Uniwersytet Wrocławski

# Programowanie Równoległe

## Message Passing Interface

(część 2)

Maciej Matyka
Instytut Fizyki Teoretycznej

Uniwersytet Wrocławski

# Strona WWW kursu

## Programowanie Rownolegle 2014/2015

### Zbigniew Koza, Maciej Matyka

**Lista wykladow:**

- Wyklad 1, (ZK) w
- Wyklad 2, (ZK) w
- Wyklad 3, (ZK) w
- Wyklad 4, (ZK) w
- Wyklad 5, (ZK) w
- Wyklad 6, (ZK) w
- Wyklad 7, MPI 1, 2014-11-18 (MM) mpi1.pdf mpi-lista1.pdf
- Wyklad 8, MPI 1, 2014-11-25 (MM) mpi1.pdf mpi-lista1.pdf

---

- Wyklad 8, MPI 2 - praktyczne (MM)
- Wyklad 9, GPU (MM)
- Wyklad 10, GPU (MM)
- Wyklad 11, GPU (MM)
- Wyklad 12, GPU (MM)

www.ift.uni.wroc.pl/~koma/pr/index.html

Uniwersytet Wrocławski

# MPI, wykład 2.

Plan:

- komunikacja kolektywna (broadcast, scatter etc.)
- Operatory MPI_Op

# Rodzaje komunikacji - powtórka

- Blokująca – nadawca wysyła komunikat i czeka na potwierdzenie odbioru
- Nieblokująca – nadawca wysyła komunikat i nie oczekuje na potwierdzenie (może je odebrać w dowolnym momencie)

Wprowadzone MPI_Send i MPI_Recv są blokujące.

# Tryby wysyłania - powtórka

- synchroniczny
  - Wysłanie i odebranie są zsynchronizowane. W przypadku blokującym nadawca po zgłoszeniu wiadomości czeka aż odbiorca zacznie odbieranie, wtedy kontynuuje pracę. Uwaga: to nie oznacza, że odbiorca skończył odbierać wiadomość.
- gotowości
  - Wymagane jest, by proces odbierający zgłosił wcześniej wiadomość do odebrania. W przypadku gdy tego nie zrobił zachowanie jest nieokreślone. Ten tryb zwiększa wydajność komunikacji.
- buforowany
  - Komunikacja przez bufor, zgłoszenie jest równoważne przesłaniu danych do bufora tymczasowego z którego jest odbierany w dowolnym momencie.

# Tryby wysyłania - powtórka

| Tryb | Blokująca | Nieblokująca |
|---|---|---|
| Standardowy | MPI_Send | MPI_Isend |
| Synchroniczny | MPI_Ssend | MPI_Issend |
| Gotowości | MPI_Rsend | MPI_Irsend |
| Buforowany | MPI_Bsend | MPI_Ibsend |

Uwaga: Nieblokujące MPI_Isend oraz MPI_Issend różnią się ze względu na sposób działania MPI_Wait/MPI_Test

MPI_Isend – kończy działanie gdy wiadomość np. w buforze
MPI_Issend – kończy się gdy wywołane „Receive”

Szczegółowy opis:
http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

# Komunikacja kolektywna

Komunikacja między grupami procesorów z użyciem wbudowanych funkcji MPI:

- są zoptymalizowane
- bardziej czytelne, łatwiejsze do debugowania

Możliwa implementacja ręczna (MPI_*Send/MPI_*Recv).

# Komunikacja kolektywna

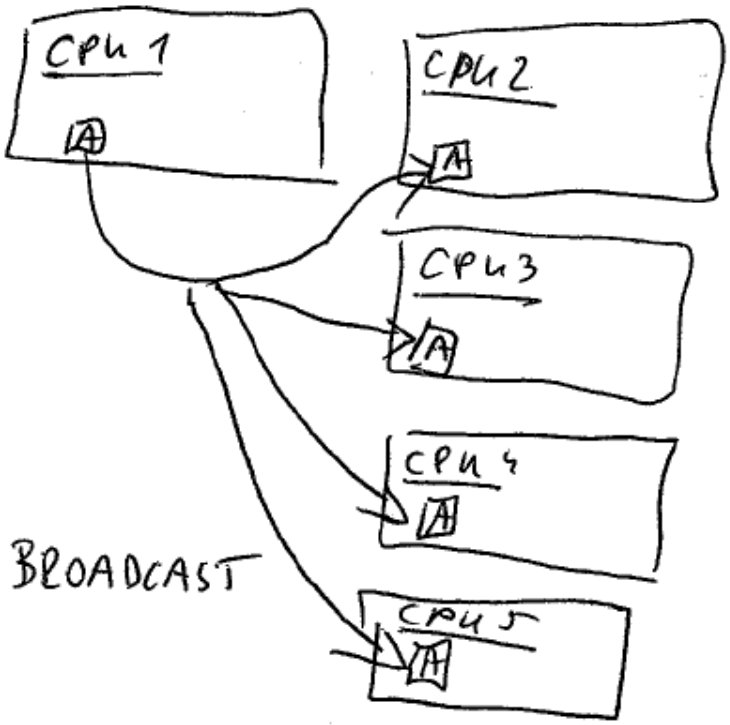
Dotychczas poznaliśmy komunikację:

- jeden do jednego (*Send/*Receive)

Dane mogą być rozsyłane pomiędzy grupami procesów:

- kopia jeden do wielu (broadcast)
- rozszczepienie jeden do wielu (scatter)
- zebranie wielu do jednego (gather)
- redukcja wielu do jednego (reduction)

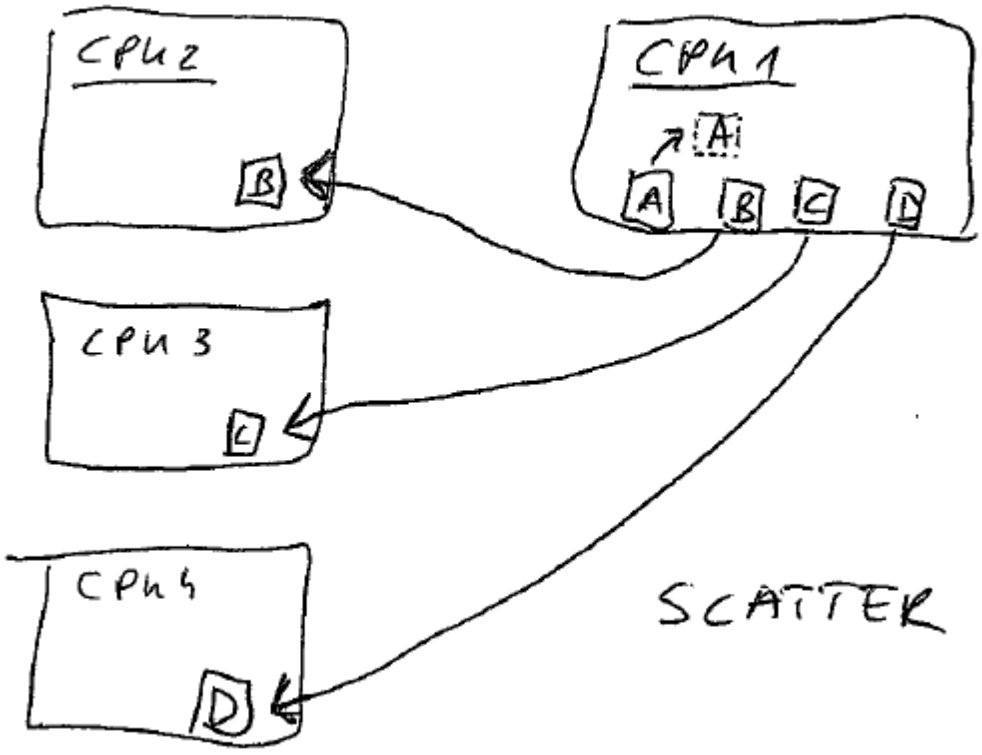
# Broadcast




- Wysłanie kopii tych danych z procesu źródłowego do grupy procesów:

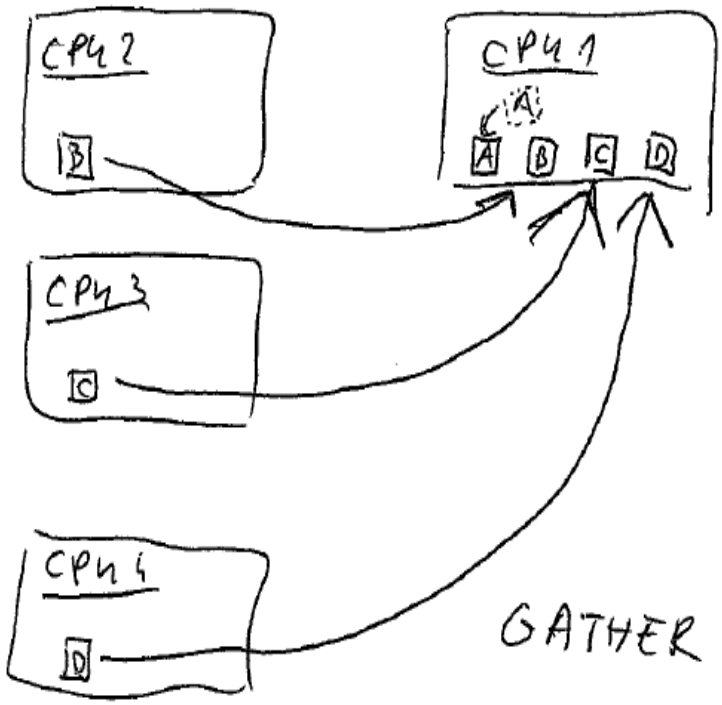
# Scatter

- Rozszczepienie zestawu danych na grupę procesów:

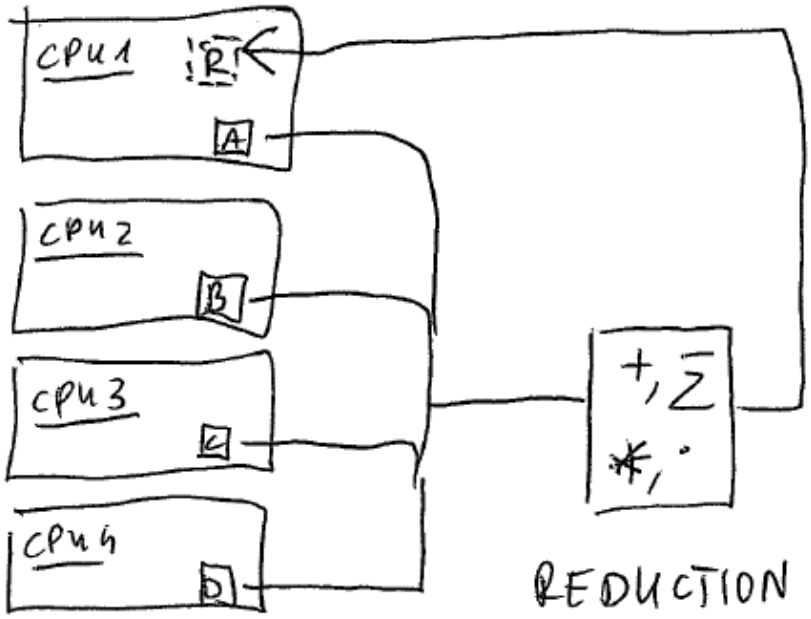


# Gather

- Zebranie danych z grupy procesów przez jeden proces:



# Reduction




- Zebranie danych z grupy procesów z zadeklarowaną operacją. Przykłady: suma, minimum, maksimum etc.

# Broadcast

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

- rank – ID nadawcy
- brak tag-u wiadomości
- wysyła również do siebie
- nie wymaga Recv (wywołanie = rozesłanie + odebranie)
- Przykład (za mpi-report):

4.4.1 Example using MPI_BCAST

**Example 4.1** Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast( array, 100, MPI_INT, root, comm);
```

# Redukcja

```
int MPI_Reduce( void* send_buffer,
                void* recv_buffer,
                int count,
                MPI_Datatype datatype,
                MPI_Op operation,
                int rank,
                MPI_Comm comm )
```

} ta sama wielkość u nadawcy

- void* send_buffer - bufor źródłowy
- void* recv_buffer - bufor wyniku (ważne w procesie głównym)
- rank – ID procesu głównego
- MPI_Op – flaga operacji

| | |
|---|---|
| MPI_MAX | maximum |
| MPI_MIN | minimum |
| MPI_SUM | sum |
| MPI_PROD | product |
| MPI_LAND | logical and |
| MPI_BAND | bit-wise and |
| MPI_LOR | logical or |
| MPI_BOR | bit-wise or |
| MPI_LXOR | logical exclusive or (xor) |
| MPI_BXOR | bit-wise exclusive or (xor) |
| MPI_MAXLOC | max value and location |
| MPI_MINLOC | min value and location |

# Reduce, Przykład 1

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main( int argc, char **argv )
{

  int send, recv=0;
  int myrank;

  MPI_Init( &argc, &argv );

  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

  // każdy proces wstawia inną daną
  send = myrank;

  // suma
  MPI_Reduce(&send, &recv, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  if(myrank==0)
    printf("Suma: %d\n", recv);

  MPI_Finalize();
  return 0;
}
```

```
maq@zero ~/pr $
maq@zero ~/pr $
maq@zero ~/pr $ gcc reduce0.c -lmpi
maq@zero ~/pr $ mpirun -np 2 ./a.out
Suma: 1
maq@zero ~/pr $ mpirun -np 4 ./a.out
Suma: 6
maq@zero ~/pr $ mpirun -np 8 ./a.out
Suma: 28
maq@zero ~/pr $ mpirun -np 64 ./a.out
Suma: 2016
maq@zero ~/pr $
```

# Reduce, Przykład 2

Wykonajmy działanie MPI_SUM na wektorach danych:


Wyniki liczone niezależnie dla każdej składowej wektora.

Uniwersytet Wrocławski

# Reduce, Przykład 2

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main( int argc, char **argv )
{

  int *send, *recv;
  int myrank, size;
  int i;

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

  send = calloc( size, sizeof(int) );  // allokacja bufora wielkości = ilości
     procesów
  recv = calloc( size, sizeof(int) );

  for(i=0; i<size; i++)                // np. 1,2,3,4 dla np=4
    send[i] = i+1;

  MPI_Reduce(send, recv, size, MPI_INT, MPI_SUM, 0,

  if(myrank==0)
  for(i=0 ;  i < size;  i++)
    printf(„ Suma[%d]: %d\n ",  i,  recv[i]);

  MPI_Finalize();
  return 0;
}
```

```
maq@zero ~/pr $ mpirun -np 4 ./a.out
Suma[0]: 4
Suma[1]: 8
Suma[2]: 12
Suma[3]: 16
maq@zero ~/pr $ mpirun -np 2 ./a.out
Suma[0]: 2
Suma[1]: 4
maq@zero ~/pr $
```

# Reduce, Uwagi

- dopuszcza różną kolejność działań (float!)
- dopuszcza własne typu danych (MPI_Datatype)
- Pozwala definiować własne operatory (MPI_Op_create)
- MPI_IN_PLACE – tę flagę można przekazać jako „sendbuf” w procesie głównym (optymalizacja)
- Przykład: suma ID procesów

# MPI_Op_create

```
int MPI_Op_create(  MPI_User_function *function,
                    int commute,
                                  MPI_Op *op)
```

- function – adres funkcji wykonującej redukcję na wektorach danych
- commute
  - true – operacja przemienna i łączna
  - false – operacja nieprzemienna (dane ułożone w kolejności id procesów)
- MPI_Op – uchwyt nowego operatora

- MPI_Reduce, MPI_Reduceall, MPI_Reduce_scatter

# Prototyp funkcji:

```
typedef void MPI_User_function(
                    void *invec,
                    void *inoutvec,
                    int *len,
                    MPI_Datatype *datatype);
```

- invec – wektor danych do redukcji (wejście)
- inoutvec – wektor danych do redukcji (we./wyj.)
- len – długość danych wejściowych
- datatype – typ danych w wektorach invec/inoutvec

- Funkcja wykonuje:

  inoutvec[i] = invec[i] * inoutvec[i], gdzie i =0..len-1

Uniwersytet Wrocławski

# MPI_Op_create

```
int MPI_Op_free( MPI_Op *op)
```

- zwolnienie zasobów operatora

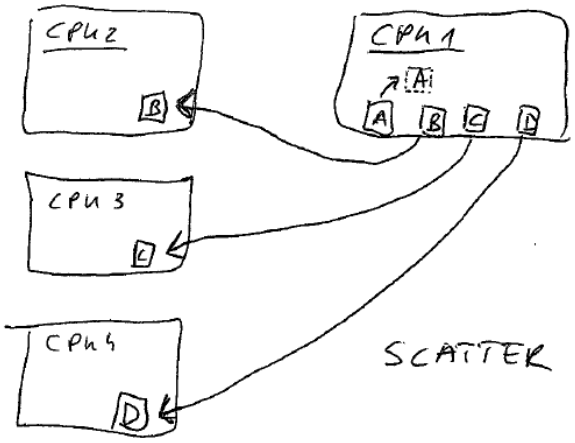
# MPI_Op_create - przykład

```
void mnozenief( double *in,  double *inout, int *len, MPI_Datatype *dptr )
{
      int i;
      double c;

      for (i=0; i< *len; ++i)
      {
        c = inout[i] * in[i];
        inout[i] = c;
      }
}



      ...
      int a[100], answer[100];

      MPI_Op myOp;

      MPI_Op_create( mnozenief, 1, &myOp );

      MPI_Reduce( a, answer, 100, MPI_DOUBLE, myOp, 0, comm );
      ...
```

- Przykład zastosowania: algebra liczb zespolonych (ale wymaga zdefiniowania nowego typu danych)

# Scatter

```
int MPI_Scatter (
            void* send_buffer,
            int send_count,
                  MPI_datatype send_type,
                  void* recv_buffer,
                  int recv_count,
                  MPI_Datatype recv_type,
                  int rank,
                  MPI_Comm comm )
```




- recv_buffer – ważne tylko dla procesu głównego
- taka sama preambuła dla MPI_Gather
- send_count – ilość elementów w buforze do wysłania
- recv_count – ilość elementów w buforze do odebrania

# Wariacje

- MPI_Allreduce – wykonuje redukcję i rozsyła wynik do wszystkich procesów (broadcast)
- MPI_Allgather – zbiera dane i rozsyła wynik do wszystkich procesów (broadcast)

(więcej podobnych przykładów (np.. MPI_Reduce_scatter etc.) w dokumentacji MPI mpi-report)

Uniwersytet Wrocławski

# Literatura

- http://people.sc.fsu.edu/~jburkardt/pdf/mpi_course.pdf (polecam!)
- MPI: A Message-Passing Interface Standard, Version 1.3
- http://aragorn.pb.bialystok.pl/~wkwedlo/PC3.pdf

Uniwersytet Wrocławski

# SLAJDY ODRZUCONE

- MPI_Struct

# MPI_Send

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

treść

koperta

- MPI_Datatype struktura określająca typ przesyłanych danych

# Typy danych w MPI

To ważne żeby używać wbudowanych typów do komunikacji (komunikacja między różnymi systemami).

| MPI datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | |
| MPI_PACKED | |

MPI_BYTE – 8 bitów, ogólny typ dla dowolnych danych
MPI_PACKED – typ danych jawnie scalonych (nie: „spakowanych”)

- MPI_TYPE_CONTIGUOUS
- MPI_TYPE_VECTOR
- MPI_TYPE_HVECTOR
- MPI_TYPE_INDEXED
- MPI_TYPE_HINDEXED