.
- W tym celu MPI udostępnia mechanizm stworzenia trwałego (ang. persistent) żądania MPI_Request i wielokrotnego wywoływania (startowania) tego żądania.

MPI_Send_init/MPI_Recv_init

```
int MPI_Recv_init( void *buf, int count, MPI_Datatype datatype, int
   source,int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Send_init( void *buf, int count, MPI_Datatype datatype, int
   dest,int tag, MPI_Comm comm, MPI_Request *request);
```

- Funkcja `MPI_Send_init` tworzy trwałe żądanie wysłania danych a funkcja MPI_Recv_init tworzy trwałe żądanie odbioru danych.
- W odróżnieniu od `MPI_Isend`/`MPI_Irecv` funkcje te nie rozpoczynają transmisji, a jedynie tworzą ***nieaktywny*** obiekt `MPI_Request`.
- Rozpoczęcie nadawania (`MPI_Send_init`) albo odbioru (`MPI_Recv_init`) następuje z chwilą wywołania funkcji

```
int MPI_Start(MPI_Request *request);
```

- Istnieje również funkcja

```
int MPI_Startall( int count, MPI_Request array_of_requests[]);
```

  rozpoczynająca `count` trwałych żądań zgromadzonych w tablicy `array_of_requests`.

- Gdy operacja się zakończy (np. po wywołaniu `MPI_Wait`) możemy ponownie wywołać funkcję `MPI_Start`.

MPI_Request_free

- W odróżnieniu od obiektów stworzonych przez funkcje MPI_Isend/MPI_Irecv obiekty trwałych żądań nie są zwalniane przez funkcje MPI_Wait/MPI_Test.
- Kiedy trwały obiekt żądania jest niepotrzebny, należy go zwolnić funkcję

```
int MPI_Request_free( MPI_Request *request )
```

- W ten sposób operacja `MPI_Irecv(.....,&request)` jest równoważna:

```
MPI_Recv_init(.....,&request);
MPI_Start(&request);
```

a operacja MPI_Wait na obiekcie request stworzonym przez MPI_Irecv jest równoważna:

```
MPI_Wait(&request,&status);
MPI_Request_free(&request);
```

- Korzystając z trwałych żądań nie ponosimy kosztów ich utworzenia i usunięcia.

Buforowana operacja send

- Objaśniając funkcję `MPI_Send`, tłumaczyłem że może ona (ale nie musi) wstrzymać proces nadawcy do momentu wywołania przez odbiorcę operacji `MPI_Recv`.
  - W praktyce komunikaty o długości mniejszej o długości progowej są buforowane i dla nich `MPI_Send` nie czeka na `MPI_Recv`.
- Standard MPI zawiera również operację `MPI_Bsend`, o identycznej składni jak operacja `MPI_Send`, która zawsze korzysta z bufora i w związku z tym powraca po skopiowaniu do niego danych.
- Bufor do operacji MPI_Bsend powinien być dostarczony przez użytkownika, funkcją:

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

  przy czym bufor powinien być dostatecznie duży, aby pomieścić wszystkie komunikaty, które muszą być wysłane, zanim operacje Recv zostaną wywołana przez odbiorców

- Dodatkowo w buforze należy zarezerwować `MPI_BSEND_OVERHEAD` bajtów na każdą operację Send.
- Po wykorzystaniu bufor zwalniamy funkcją

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

Synchroniczna operacja send

- W jaki sposób przekonać się że nie napisałyśmy „niebezpiecznego” (ang. unsafe) - tzn. mogącego wpaść w blokadę programu, którego poprawność zależy od buforowania funkcji `MPI_Send` ?
- Należy zastąpić wszystkie wywołania `MPI_Send` przez `MPI_Ssend`.
- Funkcja `MPI_Ssend` realizuje synchroniczną operację wysłania danych. Czeka ona z powrotem do momentu gdy odbiorca rozpoczyna odbiór komunikatu (poprzez wywołanie jednej z funkcji Recv).
- Jeżeli program jest „niebezpieczny” to zastąpienie `MPI_Send` przez `MPI_Ssend` ***na pewno*** doprowadzi do blokady.
- Standard MPI pozwala na to, aby `MPI_Send` zostało zaimplementowane przez `MPI_Ssend`. Pozwala również aby `MPI_Send` stosowało buforowanie dla wszystkich komunikatów.
  - Aby program był przenośny na różne implementacje MPI, powinien być tak napisany, aby działał gdy `MPI_Send` jest zaimplementowane przez `MPI_Ssend`.

MPI i wątki

- Często węzłami systemu wieloprocesorowego z przesyłaniem komunikatów są maszyny wieloprocesorowe ze wspólną pamięcią. Na przykład klaster Mordor składa się z 16 maszyn dwuprocesorowych.
- W takiej sytuacji możemy spróbować zrównoleglenia ***hybrydowego***: na każdej maszynie klastra uruchomić jeden proces komunikujący się z innymi procesami na innych maszynach. Dodatkowo ten proces możemy podzielić (za pomocą POSIX threads lub OpenMP) na dwa równoległe wątki komunikujące się poprzez wspólną pamięć.
- Takie hierarchiczne zrównoleglenie może prowadzić do większej wydajności niż traktowanie całego klastra jako zbioru procesorów wymieniających się komunikatami przez MPI.
- Niestety, nie każda implementacji MPI musi być bezpieczna dla wielowątkowości (ang. thread-safe). Jeżeli dana implementacja nie jest bezpieczna, to funkcje MPI możemy wywoływać tylko z jednego wątku.
  - Nie można współbieżnie wywoływać funkcji MPI z kilku wątków.
- MPI-2 wyposażono w specjalne funkcje pozwalające między innymi odpytać bibliotekę o wsparcie dla wielowątkowości.

Jedna rzecz o której powinienem powiedzieć na początku - MPI_Initialized

- Funkcja

```
int MPI_Initialized( int *flag )
```

  jest jedyną funkcją którą możemy wywołać przed `MPI_Init`.

- Sprawdza ona czy podsystem MPI został poprawnie zainicjalizowany poprzez `MPI_Init`, i jeżeli tak wpisuje pod adresem flag wartość 1.
- W przeciwnym wypadku wpisuje pod adresem flag wartość 0.

MPI - podsumowanie

- To wszystko na tym wykładzie jeżeli chodzi o standard MPI-1
- Pominęliśmy
  - Zaawansowane tworzenie typów.
  - Operacja na komunikatorach (przydatne w pisaniu bibliotek równoległych i interkomunikatorach)
  - Gotowe (ang. ready mode) operacje trasnmisji.
- O ile czas pozwoli, zostaną pokazane najciekawsze „nowości” standardu MPI-2
  - Plikowe wejście-wyjście (MPI-IO)
  - Jednostronne operacje komunikacyjne (ang. one-sided communication).
  - Dynamiczne tworzenie procesów.
- Obecnie MPI jest niekwestionowanym standardem jeżeli chodzi o programowanie systemów z przesyłaniem komunikatów.
- W oparciu o standard MPI piszecie Państwo projekt.