Programowanie równoległe MPI

Maciej Szpindler

m.szpindler@icm.edu.pl

Interdyscyplinarne Centrum Modelowania Matematycznego i Komputerowego

Uniwersytet Warszawski

http://www.icm.edu.pl

Podziękowania

- Pomysły i przykłady zaczerpnięte z materiałów szkoleniowych HLRS,
  - Autor: Rolf Rabenseifner
  - https://fs.hlrs.de/projects/par/par_prog_ws/
- Materiały ze szkoleń ICM UW
  - Autorzy: Maciej Cytowski, Maciej Marchwiany
- Wykorzystane materiały z dawnego Biuletynu KDM
  - Autorzy: zespół KDM ICM UW

Zakres materiału

EURO

- Wstęp
- Message Passing Interface – podstawy
- Komunikacja point-to-point
- Komunikacja grupowa
- Komunikacja nieblokująca
- Typy danych MPI
- Konstrukcja wirtualnych topologii w MPI
- MPI-2 oraz MPI-3 – przegląd
- Komunikacja jednostronna
- Podsumowanie
- Opis zadań do ćwiczeń

Wstęp: różne pojęcia równoległości

EURO

- Różne podejścia do sprzętowego zrównoleglenia przetwarzania
  - Wektoryzacja (pipelining) na poziomie procesora
  - Duplikacja jednostek przetwarzania np. FPU na poziomie procesora
  - Wiele wątków (multi/hyper-threading) na poziomie procesora
  - Wiele rdzeni (multi-core) na poziomie procesora
  - Wiele procesorów (ccNUMA*) na poziomie systemu (pojedynczego komputera)
  - Bardzo wiele procesorów (klastry serwerów, MPP**)
- Implikacje
  - Wielowątkowość/procesorowość, wspólny dostęp do pamięci --programowanie w modelu pamięci współdzielonej
  - Klastry, sieć procesorów, pamięć rozproszona -- programowanie w modelu pamięci rozproszonej

*cache coherent Non-Uniform Memory Access
** Masively Parallel Processing

Pamięć współdzielona i rozproszona

EURO

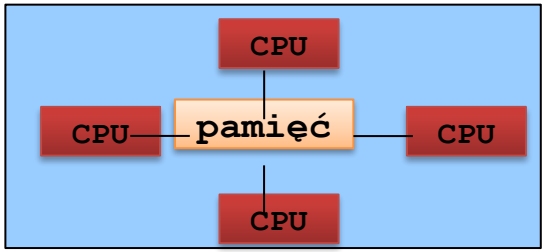
- Poszczególne procesory (węzły) posiadają prywatną pamięć i są połączone siecią
- Komunikacja odbywa się poprzez jawną wymianę danych przez sieć

- Dziś większość klastrów posiada węzły wielo-procesorowe/rdzeniowe (ccNUMA)
- Dwa typy równoległości:
  - rozproszona – między węzłami sieci (*czytaj np. OpenMP*),
  - współdzielona – w ramach jednego węzła (*czytaj np. MPI*)

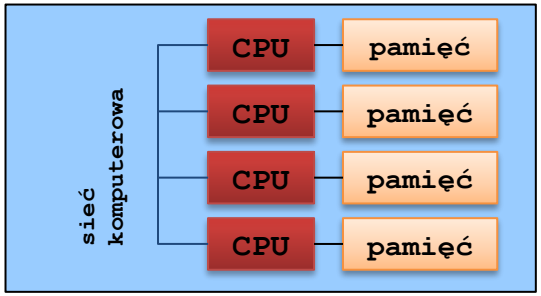
Modele programowania równoległego

EURO

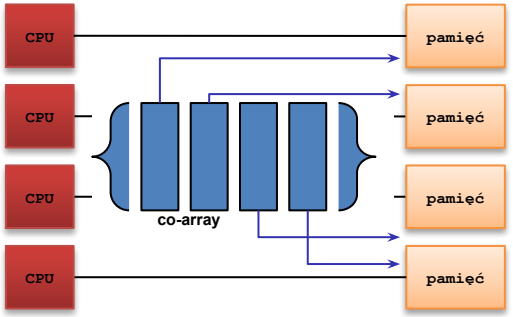
- **Pamięć współdzielona (Shared Memory)**
  - OpenMP
    - Dyrektywy kompilatora
    - Standard
  - Biblioteki wielowątkowe – programowanie z użyciem wątków
    - Pthreads, TBB, Shmem



- **Pamięć rozproszona (Distributed Memory)**
  - MPI – Message Passing Interface
    - Biblioteka do wymiany komunikatów przez sieć



- **PGAS (Partitioned Global Address Space)**
  - UPC
  - Co-array Fortran
  - Global Arrays
  - PCJ



Strategie zrównoleglania obliczeń

EURO

- Zrównoleglenie może dotyczyć
  - Danych – podział pracy względem danych w pamięci
  - Instrukcji procesora – podział pracy względem typu wykonywanych operacji
- Strategie dekompozycji obliczeń
  - Podział zadań pomiędzy procesory/procesy/wątki (task model)
  - Podział danych w rozproszonej pamięci (data decomposition)
- Wymaga to zapewnienia
  - Komunikacja między procesorami
  - Synchronizacji pamięci

Biblioteka MPI

EURO

- MPI - **Message Passing Interface**
  - Biblioteka zestawu procedur i funkcji zgodna ze standardem MPI
  - Standard ustala MPI-Forum
  - Pracuje natywnie z C, C++, Fortranem
  - Faktyczny standard programowania w oparciu o wymianę komunikatów
    - MPI-1 (Wersje 1.1, 1.2) – wprowadza około 150 funkcji
    - MPI-2 (Wersje 2.1, 2.2) – nowa funkcjonalność, razem ponad 500 funkcji
    - MPI-3 (Wersje 3.0, 3.1)
- Najnowsza wersja standardu 4.0 (2021):
  - https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
- Różne implementacje (ABI non-compatible)
  - MPICH – http://www.mpich.org
  - OpenMPI – http://www.open-mpi.org
  - Inne bazują w większości na powyższych
- Przenaszalność między implementacjami
  - https://github.com/eschnett/MPItrampoline

Czy to jest przestarzałe ?!

Podstawy: model równoległości

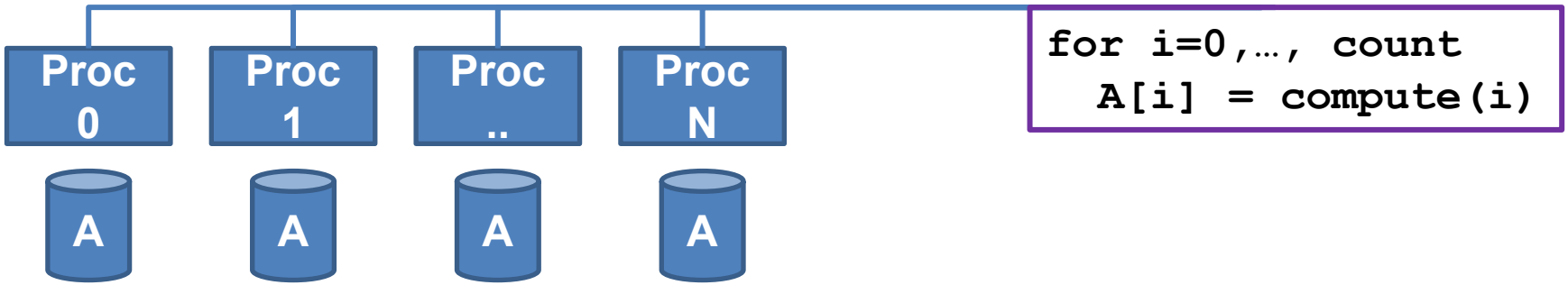
EURO

- Założenia
  - **Pamięć jest rozproszona, prywatna dla każdego procesu**
  - **Procesy wymieniają się komunikatami**
  - **Jawna równoległość**
- Model procesowy
  - Proces* odpowiada fizycznej jednostce obliczeniowej
  - Program składa się z procesów – podprogramów działających na poszczególnych procesorach
- Typowo model SPMD – **Single Program Multiple Data**
  - Ten sam podprogram pracuje na każdym procesorze
  - Napisany w tradycyjnym języku + wywołania biblioteki MPI
- Model MPMD mniej popularny, ale również wspierany

*Wikipedia: In computing, a **process** is an instance of a computer program that is being executed. It contains the program code and its current activity.

Podstawy: model SPMD

- SPMD – Single Program Multiple Data
  - Model zakładający dekompozycje danych
  - Każdy proces (podprogram) wykonują identyczny zestaw operacji ale na innym zbiorze danych
  - Wymaga dekompozycji problemu na szereg identycznych podproblemów
- Najpopularniejszy model zrównoleglania dla MPI
- Różnice terminologiczne:
  - Task (SLURM), Worker (Matlab),



Podstawy: komunikator

EURO

- Podstawowa struktura programu równoległego używającego MPI
- **Komunikator** – abstrakcja opisująca kontekst oraz zbiór (grupę) procesów mogących się wzajemnie komunikować
- W ramach komunikatora każdy proces należy do grupy z unikalnym identyfikatorem (numerem) tzw. **rank**
- Informacja o komunikatorze zawarta jest w zmiennej typu **`MPI_COMM`**
- W czasie inicjacji programu biblioteka tworzy domyślny komunikator o nazwie **`MPI_COMM_WORLD`** zawierający wszystkie dostępne procesy oraz **`MPI_COMM_SELF`** zawierający tylko lokalny proces
- Programista może definiować własne komunikatory zawierające podzbiory procesów

Podstawy: grupa procesów

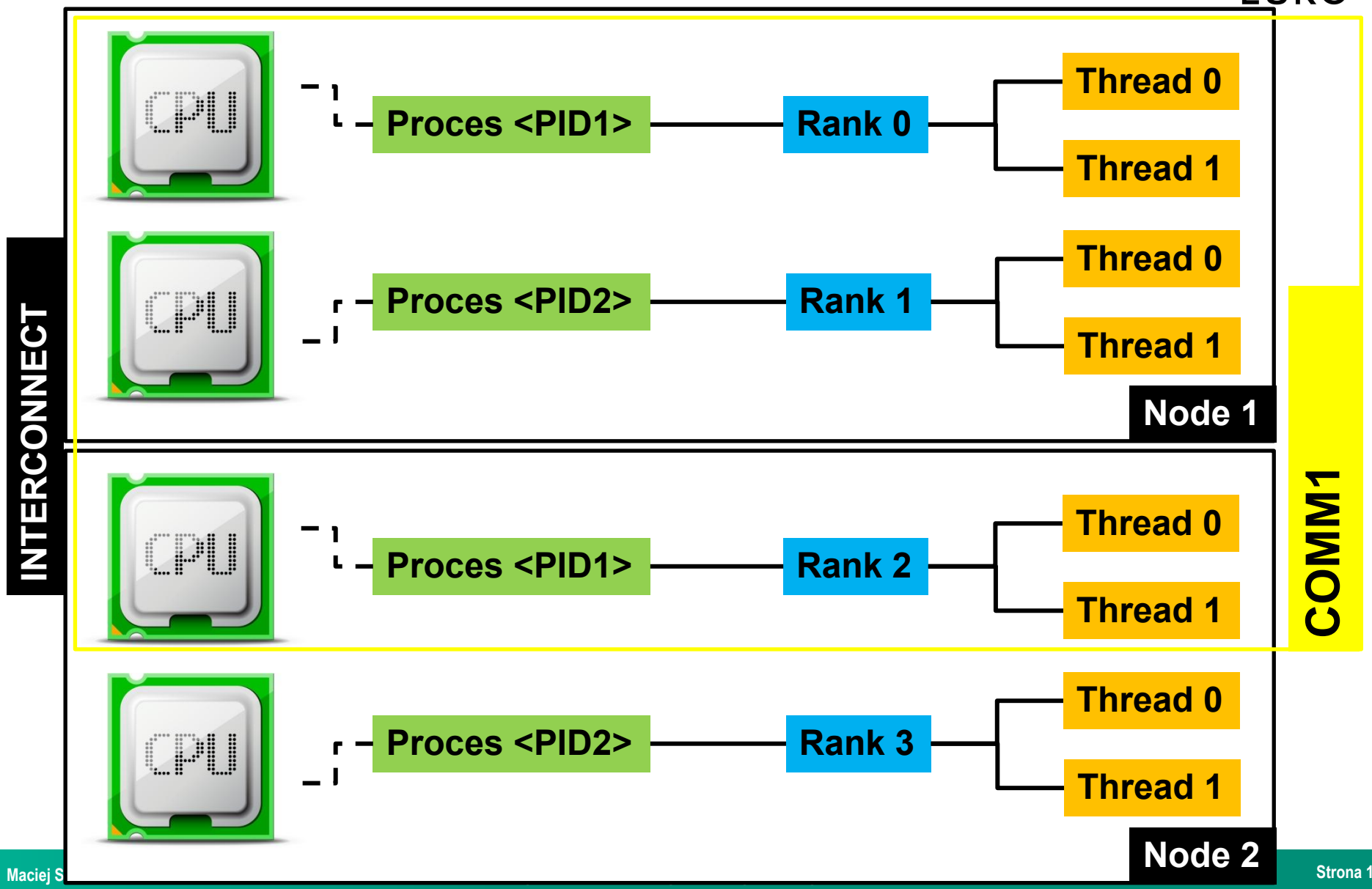
EURO

Grupa definiuje uporządkowany zbiór procesów (każdy posiadający unikalny numer - rank). Grupy definiują zakres procesów zaangażowanych w operacje komunikacyjne. Grupami można zarządzać niezależnie od komunikatorów MPI, ale tylko komunikatory pozwalają na wykonanie komunikacji.

Zarządzanie grupami, przykłady operacji:

- `MPI_COMM_GROUP`
- `MPI_COMM_CREATE_FROM_GROUP`

Podstawy: schemat graficzny



Pobieranie informacji z komunikatora

EURO

- Liczba (np) dostępnych procesów w ramach komunikatora

  `MPI_Comm_size(MPI_COMM_WORLD, &np)`

- Numer (rank) mojego procesu

  `MPI_Comm_rank(MPI_COMM_WORLD,&rank)`
  - (Uwaga Fortran) rank = 0, ..., np – 1

C:

```
int MPI_Comm_size ( MPI_Comm comm, int *np );
int MPI_Comm_rank ( MPI_Comm comm, int *rank );
```

Fortran:

```
CALL MPI_COMM_SIZE ( integer comm, integer np, integer ierror )
CALL MPI_COMM_RANK ( integer comm, integer rank, integer ierror )
```

Startowanie i zamykanie sesji

EURO

- Wywołania wszelkich funkcji i procedur biblioteki MPI muszą znajdować się pomiędzy instrukcjami:

  `MPI_Init(&argc, &argv)`
  - Inicjuje działanie biblioteki i domyślny komunikator (sesję)

  `MPI_Finalize(void)`
  - Kończy sesję (program) MPI

- Funkcje **`Init`** i **`Finalize`** muszą być wywołane przez wszystkie procesy w ramach sesji

C:
```
int MPI_Init ( int *argc, int **argv  );
int MPI_Finalize ( );
```

Fortran:
```
CALL MPI_INIT( integer ierror )
CALL MPI_FINALIZE ( integer ierror )
```

Message Passing Interface - komunikaty

EURO

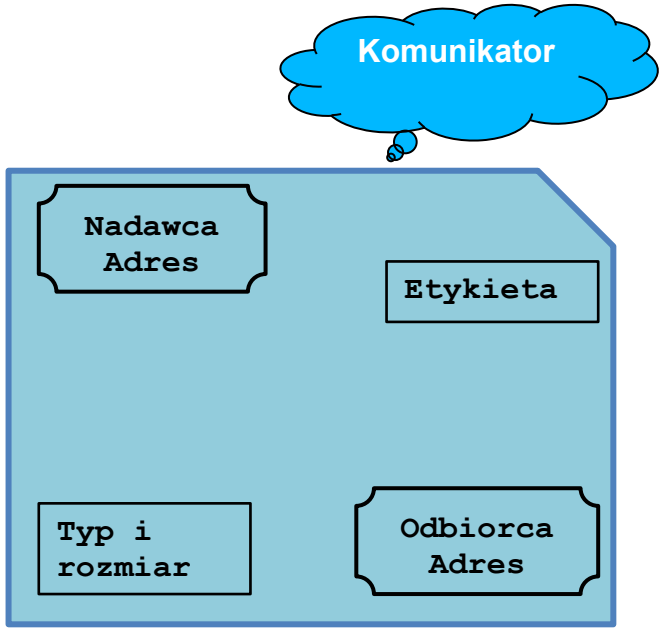
- Komunikaty
- Podstawowe typy danych w MPI
- Przesyłanie wiadomości

Komunikaty

- Komunikat – paczka danych przesyłana pomiędzy procesami przez łącze / sieć łącząca procesory w ramach komunikatora

- Paczka składa się z ustalonej liczby elementów określonego typu

- Aby przesłać paczkę należy zdefiniować:
  - Identyfikator (**rank**) procesu wysyłającego
  - Identyfikator (**rank**) procesu odbierającego
  - Adres źródłowy i adres docelowy
  - Typ i wielkość danych przesyłanych w paczce
  - Komunikator



Podstawowe typy danych

EURO

- MPI definiuje własne typy danych, które powinny być odpowiednikami typów w C
- Dodatkowo `MPI_BYTE` oraz `MPI_PACKED`

Typ MPI	Typ języka 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

Typ MPI	Typ Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

Wysyłanie wiadomości

- **buf** – adres tablicy zawierającej wiadomość długości **count**, zawierającej elementy typu **datatype**
- **dest** – numer (rank) procesu odbierającego w ramach komunikatora **comm**
- **tag** – etykieta wiadomości

C:

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

Fortran:

```
CALL MPI_SEND ( <type> buf, integer count, integer datatype,
                integer dest, integer tag, integer comm,
          integer ierror )
```

Odbieranie wiadomości

EURO

- **buf** – adres tablicy zawierającej wiadomość długości **count**, zawierającej elementy typu **datatype**
- **source** – numer (rank) procesu wysyłającego w ramach komunikatora **comm**
- **status** – zawiera informacje o komunikacji (struktura zawierająca `MPI_SOURCE` i `MPI_TAG`)
- Jedynie wiadomość o zgodnej etykiecie (**tag**) zostanie odebrana

C:

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

Fortran:

```
CALL MPI_RECV ( <type> buf, integer count, integer datatype,
                integer source, integer tag, integer comm,
           integer status, integer ierror )
```

Przesyłanie komunikatów

EURO

- Podstawowa metoda przesłania komunikatu pomiędzy dwoma procesami
  - Nadawca wywołuje funkcję **Send**
  - Odbiorca wywołuje funkcję **Receive**

- Poszczególne komunikaty odróżnia etykieta (**tag**)

- `MPI_Send` / `MPI_Recv` to funkcje blokujące tzn. blokują dalsze wykonanie programu do czasu zakończenia komunikacji

Komunikacja point-to-point

EURO

- Najprostsza forma komunikacji
- Proces wysyła komunikat do innego wybranego procesu
- Różne tryby komunikacji point-to-point (blokującej)
  - Standardowa: `MPI_Send`
  - Synchroniczna: `MPI_Ssend`
  - Asynchroniczna / buforowana: `MPI_Bsend`
  - Ready send: `MPI_Rsend`
  - różnią się tym jak restrykcyjna ma być synchronizacja
  - Wszystkie powyższe tryby są blokujące – wysyłanie kończy się w momencie gdy użytkownik może bezpiecznie odczytać i modyfikować bufor wejściowy (tablicę zawierającą komunikat)

- Odbieranie **MPI_Recv** działa ze wszystkimi trybami wysyłania

Komunikacja point-to-point : tryby wysyłania komunikatów

EURO

Tryb wysyłania	Działanie
Synchroniczny  **MPI_Ssend**	Pełna synchronizacja, nie zakłada wcześniejszego wywołania MPI_Recv, zakończy wykonanie, gdy odbiorca rozpocznie odbieranie
Standardowy **MPI_Send**	Wysyłanie blokujące, biblioteka decyduje o użyciu wewnętrznego bufora
Buforowany  **MPI_Bsend**	Korzysta z bufora użytkownika, nie wymaga wcześniejszego wywołanie MPI_Recv, zakończy wykonanie po umieszczeniu wiadomości w buforze
Ready send  **MPI_Rsend**	Zakłada wcześniejsze wywołanie MPI_Recv

Komunikacja point-to-point : tryby wysyłania komunikatów (2)

EURO

- Generyczny **MPI_Send**
  - Biblioteka MPI zadecyduje o trybie wysyłki
  - Zapewnia minimalny czas transferu
- Wysyłanie buforowane **MPI_Bsend**
  - Wymaga zapewnienia przez użytkownika bufora odpowiedniej wielkości
    - `MPI_Buffer_attach` tworzy bufor,
    - `MPI_Buffer_detach` zwalnia bufor
    - W przypadku przepełnienia bufora nastąpi błąd
    - Niskie opóźnienie kosztem przepustowości
- Synchroniczne wysyłanie **MPI_Ssend**
  - Ryzyko zakleszczenia
  - Najlepsza przepustowość kosztem opóźnień lub przestojów
- „Ready send” **MPI_Rsend**
  - Zakończy się błędem, jeżeli nie wystąpi wcześniej odpowiednie wywołanie MPI_Recv
  - Niebezpieczne dla poprawności, nie zalecane do użycia
  - Potencjalnie najszybsze

Komunikacja wieloznaczna

EURO

- Odbiorca może odbierać komunikaty w niezdefiniowanej kolejności
  - Od dowolnego odbiorcy (rank): **`MPI_ANY_SOURCE`**
  - Z dowolną etykietą (tag): **`MPI_ANY_TAG`**
- Faktyczny identyfikator nadawcy / tag odbiorca może odczytać z parametru `status` zwracanego przez MPI_Recv
- **MPI_Probe** pozwala na uzyskanie informacji o komunikacie bez odbierania go (przed wywołaniem MPI_Recv), informacja zwracana jest poprzez parametr `status`

C:

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

Fortran:

```
CALL MPI_PROBE (INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
IERROR)
```

Komunikacja grupowa (kolektywna)

EURO

- Komunikacja grupowa, czyli taka, w której uczestniczy grupa procesów MPI
- Dana funkcja wywoływana jest przez wszystkie procesy w komunikatorze
- W MPI wyróżniamy następujące typy komunikacji grupowej:
  - Bariera synchronizacyjna
  - operacja Broadcast – nadaj wiadomość wielu procesom
  - operacja typu rozrzuć (z ang. scatter)
  - operacja typu zbierz (z ang. gather)
  - operacja wszyscy do wszystkich
  - operacja redukcji (np. globalna suma, globalne maksimum,..)

Komunikacja grupowa: bariera synchronizacyjna

EURO

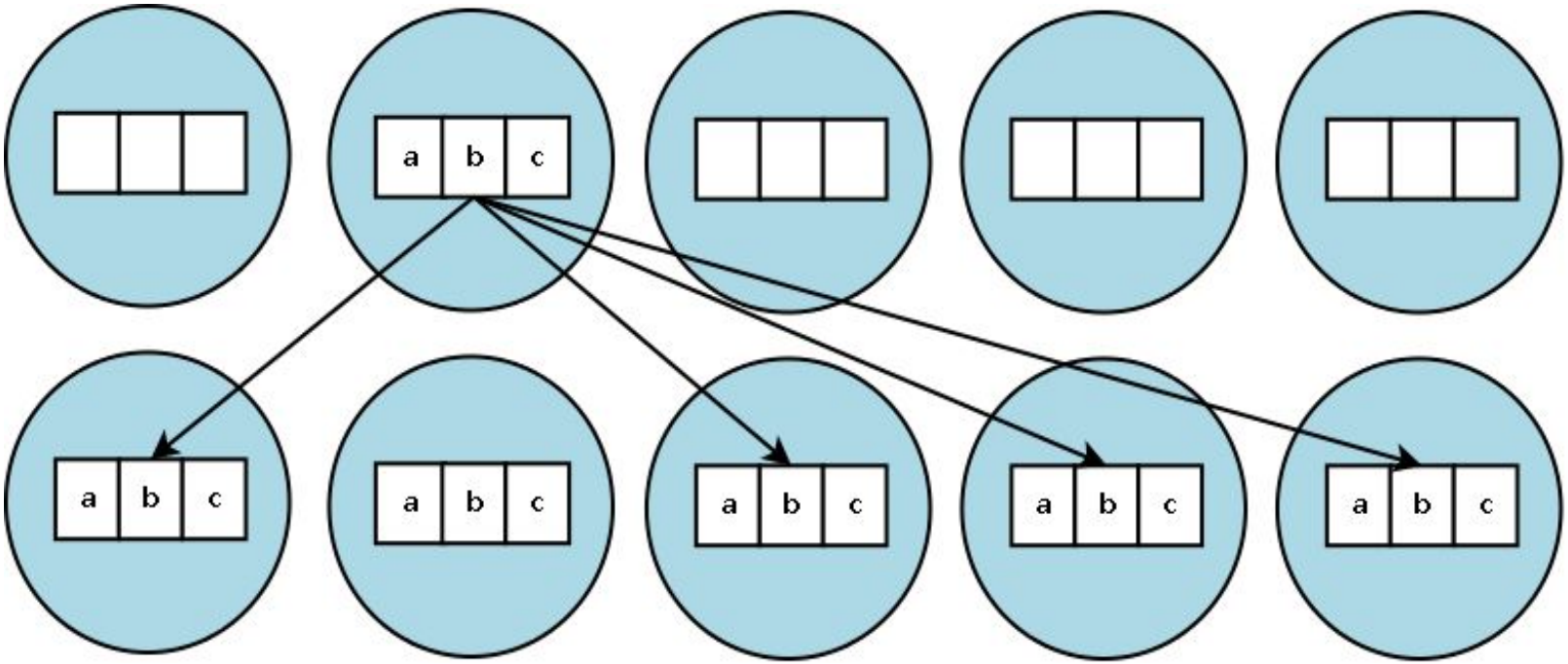
- Zwykle bariera nie jest potrzebna, gdyż odpowiednia synchronizacja powinna być zapewniona na bazie interfejsu przesyłania komunikatów

- Może być używana do:
  - debugowania programów równoległych
  - profilingu programów równoległych
  - synchronizacji do operacji I/O

C:
```
int MPI_Barrier(MPI_Comm comm);
```
Fortran:
```
CALL MPI_BARRIER(integer comm, integer ierror)
```

Komunikacja grupowa: Broadcast

- Operacja polegająca na rozesłaniu tej samej wiadomości do wszystkich procesów grupy



C:

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

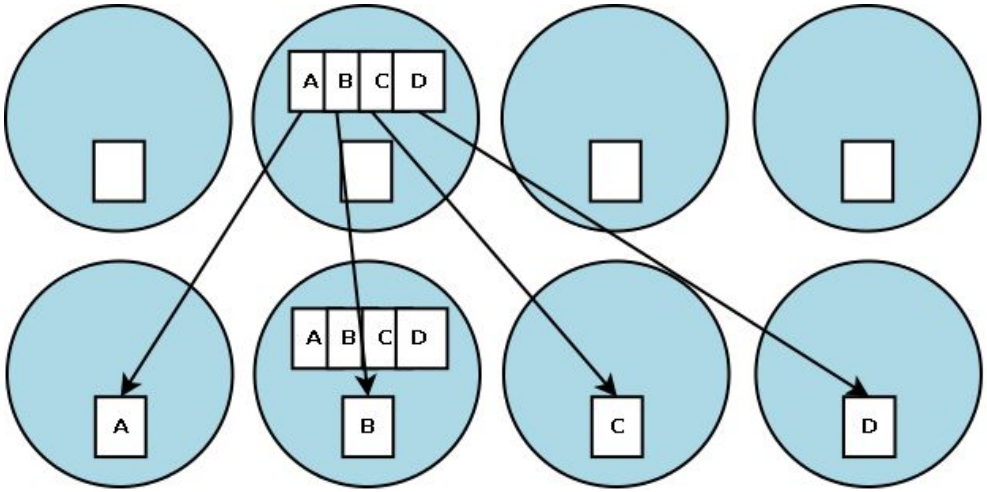
Fortran:

```
CALL MPI_BCAST(choice buffer, integer count, integer datatype, integer
root, integer comm, integer ierror)
```

Komunikacja grupowa: Scatter

EURO

- Operacja polegająca na rozesłaniu przez proces `root` wektora `sendbuf` podzielonego na kawałki `sendcount` do procesów w grupie (również do siebie)



C:

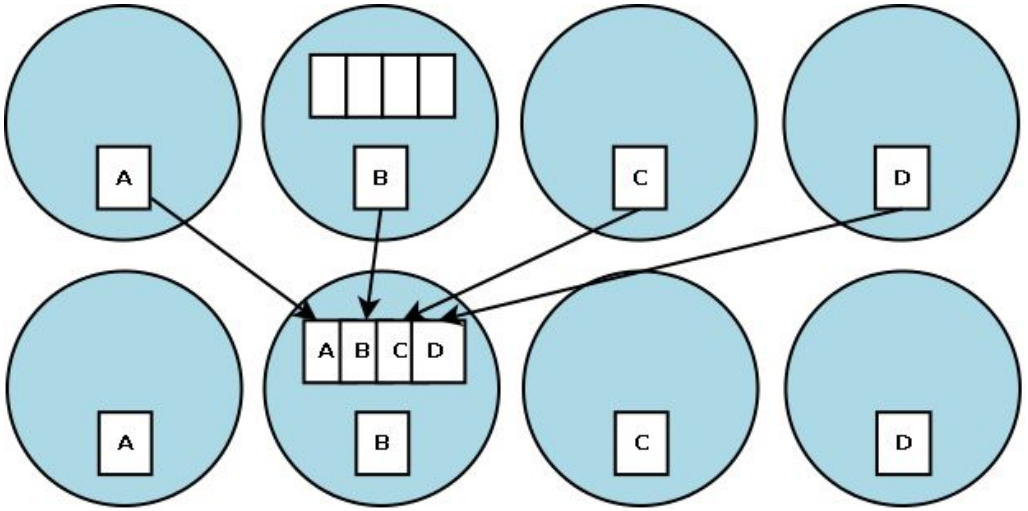
```
int MPI_Scatter(void* sendbuf,int sendcount,MPI_Datatype
sendtype,void* recvbuf, int recvcount,MPI_Datatype recvtype,int
root,MPI_Comm comm);
```

Fortran:

```
CALL MPI_SCATTER(choice sendbuf, integer sendcount, integer sendtype,
choice recvbuf, integer recvcount, integer recvtype, integer root,
integer comm, integer ierror)
```

Komunikacja grupowa: Gather

- Operacja odwrotna do scatter, czyli zebranie kawałków wektora z procesorów w grupie i zapisanie ich w buforze wybranego procesora



C:

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

Fortran:

```
CALL MPI_GATHER(choice sendbuf, integer sendcount, integer sendtype,
choice recvbuf, integer recvcount, integer recvtype, integer root,
integer comm, integer ierror)
```

Komunikacja grupowa: wszyscy do wszystkich

EURO

- Każdy z procesów wysyła do procesu *i*-tego `sendcount` kolejnych elementów typu `sendtype`, poczynając od elementu *i*sendtype* tablicy `sendbuf`

C:

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

Fortran:

```
CALL MPI_ALLTOALL(choice sendbuf, integer sendcount, integer
sendtype, choice recvbuf, integer recvcount, integer recvtype,
integer comm, integer ierror)
```

Komunikacja grupowa: operacja redukcji

- Służy do wykonywania globalnej operacji na elementach procesów należących do grupy

- Operacja typu: $d_0 \mathbf{o} d_1 \mathbf{o} d_2 \mathbf{o} \ldots \mathbf{o} d_{n-1}$
  - d_i to dane w procesie o numerze i (skalar lub wektor)
  - **o** to pewna zdefiniowana operacja

- Istnieje możliwość skorzystania z gotowych definicji operacji takich jak suma czy maksimum

- Istnieje możliwość zdefiniowania własnych operacji funkcją `MPI_Op_create(..)`

Komunikacja grupowa: przykład – globalna suma

EURO

- Chcemy dokonać sumowania pewnej wartości zawartej w `inbuf` po wszystkich procesach w grupie
- Użyjemy operacji `MPI_SUM`
- Wynik ma zostać zwrócony na `resultbuf`

C:
```
int MPI_Reduce( &inbuf, &resultbuf, 1, MPI_INT, MPI_SUM, root,
MPI_COMM_WORLD );
```

Fortran:
```
CALL MPI_REDUCE( inbuf, resultbuf, 1, MPI_INTEGER, MPI_SUM,
root, MPI_COMM_WORLD, ierror)
```

- Dostępna jest również operacja **MPI_ALLREDUCE** (bez parametru `root`), która jest połączeniem operacji Reduce i Broadcast – zwraca wynik w każdym procesie

Komunikacja grupowa: dostępne operacje redukcji

EURO

Nazwa operacji	Funkcja
MPI_MAX	Maksimum
MPI_MIN	Minimum
MPI_SUM	Suma
MPI_PROD	Iloczyn
MPI_LAND	Logiczna koniunkcja
MPI_BAND	Koniunkcja binarna
MPI_LOR	Logiczna alternatywa
MPI_BOR	Alternatywa binarna
MPI_LXOR	Logiczna alternatywa wykluczająca
MPI_BXOR	Binarna alternatywa wykluczająca
MPI_MAXLOC	Maksimum i jego pozycja
MPI_MINLOC	Minimum i jego pozycja

Blokada (deadlock)

EURO

0 Wywołuje: MPI_Ssend i czeka aż odbiorca (1) rozpocznie odbieranie

1 Wywołuje: MPI_Ssend i czeka aż odbiorca (0) rozpocznie odbieranie

- **Nikt nie odbiera!**
- Komunikacja odbywa się w trybie blokującym
- Ssend czeka na rozpoczęcie odbierania (Recv)
- **Rozwiązanie**
- Wywołać Send/Recv naprzemiennie
- Użyć funkcji **MPI_Sendrecv**
- Użyć komunikacji **nieblokującej**

Komunikacja nieblokująca

EURO

- zwróć wykonanie do procesu od razu i pozwól na wykonywanie innej pracy
- w dalszej części proces powinien sprawdzić (**test**) lub poczekać (**wait**) na zakończenie operacji nieblokującej – są to operacje wymagane
- operacja nieblokująca + następująca po niej od razu operacja **wait** = operacja blokująca
- stosowane w celu uniknięcia impasów (blokad, z ang. deadlocks) oraz bezczynności

Komunikacja nieblokująca (2)

EURO

- Komunikacja realizowana w **3 krokach**:
  1. inicjalizacja nieblokującej komunikacji
     - natychmiast zwraca wykonanie do procesu
     - nazwa funkcji rozpoczyna się od **MPI_I**… (immediate)
  2. wykonanie pracy, obliczenia, inna komunikacja
  3. czekanie na zakończenie komunikacji nieblokującej
- Nieblokujące wysłanie:

```
MPI_Isend(…)
//wykonanie pracy
MPI_Wait(…)
```

- Nieblokujący odbiór:

```
MPI_Irecv(…)
//wykonanie pracy
MPI_Wait(…)
```

Komunikacja nieblokująca: wysłanie

EURO

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

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

Fortran:
  CALL MPI_ISEND( choice buf, integer count, integer datatype,
                  integer dest, integer tag, integer comm,
                  integer request, integer ierror )

  CALL MPI_WAIT( integer request, integer
                 status(MPI_STATUS_SIZE), integer ierror )
```

Komunikacja nieblokująca: odbiór

EURO

- **UWAGA:** **buf** nie powinien być używany pomiędzy **Irecv** a operacją **wait**

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

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

Fortran:
  CALL MPI_ISEND( choice buf, integer count, integer datatype,
                  integer source, integer tag, integer comm,
                  integer request, integer ierror )

  CALL MPI_WAIT( integer request, integer
                 status(MPI_STATUS_SIZE), integer ierror )
```

Komunikacja nieblokująca: uwaga na Fortran!

- Kompilatory Fortran wykorzystują intensywną optymalizację (tzw. Register Optimization) zmieniająca kod programu bez naszej wiedzy
- Kod:

```
CALL MPI_IRECV( buf, … , request, ierror)

CALL MPI_WAIT(request, status, ierror )

WRITE(*,*) buf
```

- Może zostać zamieniony na:

```
CALL MPI_IRECV( buf, … , request, ierror)
registerA = buf
CALL MPI_WAIT(request, status, ierror )
WRITE(*,*) registerA
```

- **Potencjalne źródło błędów trudnych do wykrycia**
- Rozwiązanie: wywołać **MPI_Get_address**(buf, …) bezpośrednio po MPI_Wait

Komunikacja nieblokująca: synchronizacja

EURO

- Oczekiwanie na zakończenie operacji nieblokujących:
  - z użyciem blokującej funkcji **MPI_Wait**
  - z użyciem pętli, w której sprawdzamy wartość flagi zwróconej z wywołania funkcji **MPI_Test** (`flag == 1` lub `.TRUE.`)

C:

```
int MPI_Wait ( MPI_Request *request, MPI_Status *status );

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

Fortran:

```
CALL MPI_WAIT( integer request, integer
               status(MPI_STATUS_SIZE), integer ierror )

CALL MPI_TEST( integer request, integer flag, integer
               status(MPI_STATUS_SIZE), integer ierror)
```

Komunikacja nieblokująca: request handles

EURO

- Zmienne typu **MPI_Request** (Fortran - Integer) służą do synchronizacji komunikacji nieblokującej
- Muszą być zmiennymi lokalnymi
- Wartość zmiennej typu request jest
  - generowana przez nieblokująca funkcję komunikacyjną
  - używana i zwalniana przesz funkcję MPI_Wait

Komunikacja nieblokująca i blokująca

EURO

- Wysyłanie i obieranie może być blokujące lub nieblokujące
- Blokujące wysyłanie może być użyte z nieblokującym odbieraniem i vice-versa
- Nieblokujące wysyłanie może używać trybów analogicznych do blokującego
  - Generyczne **MPI_Isend**
  - Synchroniczne **MPI_Issend**
  - Buforowane **MPI_Ibsend**
  - Ready send **MPI_Irsend**
- W każdym z tych trybów sterowanie jest zwracane od razu (immediate) do programu
- Operacja nieblokująca i bezpośrednio następujące wait jest równoważne analogicznej operacji blokującej

Komunikacja nieblokująca: wiele komunikatów

EURO

- Kilka funkcji operujących na grupie komunikacji nieblokujących:
  - poczekaj na zakończenie lub sprawdź status jednej wiadomości z grupy:

```
int MPI_Waitany(int count,MPI_Request array_of_requests[],int *index,
                MPI_Status *status )
int MPI_Testany(int count,MPI_Request array_of_requests[],int *index,
                int *flag, MPI_Status *status )
```

  - poczekaj na zakończenie lub sprawdź status wszystkich wiadomości z grupy:

```
int MPI_Waitall(int count,MPI_Request array_of_requests[],MPI_Status
                array_of_statuses[] )
int MPI_Testall(int count,MPI_Request array_of_requests[],int *flag,
                MPI_Status array_of_statuses[] )
```

  - poczekaj na zakończenie lub sprawdź status tylu wiadomości ile jest możliwe:

```
int MPI_Waitsome(int incount,MPI_Request array_of_requests[],int
      *outcount,int array_of_indices[],MPI_Status
array_of_statuses[])
int MPI_Testsome(int incount,MPI_Request array_of_requests[],int
      *outcount,int array_of_indices[],MPI_Status
array_of_statuses[])
```

Typy danych w MPI

EURO

- Typy określają sposób ułożenia przesyłanych i odbieranych wiadomości w pamięci

```
int MPI_Isend(&buf,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD,&req);
```

- Rozróżniamy:
  - typy podstawowe zdefiniowane w ramach biblioteki MPI

    `MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG,`
    `MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED,`
    `MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE,`
    `MPI_LONG_DOUBLE, MPI_BYTE`
  - typy upakowane, specyfikowane przez `MPI_PACKED`
  - typy definiowalne, np. wektory, struktury, inne

Typy danych: typy podstawowe

Typ MPI	Typ języka 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	
MPI_PACKED	

Typ MPI	Typ Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Typy danych: typy MPI

EURO

- Typ danych wyspecyfikowany w komendzie send powinien dokładnie odpowiadać typowi wyspecyfikowanemu w komendzie recv
- **MPI_BYTE** = 8 znaków binarnych
- **MPI_PACK** umożliwia pakowanie elementów różnych typów do jednego bufora
- **MPI_PACK** może również reprezentować dowolny typ danych
- Jeśli nasza wiadomość nie jest tablicą elementów określonego typu, tylko niejednorodną wiadomością zawierająca np. dwie liczby całkowite i tablicę liczb zmiennoprzecinkowych, wówczas musimy skorzystać z jednego z dwóch mechanizmów tworzenia takich wiadomości:
  - pakowanie danych
  - tworzenie własnych typów danych MPI

Typy danych: Pakowanie danych

EURO

- MPI udostępnia możliwość przesłania bufora pamięci zawierającego spakowane elementy różnych typów
- Przesłanie takiego elementu wykonujemy przy użyciu typu **MPI_PACKED**
- Aby wykorzystać ten mechanizm należy:
  - Spakować dane za pomocą funkcji **MPI_PACK**
  - Przesłać bufor pamięci za pomocą komendy **send**
  - Odebrać bufor pamięci za pomocą komendy **recv**
  - Rozpakować dane za pomocą komendy **MPI_UNPACK**
- Rozwiązanie dosyć wygodne, ale związane z kopiowaniem danych do nowego miejsca w pamięci

Typy danych: Pakowanie danych (2)

C:

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

Fortran:

```
CALL MPI_PACK(choice inbuf, integer incount, integer datatype,
choice outbuf, integer outsize, integer position, integer comm,
integer ierror)
```

- Pakowanie danych do bufora
  - inkrementalne dodawanie elementów do bufora
  - pakowanie do bufora o nazwie `outbuf` i rozmiarze `outcount` , począwszy od pozycji `position`, danych w liczbie `incount` pochodzących z bufora `inbuf` o typie `datatype`
  - dodatkowo istnieje możliwość skorzystania z grup komunikatorów (np. grup odpowiadających architekturom procesorów)
  - argument `position` musi zostać przekazany przez adres, gdyż zwracane jest na nim pozycja w buforze wyjściowym po dodaniu nowego elementu (w bajtach)

Typy danych: Pakowanie danych (3)

- Aby przygotować bufor wyjściowy odpowiedniej wielkości możemy wpierw sprawdzić wymagany rozmiar za pomocą funkcji **MPI_Pack_Size**
- Funkcja ta zapisuje pod zmienną `size` ilość bajtów potrzebną do upakowania `incount` elementów typu `datatype`

C:
```
int MPI_Pack_size (int incount, MPI_Datatype datatype, MPI_Comm comm,int *size )
```
Fortran:
```
CALL MPI_PACK_SIZE(integer incount, integer datatype, integer comm, integer size, integer ierror)
```

- Do rozpakowania danych z przesłanego bufora możemy użyć funkcji **MPI_Unpack**

C:
```
int MPI_Unpack ( void *inbuf, int insize, int *position,void *outbuf, int outcount, MPI_Datatype datatype,MPI_Comm comm )
```
Fortran:
```
CALL MPI_UNPACK(choice inbuf, integer insize, integer position, choice outbuf, integer outcount, integer datatype, integer comm, integer ierror)
```

Typy danych: przykład pakowania danych

EURO

C:

```
int position, i, j, a[2];
char buff[1000];


....
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
  …
  position = 0;
  MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
  MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
} else   /* RECEIVER CODE */
  MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD)
}
```

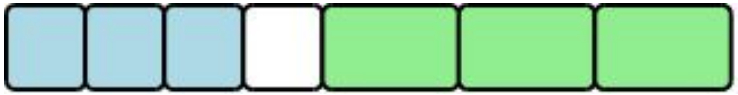
Typy danych: definiowalne typy MPI

EURO

- MPI umożliwia nam definiowanie 3 rodzajów własnych typów:
  - typy reprezentujące ciągły fragment pamięci zawierający elementy tego samego typu – `MPI_Type_Contiguous`

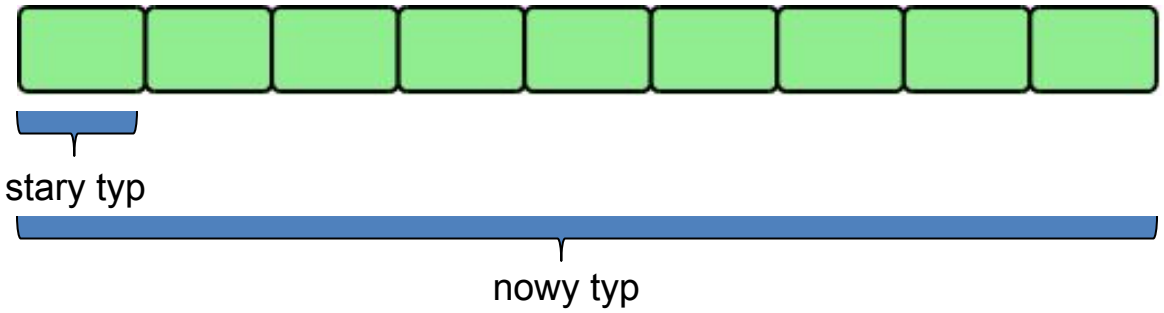
  - typy będące wektorami elementów danego typu, które nie reprezentują ciągłego fragmentu pamięci – `MPI_Type_Vector`

  - typy będące strukturami zawierającymi elementy różnych typów, które niekoniecznie reprezentują ciągły fragment pamięci – `MPI_Type_Struct`



Typy danych: ciągłe fragmenty pamięci

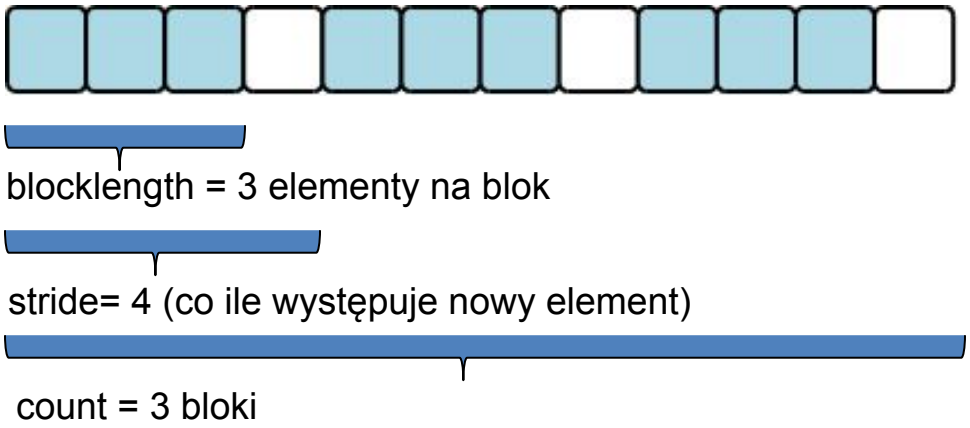
- Najprostszy definiowalny typ danych MPI



C:
```
int MPI_Type_contiguous(int count,MPI_Datatype oldtype, MPI_Datatype
*newtype);
```
Fortran:
```
CALL MPI_TYPE_CONTIGUOUS(integer count, integer oldtype, integer
newtype, integer ierror)
```

Typy danych: typ wektorowy

- Używane jeśli interesujące nas wektory danego typu nie reprezentują ciągłego fragmentu pamięci, np. pomiędzy nimi znajdują się elementy, które nie powinny być przesłane/odebrane



C:

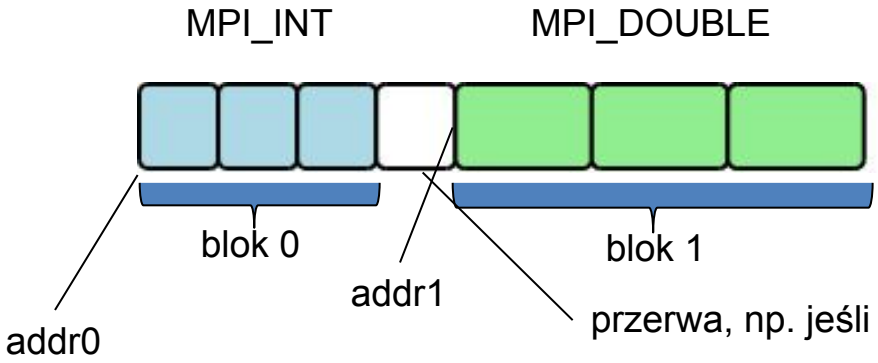
```
int MPI_Type_vector(int count,int blocklength,int stride, MPI_Datatype
oldtype,MPI_Datatype *newtype);
```

Fortran:

```
CALL MPI_TYPE_VECTOR(integer count, integer blocklength, integer
stride, integer oldtype, integer newtype, integer ierror)
```

Typy danych: struktury

- Używane do przesyłania struktur zawierających elementy różnego typu, które niekoniecznie reprezentują ciągły fragment pamięci




```
Count = 2
Array_of_blocklengths = {3,3}
Array_of_displacements =
{0,addr1-addr0}
Array_of_types = {MPI_INT,MPI_DOUBLE}
```

C:

```
int MPI_Type_struct(int count,int *array_of_blocklengths, MPI_Aint
*array_of_displacements,MPI_Datatype *array_of_types, MPI_datatype
*newtype);
```

Fortran:

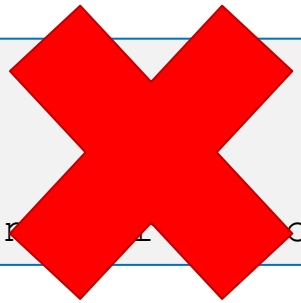
```
CALL MPI_TYPE_STRUCT(integer count, integer array_of_blocklengths(*),
integer array_of_displacements(*), integer array_of_types(*), integer
newtype, integer ierror)
```

Typy danych: struktury (2)

- Aby obliczyć przemieszczenie potrzebne są nam adresy poszczególnych bloków
  - Możemy do tego celu użyć funkcji MPI

MPI-1

```
C:
int MPI_Address(void* location,MPI_Aint *address);
Fortran:
CALL MPI_ADDRESS(choice location, integer address, ierror)
```



Usunięte w MPI-3

MPI-2

```
C:
int MPI_Get_address(void *location, MPI_Aint *address);
Fortran:
CALL MPI_GET_ADDRESS(choice location(*), integer
(KIND=MPI_ADDRESS_KIND) address, integer ierror)
```

Typy danych: zatwierdzenie nowego typu

- Zanim użyjemy nowego typu danych MPI, **musimy go zatwierdzić za pomocą komendy MPI_TYPE_COMMIT**
- Musi być to wykonane tylko raz dla każdego procesu MPI

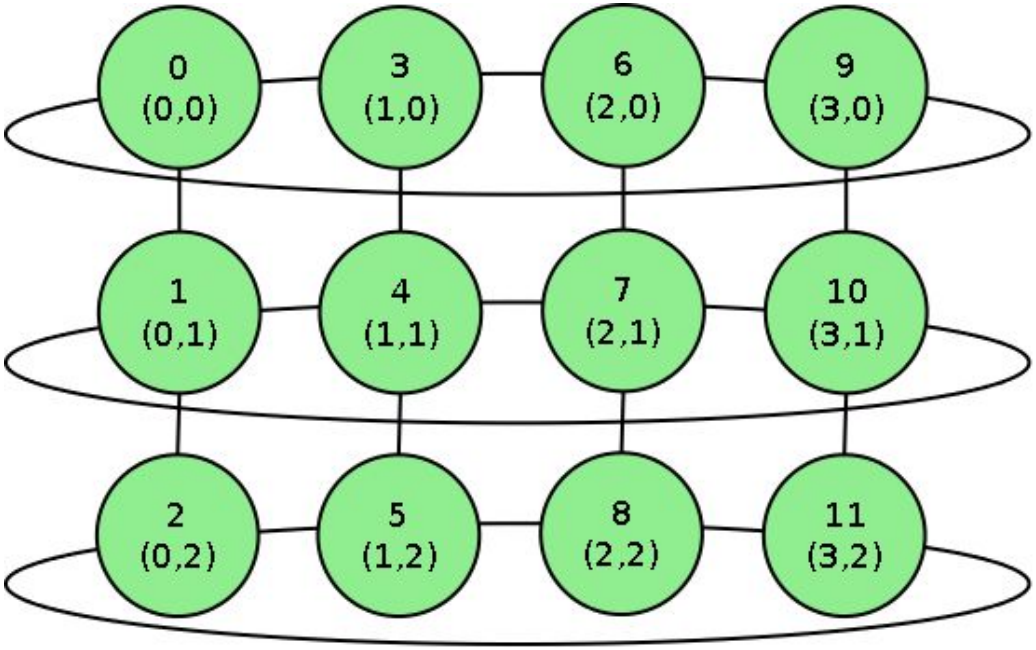
C:
```
int MPI_Type_commit(MPI_Datatype *datatype);
```

Fortran:
```
CALL MPI_TYPE_COMMIT(integer datatype, integer ierror)
```

Wirtualne topologie

EURO

- Wirtualne topologie to mechanizm pozwalający nazywać procesy w komunikatorze w sposób, który lepiej pasuje do schematu komunikacji pomiędzy procesami
- Wirtualnych topologii użyjemy na przykład jeśli obliczenia wykonywane są na dwu-wymiarowej siatce i każdy proces komunikuje się tylko i wyłącznie ze swoimi najbliższymi sąsiadami w siatce



Topologia kartezjańska (periodyczna siatka dwuwymiarowa)

Wirtualne topologie (2)

EURO

- Za mechanizmem wirtualnych topologii kryje się czysta arytmetyka liczb całkowitych
- Zyski ze stosowania tego mechanizmu:
  - Prostszy kod
  - Zmienne odwzorowujące schemat komunikacji
  - umożliwia lepsza wydajność – podpowiadamy bibliotece jak wygląda schemat komunikacji
- Nowa topologia tworzy nowy komunikator
- MPI zapewnia funkcje mapowania indeksów
  - w celu obliczenia numeru procesu na podstawie nazewnictwa stosowanego w topologii
  - w celu obliczenia współrzędnych w topologii na podstawie numeru procesu (topologia kartezjańska)

Wirtualne topologie: rodzaje

EURO

- Topologie kartezjańskie
  - Każdy proces jest połączony z sąsiadami w wirtualnej siatce
  - Brzegi mogą być periodyczne (cykliczne, z zapętleniem) lub bez
  - Procesy są identyfikowane współrzędnymi kartezjańskimi w formie (x1,x2,…)
- Topologie grafowe
  - Dowolne grafy
  - Dwie formy
    - **GRAPH** (MPI-1)
    - **DIST_GRAPH** (skalowalna wersja MPI-2.2 i MPI-3)

Wirtualne topologie: topologie kartezjańskie

- Utworzenie nowego komunikatora o topologii kartezjańskiej
- `ndims` – wymiar topologii kartezjańskiej (1 – pierścień, 2 – cylinder lub siatka 2D, 3 – torus lub siatka 3D)
- `dims` – tablica definiująca liczbę procesów w każdym wymiarze
- `periods` – tablica definiująca czy siatka zawiera zapętlenia w każdym z wymiarów
- `reorder` – czy pozwalamy na zmianę numerów procesów w nowym komunikatorze

C:

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

Fortran:

```
CALL MPI_CART_CREATE(INTEGER COMM_OLD,INTEGER NDIMS,INTEGER DIMS(*),
INTEGER PERIODS(*),INTEGER REORDER,INTEGER COMM_CART,INTEGER IERROR)
```

Wirtualne topologie: mapowanie procesów

EURO

C:

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

Fortran:

```
CALL MPI_CART_COORDS(INTEGER COMM,INTEGER RANK,INTEGER MAXDIMS,
INTEGER COORDS(*),INTEGER IERROR)
```

C:

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

Fortran:

```
CALL MPI_CART_RANK(INTEGER COMM,INTEGER COORDS(*),INTEGER RANK,
INTEGER IERROR)
```

- Funkcja zapisuje pod adresem `rank` numer procesu o współrzędnych określonych w tablicy `coords`.

Wirtualne topologie: procesy sąsiednie

EURO

- Funkcja ta przesuwa numer lokalnego procesu wzdłuż wyspecyfikowanego wymiaru, w celu wygenerowania numerów procesów źródłowego i docelowego
- Numer źródła otrzymany jest poprzez odjęcie `displ` od n-tej współrzędnej lokalnego procesu, gdzie n równe jest `direction`. Numer docelowego procesu otrzymywany jest poprzez dodawanie `disp` do n-tej współrzędnej.
- **Współrzędne numerowane są od zera.**

C:

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

Fortran:

```
CALL MPI_CART_SHIFT(INTEGER COMM,INTEGER DIRECTION,INTEGER DISP,
INTEGER RANK_SOURCE,INTEGER RANK_DEST,INTEGER IERROR)
```

- Znajdowanie sąsiadów w topologii kartezjańskiej

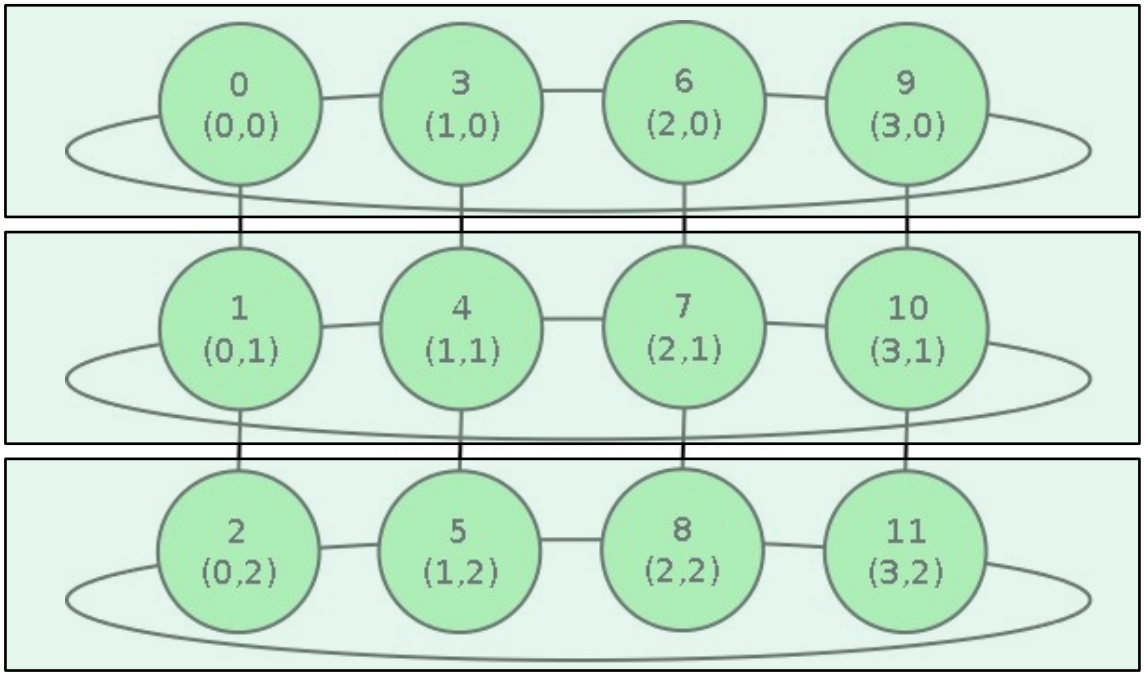
Wirtualne topologie: procesy sąsiednie (2)

EURO

- Jeśli topologia nie jest periodyczna w wybranym wymiarze, to przy wyliczeniach sąsiadów procesorów brzegowych zostanie zwrócona wartość **MPI_PROC_NULL**
- **MPI_PROC_NULL** jest także poprawnym identyfikatorem (rank) odbiorcy lub nadawcy przy komunikacji
- W takim przypadku komunikacja nie następuje ale może to znacznie uprościć kod (nie wymaga odróżniania przypadków na brzegu siatki)

Wirtualne topologie: podziały kartezjańskie

- Podział siatki na przekroje wzdłuż wybranego wymiaru
- Każdy przekrój tworzy nowy komunikator
- W ramach każdego przekroju można wykonać dedykowaną komunikację kolektywną (następny rozdział)



Wirtualne topologie: podziały kartezjańskie (2)

EURO

- Tworzy podzbiór siatki (nowy komunikator o topologii kartezjańskiej)
- Nowy komunikator zawierający przekroje siatki dla wymiarów odpowiadających elementom tablicy `remain_dims` (1 – ten wymiar pozostaje w nowym komunikatorze, 0 – ten wymiar nie uczestniczy w nowym komunikatorze)

C:

```
int MPI_Cart_sub ( MPI_Comm comm_cart, int* remain_dims, MPI_Comm*
comm_sub)
```

Fortran:

```
CALL MPI_CART_SUB(INTEGER COMM_CART,INTEGER REMAIN_DIMS(*),INTEGER
COMM_SLICE,INTEGER IERROR)
```

Podziały komunikatorów

`MPI_Comm_split_type` - dzieli komunikator na rozłączne podzbiory według określonego parametru:

- `MPI_COMM_TYPE_SHARED`
  This type splits the communicator into subcommunicators, each of which can create a shared memory region.

  Nowe w standardzie 4.0:
- `MPI_COMM_TYPE_HW_GUIDED`—this value specifies that the communicator comm is split according to a hardware resource type (for example a computing core or an L3 cache) specified by the "mpi_hw_resource_type" info key.
- `MPI_COMM_TYPE_HW_UNGUIDED`—the group of MPI processes associated with newcomm must be a strict subset of the group associated with comm and each newcomm corresponds to a single instance of a hardware resource type (for example a computing core or an L3 cache).

MPI i wątki

EURO

- Standard MPI-3 określa wszyskie funkcje MPI jako **thread-safe**
- Operacje blokujące blokują wykonanie tylko wywołującego wątku
- Specjalna forma inicjalizacji MPI zalecana przy użyciu wątków

C:
```
int MPI_Init_thread (int *argc, char ***argv, int required, \
   int *provided)
```
Fortran:
```
CALL MPI_INIT(INTEGER REQUIRED, INTEGER PROVIDED, INTEGER IERROR)
```

- Argument `required` określa pożądany poziom współpracy biblioteki z wątkami,
- faktyczny poziom wsparcia zwraca `provided`
  - **MPI_THREAD_SINGLE** tylko jeden wątek per proces (brak wsparcia)
  - **MPI_THREAD_FUNNELED** tylko jeden wątek w ramach procesu używa MPI
  - **MPI_THREAD_SERIALIZED** tylko jeden wątek **jednocześnie** używa MPI
  - **MPI_THREAD_MULTIPLE** pełne wsparcie dla wątków, **bez ograniczeń**

Dodatek: jak sprawdzić wersję biblioteki MPI

EURO

- Operacje typu „sparse-collectives” dostępne są od wersji MPI-3
- Aby sprawdzić dostępną wersję (którą wersję standardu używana biblioteka MPI spełnia) można użyć:
  - Funkcji **MPI_Get_version**

C:

```
int MPI_Get_version(int *version, int *subversion)
```

Fortran:

```
CALL MPI_GET_VERSION(INTEGER VERSION,INTEGER SUBVERSION,INTEGER IERROR)
```

```
const int ver = MPI_VERSION;
```

MPI-2 oraz MPI-3 – przegląd

EURO

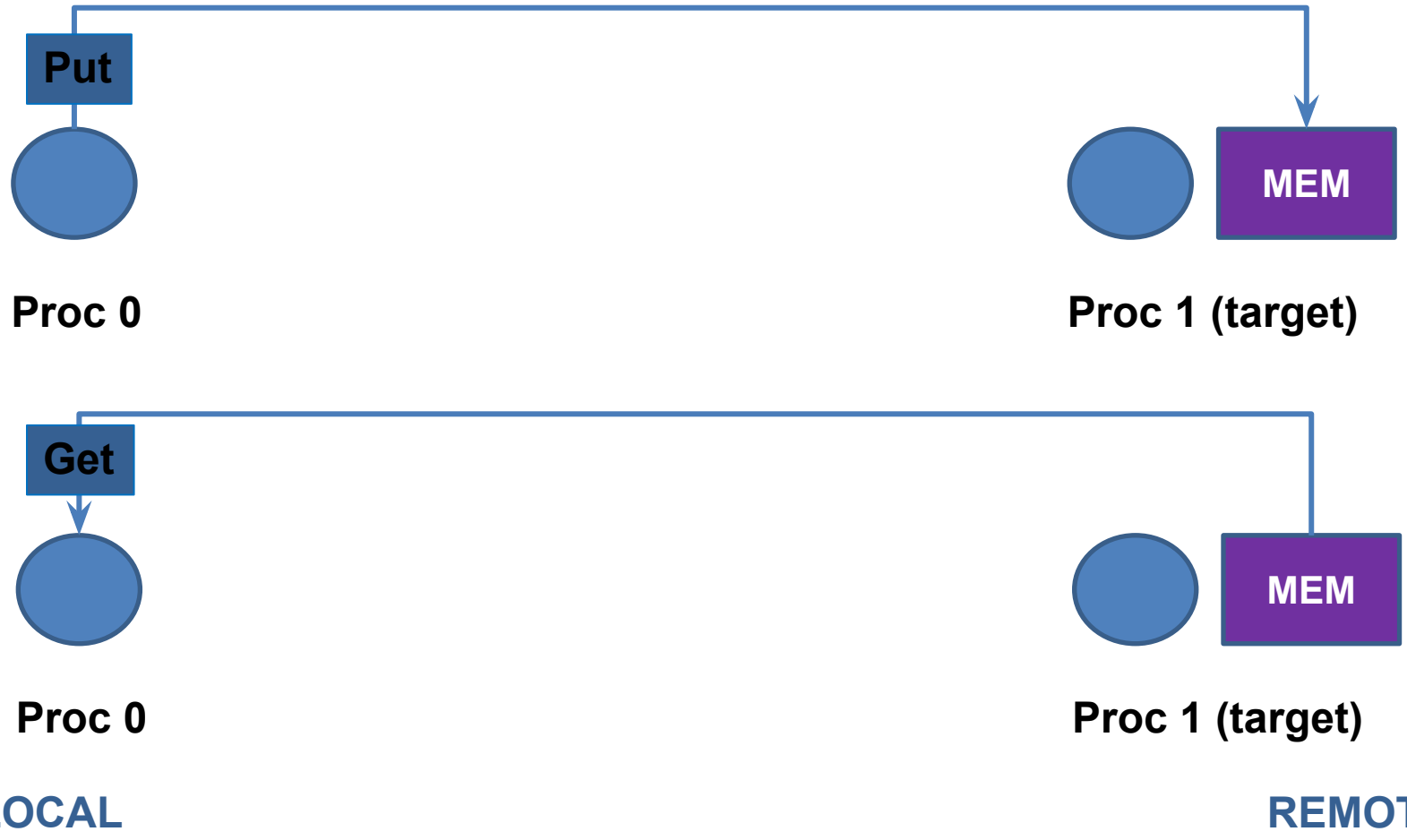
- Funkcjonalności MPI-2
  - Dynamiczne tworzenie i zarządzanie procesami
  - Komunikacja jednostronna
    - RMA (Remote Memory Access)
    - MPI_Put, MPI_Get do zdalnego dostępu do pamięci innego procesu
  - MPI I/O
    - Równoległy dostęp do plików
  - Integracja procesów MPI i wątków (w szczególności OpenMP)
- Funkcjonalności MPI-3
  - Nieblokująca komunikacja grupowa
  - Komunikacja grupowa z sąsiadami
  - Rozszerzona komunikacja jednostronna, m.in. o obsługę pamięci współdzielonej

Wstęp: przekazywanie komunikatów a komunikacja jednostronna

EURO

- Przekazywanie komunikatów (message passing)
  - Komunikacja dwustronna – pomiędzy parami procesów
  - Nadawca i odbiorca są jawnie zaangażowani w komunikację
  - Podstawowe operacje: Send, Recv, Sendrecv
  - Tryby przesyłania blokujące i nieblokujące (+ synchronizacja)
- Komunikacja jednostronna (remote memory access)
  - Komunikacja jednostronna – tylko jeden proces aktywnie uczestniczy
  - Procesy komunikują się poprzez zdalny dostęp do pamięci
  - Zdalny proces udostępnia pamięć, proces lokalny wykonuje transfer danych
  - Podstawowe operacje: Put, Get
  - Różne tryby zależnie od zaangażowanie procesu zdalnego (target)
  - Różne mechanizmy synchronizacji

Wstęp: komunikacja jednostronna



RMA : model pamięci

EURO

- Okno (**window**) to nazwa obszaru w lokalnej pamięci procesu, który podlega operacjom zdalnego dostępu (RMA)
- Zakres dostępu do pamięci zdalnego procesu obejmuje jedynie dane znajdujące się w utworzonych oknach
- Okna tworzy proces (lokalny), który jest właścicielem danych (obszaru pamięci)

Okna pamięci dla RMA

EURO

- Okno to obszar w lokalnej pamięci procesu, do którego mogą mieć dostęp inne procesy
- Inne procesy wykonują operacje na pamięci przypisanej do okna (Put, Get lub Accumulate)
- Okno jest obiektem udostępnianym dla grupy procesów należących do danego komunikatora

Tworzenie okien

- Utworzenie okna poprzez przypięcie wcześniej zalokowanego bufora (buffer)

  `MPI_Win_create(buffer, buffer_size, disp_unit, info, MPI_COMM_WORLD, &win)`
- Operacja kolektywna w grupie procesów (tutaj `MPI_COMM_WORLD`)

- Zniszczenie okna

  `MPI_Win_free(&win)`

C:

```
 int MPI_win_create ( void *base, MPI_Aint size, int disp_unit,
MPI_Info info, MPI_Comm comm, MPI_Win *win);
 int MPI_Win_free ( MPI_Win *win );
```

- Parametr `disp_unit` (displacement) to rozmiar elementów bufora – rozmiar typu (w bajtach) lub 1 dla manipulacji pojedynczymi bajtami

Przesyłanie danych (Put, Get)

EURO

- Operacja **Put** wpisuje dane do zdalnego bufora
- Opis przesyłanych danych jest analogiczny jak w przypadku operacji Send (przesyłanie komunikatów)
- Nazewnictwo:
  - origin – lokalny proces inicjujący operacje RMA (Put, Get)
  - target – zdalny proces udostępniający swoją pamięc
- Operacja **Get** transferuje dane w odwrotnym kierunku – z pamięci procesu *target* do procesu *origin*

C:

```
 int MPI_Put (const void *origin_addr, int origin_count, MPI_Datatype
origin_type, int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_type, MPI_Win win);

 int MPI_Get (const void *origin_addr, int origin_count, MPI_Datatype
origin_type, int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_type, MPI_Win win);
```

Operacja Accumulate

- Operacja podobna do PUT, gdzie dane nie są nadpisywane w oknie procesu target tylko akumulowane (operacja redukcji)

```
C:
 int MPI_Accumulate(const void *origin_addr, int origin_count,
MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
int target_count, MPI_Datatype target_datatype,  MPI_Op op, MPI_Win
win)
```

- Zestaw operacji `MPI_Op` analogiczny jak w przypadku `MPI_Reduce`

Adresacja buforów RMA

EURO

- Przesunięcie względem adresu początkowego bufora zdalnego (`target_disp`) jest parametrem operacji Put i Get
- Jednostką przesunięcia nie są bajty tylko `displacement_unit` zdefiniowane przy tworzeniu odpowiedniego okna (`MPI_Win_create`)

Synchronizacja pamięci zdalnej

EURO

- Dwa rodzaje komunikacji RMA w MPI:
  - **active target**: obydwa procesy wymieniające się danymi uczestniczą aktywnie, proces *origin* steruje przepływem danych, proces *target* uczestniczy jedynie w synchronizacji
  - **passive target**: jedynie proces origin jest jawnie zaangażowany w komunikację
- Dostępne trzy mechanizmy synchronizacyjne:
  - FENCE (bariera), POST-START-COMPLETE-WAIT
  - LOCK (zamek)

Synchronizacja pamięci zdalnej - Fence

EURO

- Najprostsza forma synchronizacja okna – bariera RMA (*active target*):

  `MPI_Win_fence(0, win)`
- Działa podobnie jak `MPI_Barrier` w przypadku trybu przesyłania komunikatów
- Powoduje zakończenie wszystkich transferów RMA rozpoczętych na danym oknie (`win`) od momentu poprzedniego wywołania operacji *Fence*
- Operacja kolektywna wywoływana przez wszystkie procesy używające danego okna
- Podstawowe użycie – przełączanie trybu dostępu do okna: lokalny (zapis do pamięci) i zdalny (operacje Put i Accumulate)
- Parametr `assert` identyfikuje daną barierę dla MPI, może być równy 0

C:

```
int MPI_Win_fence (int assert, MPI_Win win);
```

Synchronizacja pamięci zdalnej - PSCW

EURO

- Mniej restrykcyjna forma synchronizacji pozwalająca na ograniczenie tylko do par komunikujących się procesów (active target)
- Proces *target* udostępnia okno grupie procesów wywołaniem **POST** i zamyka dostęp wywołaniem **WAIT**
- Proces *origin* rozpoczyna dostęp do okna od wywołania **START** i kończy wywołaniem **COMPLETE**

Synchronizacja pamięci zdalnej – Lock

- Zapewnia wyłączność* na dostęp do okna pamięci dla danego procesu o numerze **`rank`** (proces *origin)*
- Proces *target* nie jest zaangażowany (passive target)

```
C:
  int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win);
```

- Odblokowanie i synchronizacja okna poprzez zwolnienie zamka

```
C:
  int MPI_Win_unlock(int rank, MPI_Win win);
```

- Zamki typu MPI_LOCK_SHARED pozwalają na grupowy (nie kolektywny) dostęp poprzez funkcje `MPI_Win_lock_all` i `MPI_Win_unlock_all`

Tworzenie okien (2)

EURO

- Dwa dodatkowe tryby tworzenia okien dla operacji RMA:
  - Okno z alokacją pamięci

```
C:
  int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
     MPI_Comm comm, void *baseptr, MPI_Win *win);
```

  - Okno z dynamicznie przypinaną pamięcią

```
C:
 int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)

 int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size);
 int MPI_Win_detach(MPI_Win win, const void *base)
```

  - **Uwaga** w przypadku okien z pamięcią przypinaną dynamicznie, argument **`target_disp`** w operacjach na tym oknie jest adresem względnym (należy go wcześniej przekazać do procesu *origin*)

Obiekty typu MPI_Info

EURO

- Pusty obiekt typu **`MPI_Info`** tworzy funkcja:
  - **`MPI_Info_create( info)`**
- Dodawanie zawartość klucz-wartość obiektu:
  - **`MPI_Info_set(info, key, value)`**
- Usuwanie zawartości klucz-wartość:
  - **`MPI_Info_delete(info, key)`**
- Dodatkowo do zarządzanie obiektami typu **`MPI_Info`** służą funkcje:
  - **`MPI_Info_dup`** – duplikuje obiekt
  - **`MPI_Info_free`** – zwalnia obiekt
  - **`MPI_Info_get`** – zwraca wartość odpowiadająca kluczowi
  - **`MPI_Info_get_nkeys`** – zwraca liczbę zdefiniowanych kluczy
- Obiekty tego typu używa się np. przy tworzeniu okien dla transferów RMA

Podsumowanie

EURO

- **Omówione zagadnienia**
  - MPI jako narzędzie tworzenia zrównoleglonych programów
  - Podstawy użycia, tworzenia prostych programów z wykorzystaniem
    - Komunikacji point-to-point blokującej i nieblokującej
    - Komunikacji grupowej (kolektywnej)
    - Wirtualnych topologii
  - Typy danych w MPI i podstawowe użycie
  - MPI i programowanie w modelu RMA
  - Podstawy użycia, tworzenia programów z wykorzystaniem
    - Operacji PUT i GET
    - Różnych trybów synchronizacji
  - Materiał obejmował tylko podstawową funkcjonalność MPI-1 i MPI-2
  - Kilka przykładów nowości z MPI-3
  - Zadania pozwalające na pierwsze kroki w zrównoleglaniu własnych programów

- **Nieomówione zagadnienia**
  - Komunikacja point-to-point
    - MPI_Sendrecv i MPI_Sendrecv_replace
    - MPI_Iprobe, MPI_Request_free, MPI_Cancel
  - Komunikacja grupowa
    - MPI_Alltoall, MPI_Scan i MPI_Exscan
    - MPI_Gatherv, Scatterv, Allgatherv, Alltoallv, Alltoallv
  - Wirtualne topologie
    - MPI_Dims_create, MPI_Graph_neighbors, MPI_Graph_neighbors_count
    - MPI_Dist_graph_create_adjacent, MPI_Dist_graph_neighbors
  - Stałe specjalne MPI i predefiniowane zmienne, zmienne środowiskowe
  - Współpraca z wątkami
    - MPI_Is_thread_main, MPI_Query_thread
  - MPI IO dla równoległych operacji na plikach

Zasoby HPC dla nauki

- Zasoby ICM UW
  - Okeanos
  - Topola
  - Rysy (GPU)
- EuroHPC.pl
  - https://www.eurohpc.pl
- EuroHPC JU
  - LUMI
  - https://lumi-supercomputer.eu/

Ćwiczenia

Zadania

EURO

Propozycje ćwiczeń

- Hello - komunikator i przesyłanie komunikatów
- Kwadratura - komunikacja kolektywna
- Ping-pong (samodzielnie)

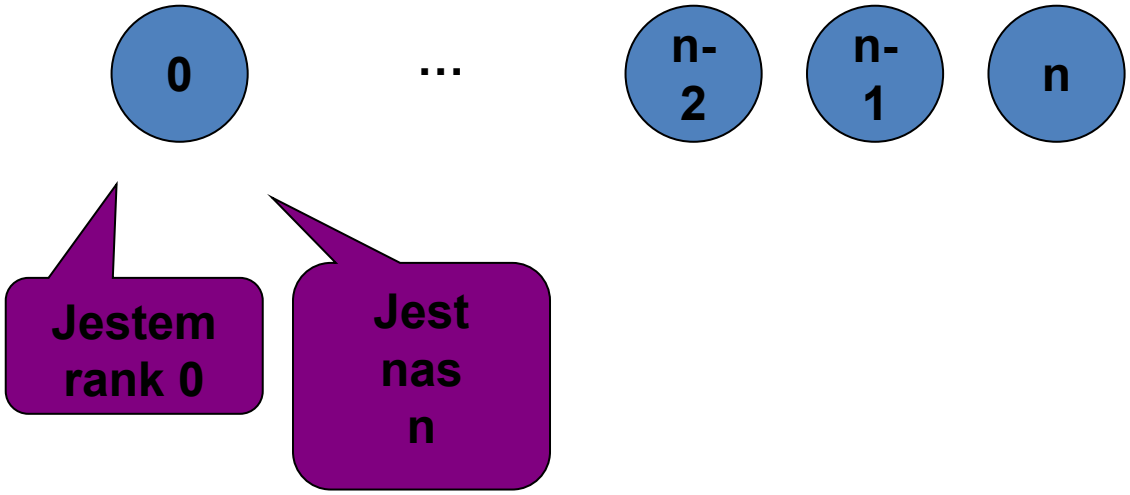
Zadanie 1

POWITANIE

`hello/hello0.c`

Pobieranie informacji z komunikatora

Zadanie: każdy proces wypisuje swój identyfikator systemowy (PID) numer (rank) oraz liczbę wszystkich procesów w domyślnym komunikatorze



Pliki nagłówkowe

EURO

Pliki nagłówkowe

C:

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

Fortran 90:

```
    use mpi
```

Fortran 77:

```
    include 'mpif.h'
```

Podstawy: kompilacja i uruchomienie

EURO

- **Ustawienie sesji (SLURM) i środowiska**

```
>> srun -n 4 -reservation=computing --pty --time 15 bash -l

>> module load common/mpi/openmpi/4.0.5
```

- **Kompilacja**

```
>> mpicc hello0.c
```

- **Uruchomienie**

```
>> mpiexec ./a.out
```

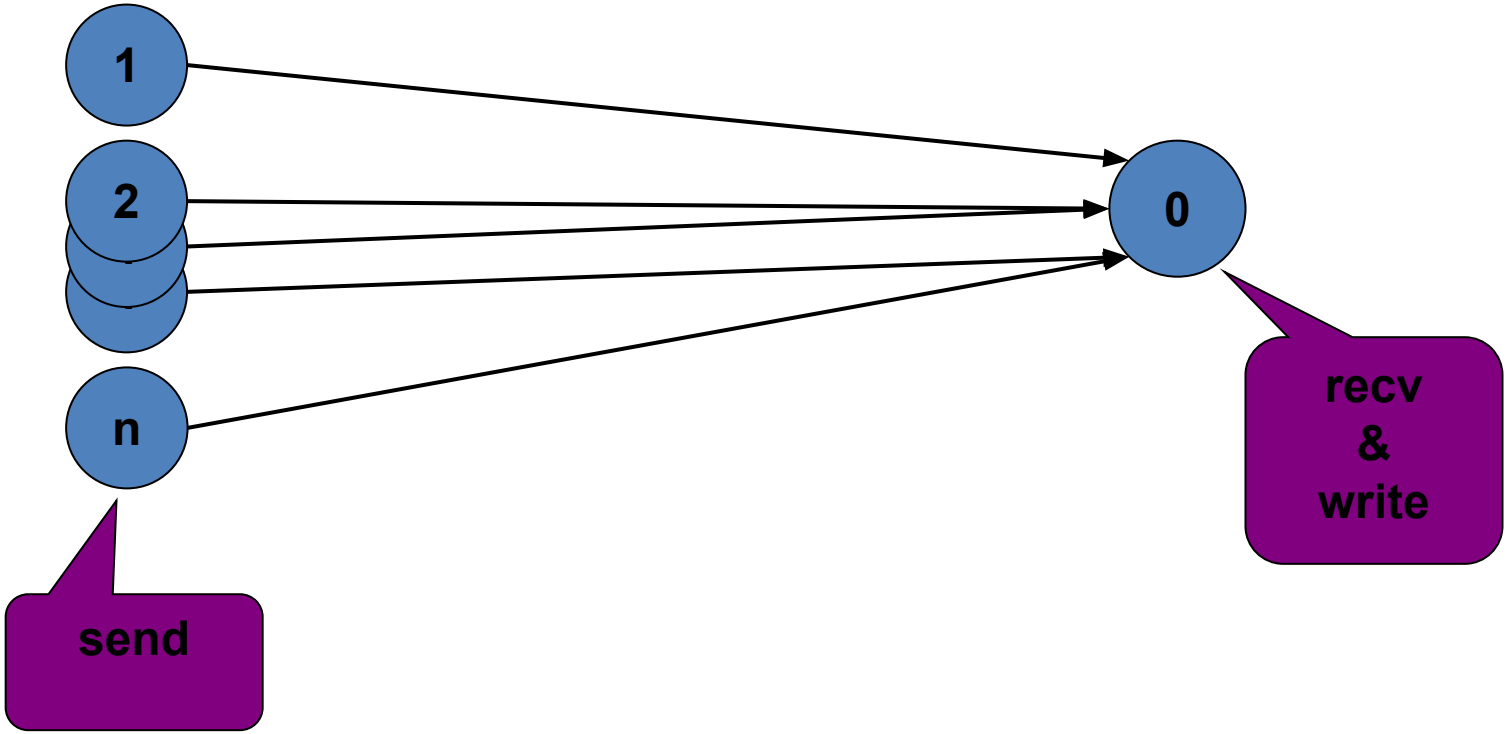
Zadanie 2

POWITANIE (WERSJA 2)

`hello/hello1.c`

Pierwszy program z komunikacją

Zadanie: wyróżniony proces (0) odbiera i wypisuje komunikaty wysyłane przez pozostałe procesy (1,2,…,n) w ramach komunikatora



Powitanie: pliki nagłowkowe

EURO

Pliki nagłówkowe

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

Fortran 90:

    use mpi

Fortran 77:

    include 'mpif.h'
```

Powitanie: inicjalizacja MPI

EURO

Wspólna część programu

```
C:
  int main(int argc, char** argv)  {

        int numranks, src, dest, myrank;
        int tag = 1;
        char mes[50];
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &numranks);
     ...

     MPI_Finalize();

        return 0;
}
```

Powitanie: proces 0

EURO

Część programu wykonywana przez proces odbierający (0)

```
C:
    for (src = 1; src < numranks; src++)  {

          MPI_Recv(mes, 50, MPI_CHAR, src, tag, \

          MPI_COMM_WORLD, &status);

          printf("%s\n", mes);
       }
```

Powitanie: pozostałe procesy 1,…, n-1

EURO

Część programu wykonywana przez pozostałe procesy

```
C:
    sprintf(mes, "Halo, tu proces %d", myrank);

    dest = 0;

    MPI_Send(mes, strlen(mes) + 1, MPI_CHAR, dest, tag, \

    MPI_COMM_WORLD);
```

Powitanie: kompletny program

EURO

```
int main(int argc, char** argv)  {

     int numranks, src, dest, myrank;
     int tag = 1;
     char mes[50];
     MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
     MPI_Comm_size(MPI_COMM_WORLD, &numranks);

    if ( myrank != 0 ) {
          sprintf(mes, "Halo, tu proces %d", myrank);
          dest = 0;
          MPI_Send(mes, strlen(mes) + 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else  {
          for (src = 1; src < numranks; src++) {
             MPI_Recv(mes, 50, MPI_CHAR, src, tag, MPI_COMM_WORLD, &status);
             printf("%s\n", mes);
          }
     }

     MPI_Finalize();
     return 0;
}
```

Uruchomienie w systemie kolejkowym

EURO

- **Definicja zadania dla systemu kolejkowego (skrypt kolejowy SLURM)**

hello.sl:

```
#!/bin/bash -l
#SBATCH --time=00:10:0
#SBATCH --nodes=1
#SBATCH --ntasks-per-node 4
#SBATCH --job-name="hello_mpi"
#SBATCH --output="hello.out"
#SBATCH --error="hello.err"

module load common/mpi/mpich
srun ./a.out
```

```
>> sbatch --account=[..] --reservation=[..] hello.sl
```

Powitanie: kolejność komunikatów

- Załóżmy, że zamiast
- odbierać komunikaty od procesów o numerach 1,…,numranks-1 kolejno
- odbieramy od dowolnego procesu, który w danym momencie nadaje komunikat:

C:

```
    for (src = 1; src < numranks; src++)  {

          MPI_Recv(mes, 50, MPI_CHAR, MPI_ANY_SOURCE, tag, \

          MPI_COMM_WORLD, &status);

          printf("%s\n", mes);
       }
```

Powitanie: podsumowanie

EURO

- Wprowadź zmianę z poprzedniego slajdu i uruchom program kilkakrotnie
  - Co można zaobserwować?
  - Dlaczego tak się dzieje?

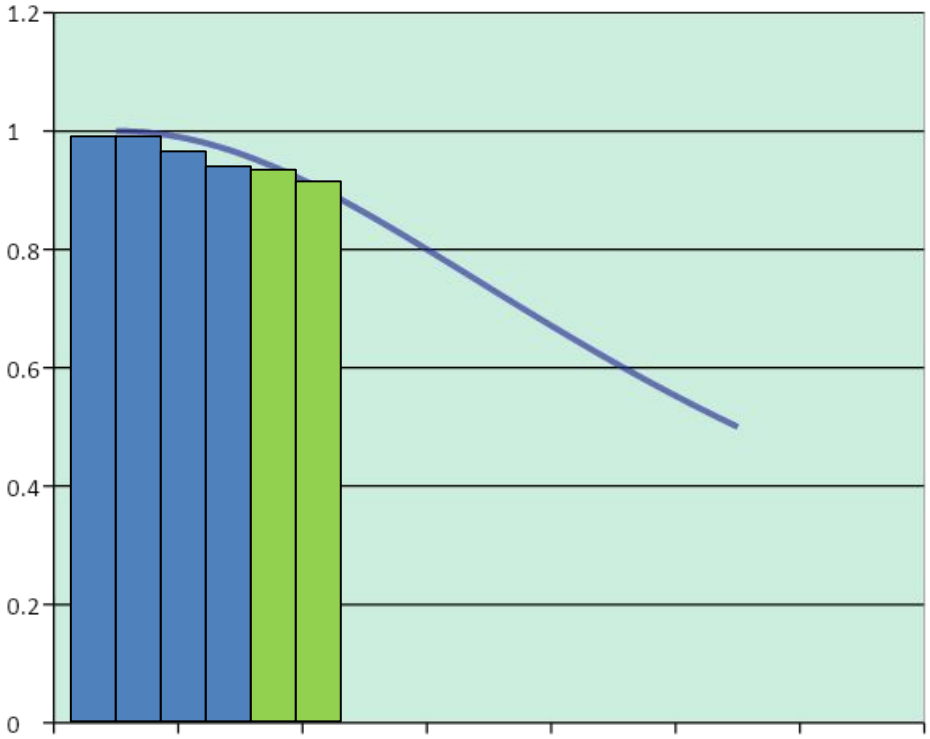
Zadanie 3

KWADRATURA PROSTOKĄTÓW

`pi/pi.c`

Prosta kwadratura (całka numeryczna)

- Całkowanie metodą prostokątów następującego wzoru:



$$\Pi = 4\int_0^1 \frac{1}{1+x^2} \approx \sum_{i=0}^{n} \frac{1}{1+(\frac{i+0.5}{n})^2} \cdot h$$

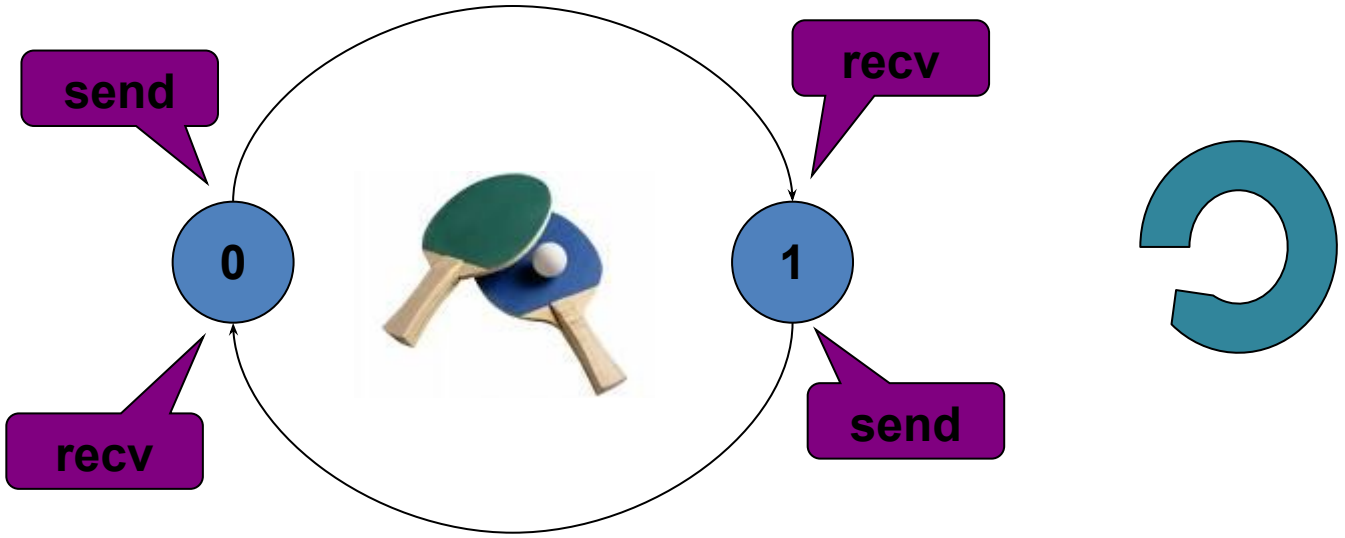
$$f(x) = \frac{1}{1+x^2}$$

Zadanie 4

PING_PONG

Program ping pong

- **Zadanie:** dwa procesy komunikują się naprzemiennie. Pierwszy proces (0) wysyła wiadomość (np. liczbę 1) do drugiego procesu (1). Następnie wiadomość jest odsyłana z powrotem. Schemat powtarzamy kilkakrotnie.



Program ping pong (2)

EURO

- Użyj trybu MPI_Ssend

C:

```
    MPI_Ssend(&snd_buf, 1, MPI_INT, dest, ping_tag, \
        MPI_COMM_WORLD);

    MPI_Recv(&rcv_buf, 1, MPI_INT, source, pong_tag, \
        MPI_COMM_WORLD, &stat);
```