Programowanie współbieżne... (6)

Andrzej Baran 2010/11

LINK: http://kft.umcs.lublin.pl/baran/prir/index.html

Powtórzenie. Proste operacje MPI

`MPI_send, MPI_recv,...`

Send-receive przykład

```
PROGRAM simple_send_and_receive
  INCLUDE 'mpif.h'
  INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
  REAL a(100)

  ! Initialize MPI:
  call MPI_INIT(ierr)

  ! Get my rank:
  call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

  ! Process 0 sends, process 1 receives:
  if( myrank.eq.0 )then
     call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
  else if ( myrank.eq.1 )then
     call MPI_RECV( a, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status, ierr )
  end if

  ! Terminate MPI:
  call MPI_FINALIZE(ierr)
END
```

Prosty impas – deadlock

```
PROGRAM simple_deadlock
  INCLUDE 'mpif.h'
  INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
  REAL a(100), b(100)
  ! Initialize MPI:
  call MPI_INIT(ierr)
  ! Get my rank:
  call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
  ! Process 0 receives and sends; same for process 1
  if( myrank.eq.0 )then
    call MPI_RECV( b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, status, ierr )
    call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
  else if ( myrank.eq.1 )then
    call MPI_RECV( b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status, ierr )
    call MPI_SEND( a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr)
  Endif
  ! Terminate MPI:
```

Bezpieczna wymiana

```
PROGRAM safe_exchange
  INCLUDE 'mpif.h'
  INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
  REAL a(100), b(100)
  call MPI_INIT(ierr)    ! Initialize MPI
  call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)   ! Get my rank
  ! Process 0 receives and sends; process 1 sends and receives
  if( myrank.eq.0 )then
     call MPI_RECV( b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, status, ierr )
     call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
  else if ( myrank.eq.1 )then
     call MPI_SEND( a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr )
     call MPI_RECV( b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status, ierr)
  endif

  ! Terminate MPI:
  call MPI_FINALIZE(ierr)
END
```

Proste algorytmy paralelizacji

- Opieramy się tytaj na pracy: *Yukiya Aoyama, Jun Nakano: Practical MPI programming. RS/6000 SP, IBM* (patrz: http://www.redbooks.ibm.com)

- W zależności od złożoności pętli ich paralelizacja może być banalna lub bardzo złożona

- Uwaga: Jest w sprzedaży książka: *Zbigniew Czech: Wprowadzemie do obliczeń równoległych. PWN, Warszawa, 2010*.

Proste algorytmy...

- Paralelizacja pętli `do`. Będziemy zakładać, że granice pętli `do` są w jakiś sposób obliczone i będziemy je nazywać `ista` (wartość początkowa) oraz `iend` (wartość końcowa)

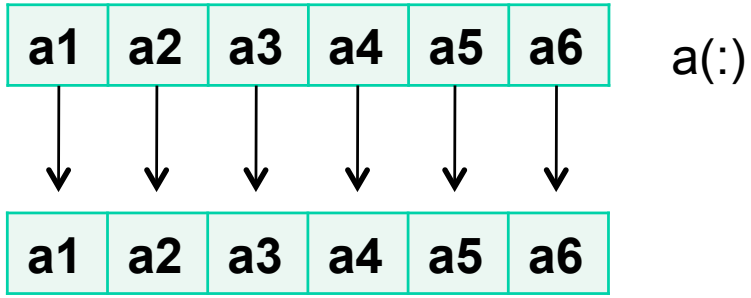
(Zadanie 1. Napisać program obliczania granic pętli `do` dla p procesorów jeśli naturalne granice pętli są 1 i n (Dekompozycja blokowa).

Zadanie 2. W cyklicznym (karuzelowym) przydzielaniu zadań obciąża się kolejno procesy. Gdy ostatni proces jest obciążony (ma już zadanie) kolejne zadanie przydziela się procesowi pierwszemu. Zapisać to dla pętli o granicach 1, n)

Proste algorytmy...

- Prosta pętla do

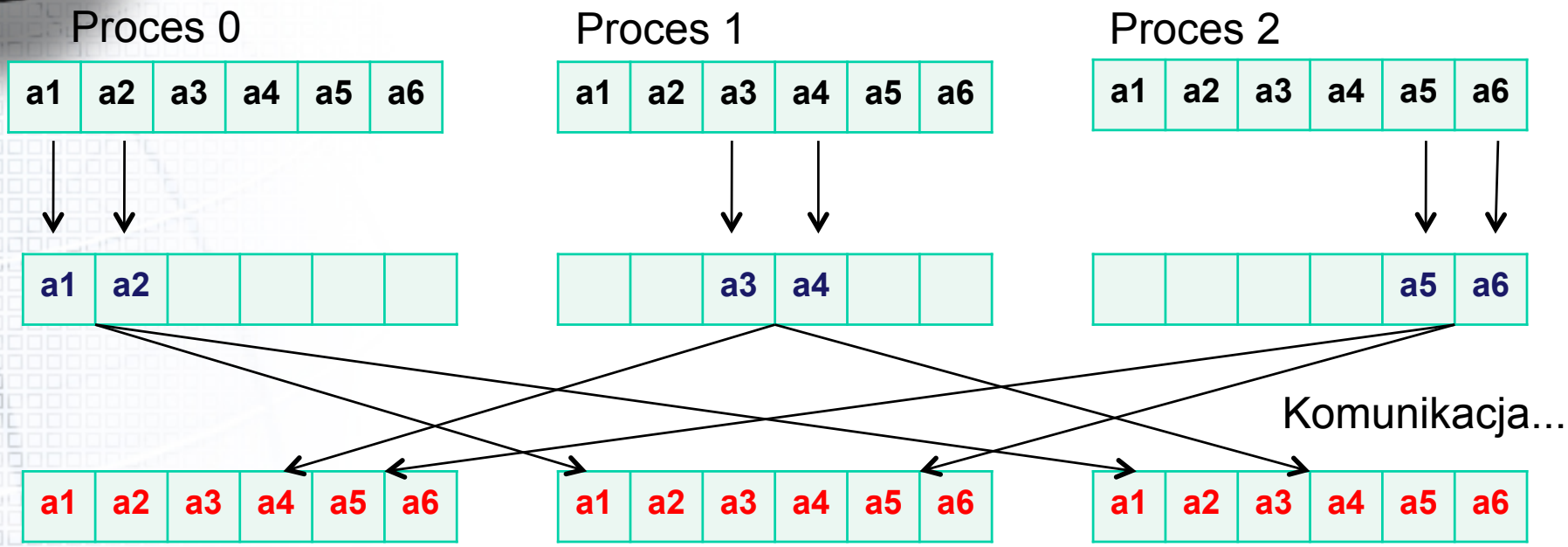
```
do i=1,6
  ...
  a(i) = ...
  ...
end do
...= a(j) + ...
```



- Wersja równoległa

```
do i=ista, iend
   ...
   a(i) = ...
   ...
end do
call sync()
! O sync: dalej
... = a(j) + ...
```

do – schemat w. równoległej



Po obliczeniach w przydzielonych rejonach procesy przesyłają
Do siebie nawzajem otrzymane wyniki... (kolory → wyniki)

do – inny przykład

- W. sekwencyjna:

```
...
REAL a(9), b(9)
...
DO i = 1, 9
   a(i) = i
END DO
DO i = 1, 9
   b(i) = b(i) * a(1)
END DO
...
```

mnożenie b() przez element a(1)

- Wersja równoległa:

```
   ...
REAL a(9), b(9)
   ...
DO i = ista, iend
   a(i) = i
END DO
CALL MPI_BCAST(a(1), 1, MPI_REAL,
          0, MPI_COMM_WORLD, ierr)
DO i = ista, iend
   b(i) = b(i) * a(1)
END DO
...
```

do – metoda potokowa

Mamy pętlę *do* :

```
do i=1, n
   x(i) = x(i-1) + 1.
end do
```

W jawnej postaci:

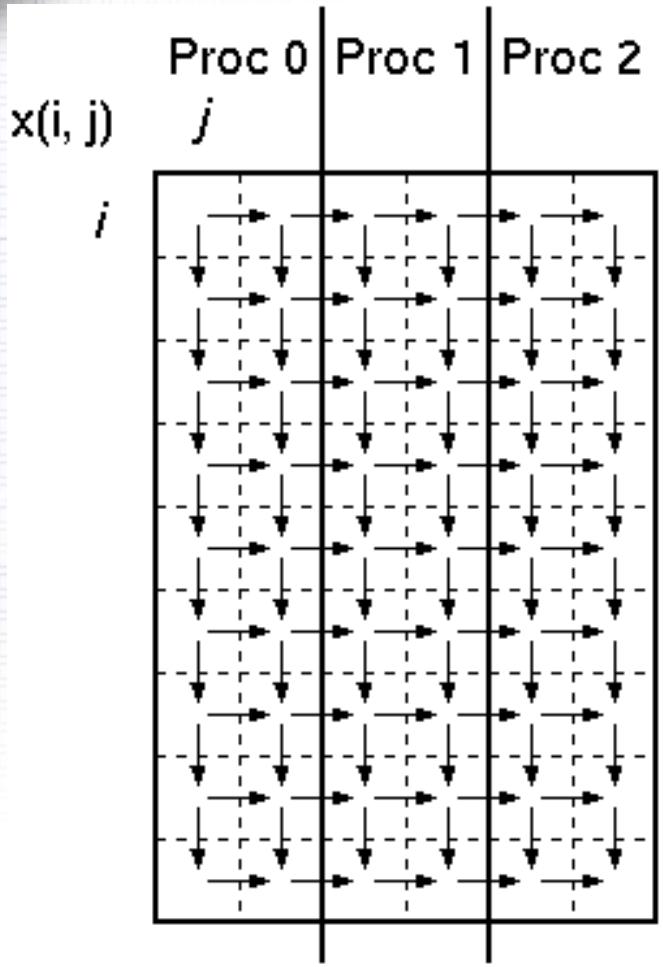
```
x(1) = x(0) + 1   (iter 1)
x(2) = x(1) + 1   (iter 2)
x(3) = x(2) + 1   (iter 3)
...
```

Każda iteracja musi być wykonana
po następnej bo jest od niej zależna

Podobnie, w przypadku forward elimination i backward substitution:

```
dimension x(0:mx,0:my)
...
do j=1, my
   do i=1, mx
      x(i,j) = x(i,j) + &
         x(i-1,j) + x(i,j-1)
    end do
end do
...
```

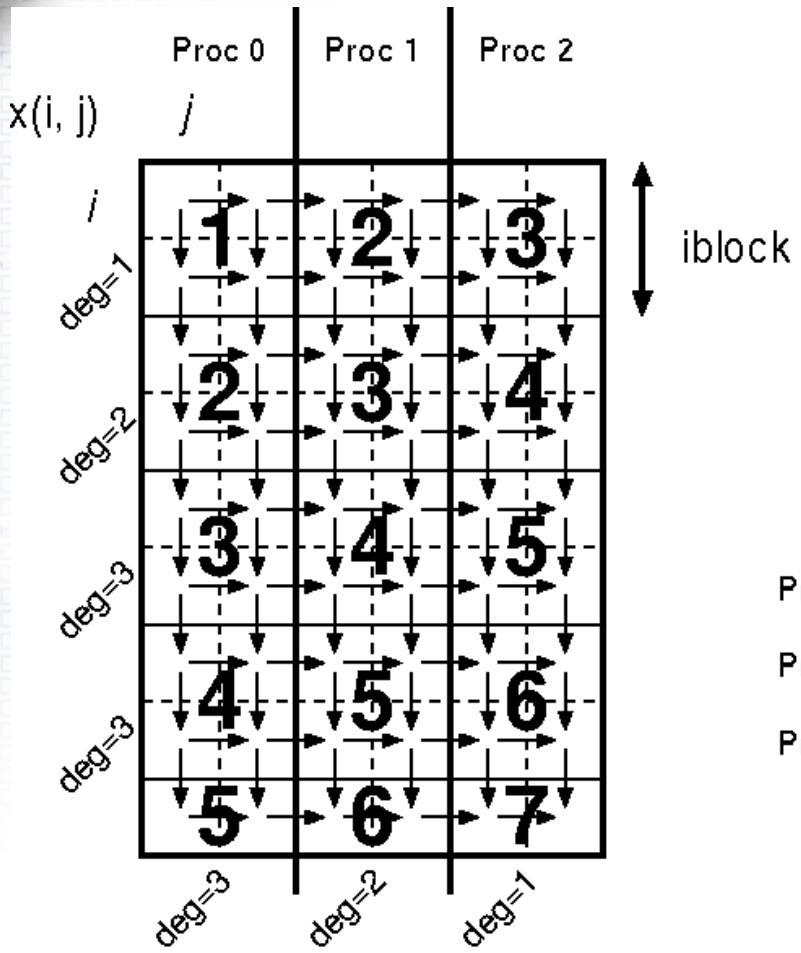
do – metoda potokowa



Zależności między elementami są złożone i problem jest trudny do paralizacji.

Pomaga w takim przypadku metoda zwana potokową

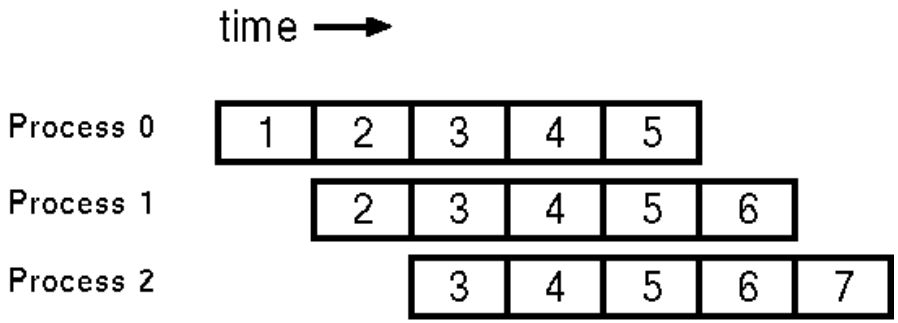
do – metoda potokowa



Bloki i dekompozycja danych

(Aoyama, Nakano)

Podział przetwarzania bloków



do – metoda potokowa

```
PROGRAM mainp
INCLUDE 'mpif.h'
PARAMETER (mx = ..., my = ...)
DIMENSION x(0:mx,0:my)
INTEGER istatus(MPI_STATUS_SIZE)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL para_range(1, my, nprocs, myrank, jsta, jend)
inext = myrank + 1
IF (inext == nprocs) inext = MPI_PROC_NULL
iprev = myrank - 1
IF (iprev == -1) iprev = MPI_PROC_NULL
iblock = 2
...
```

do – metoda potokowa

```
DO ii = 1, mx, iblock
   iblklen = MIN(iblock, mx - ii + 1)
   CALL MPI_IRECV(x(ii, jsta - 1), iblklen, MPI_REAL, &
                  iprev, 1, MPI_COMM_WORLD, ireqr, ierr)
   CALL MPI_WAIT(ireqr, istatus, ierr)
   DO j = jsta, jend
      DO i = ii, ii + iblklen - 1
         x(i,j) = x(i,j) + x(i-1,j) + x(i,j-1)
      END DO
   END DO
   CALL MPI_ISEND(x(ii,jend), iblklen, MPI_REAL, &
                  inext, 1, MPI_COMM_WORLD, ireqs, ierr)
   CALL MPI_WAIT(ireqs, istatus, ierr)
END DO
...
```

do – metoda sum prefiksowych

(Patrz: Aoyama, Nakano; Czech)

```
Program main
Parameter (n=...)
real a(0:n), b(n)
...
do i=1, n
   a(i) = a(i-1) + b(i)
end do
...
```

Zależności:

a(0) → a(1) → a(2) →...→ a(n)

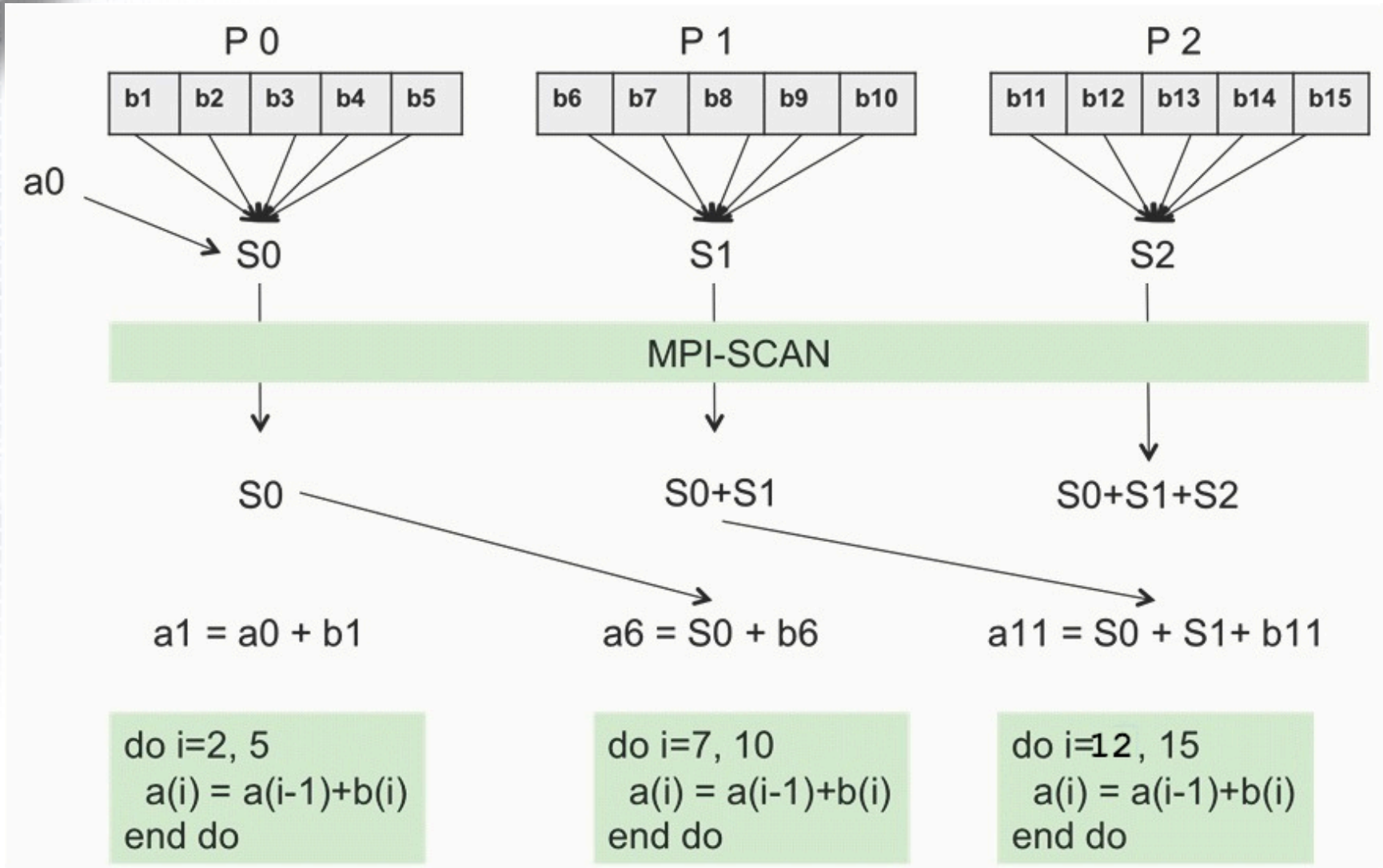
Jawnie:

```
a(1) = a(0) + b(1)
a(2) = a(0) + b(1) + b(2)
a(3) = a(0) + b(1) + b(2) + b(3)
...
a(n) = a(0) + b(1) + ... +  b(n)
```

Ogólniej:

```
a(i)=a(i-1) op b(i)
```

do – metoda sum prefiksowych



do – metoda sum prefiksowych

```
PROGRAM main
INCLUDE 'mpif.h'
PARAMETER (n = ...)
REAL a(0:n), b(n)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL range(1, n, nprocs, myrank, ista, iend)
...
sum = 0.0
DO i = ista, iend
   sum = sum + b(i)
ENDDO
IF (myrank == 0) THEN
   sum = sum + a(0)
ENDIF
```

do – metoda sum prefiksowych

```
!...ciąg dalszy...

CALL MPI_SCAN(sum, ssum, 1, MPI_REAL, MPI_SUM, &
                                 MPI_COMM_WORLD, ierr)
a(ista) = b(ista) + ssum - sum
IF (myrank == 0) THEN
  a(ista) = a(ista) + a(0)
END IF
DO i = ista+1, iend
  a(i) = a(i-1) + b(i)
END DO
...
```

Problemy...?

Definiowane typy danych

- Przypomnienie. Typy proste w MPI (Fortran):
  - MPI_CHARACTER
  - MPI_INTEGER
  - MPI_REAL
  - MPI_DOUBLE_PRECISION
  - MPI_COMPLEX
  - MPI_DOUBLE_COMPLEX
  - MPI_LOGICAL
  - MPI_BYTE
  - MPI_PACKED

Definiowane typy danych

- Przesyłanie danych może w wielu wypadkach zająć wiele czasu... dane nieciągłe, rozproszone, dane różnych typów
- Można grupować dane w celu ich przesyłania – to może być czasochłonne – wybieranie, kopiowanie do jednolitego, ciągłego bufora
- Można próbować zamieniać typy danych by uzyskać jednorodność danych przesyłąnych – niebezpieczne (różne realizacje w różnych systemach)

Definiowane typy danych

- MPI pozwala definiować własne typy danych:
  - sąsiadujące, przyległe (contiguous)
  - wektorowe
  - indeksowane
  - strukturalne

Definiowane typy danych

- **MPI_TYPE_CONTIGUOUS (count, oldtype, newtype, ierr)**

Konstruktor ten tworzy **count** kopii istniejącego typu danych

- **MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype, ierr)**

Podobnie jak wyżej, ale pozwala na regularne przerwy w przemieszczeniach (stride)

- **MPI_TYPE_INDEXED (count, blocklens(), offsets(), old_type, newtype, ierr)**

Podaje się tu długości bloków i dodatkowo tablicę przesunięć

Definiowane typy danych

- **MPI_TYPE_STRUCT (count, blocklens(), offset(), old_types, newtype, ierr)**

Nowy typ specyfikowany jest przez podanie kompletnego odwzorowania typów składowych

- Odmiany procedur tworzenia typów:
  - **MPI_Type_hvector**
  - **MPI_Type_hindexed**

Definiowane typy danych

- Inne procedury, *informacyjne:*
  - **MPI_TYPE_EXTENT (datatype,extent,ierr)**
  - **MPI_TYPE_COMMIT (datatype,ierr)** – przekazuje systemowi informacje o typie; zwraca typ do systemu
  - **MPI_TYPE_FREE (datatype,ierr)** – zwalnia typ
- C, C++
  - podobnie jak wyżej; funkcje zwracające **ierr**

Typy, przykłady

- `call MPI_Type_contiguous(2, MPI_DOUBLE, MPI_2D_POINT, ierr);`

- `call MPI_Type_contiguous(3, MPI_DOUBLE, MPI_3D_POINT, ierr);`

Definicja typu MPI_2D_POINT i typu MPI_3D_POINT na podstawie typu MPI_DOUBLE. Na dane tych typów składają się punkty (dwie lub trzy liczby typu mpi_double).

Typy, przykłady

```
CALL MPI_TYPE_VECTOR(M, N, NN, MPI_DOUBLE, &
     MY_MPI_TYPE, IERROR)
CALL MPI_TYPE_COMMIT(MY_MPI_TYPE,IERROR)
CALL MPI_SEND(A(K,L), 1, MY_MPI_TYPE, &
     DEST, TAG, MPI_COMM_WORLD, IERROR)
CALL MPI_TYPE_FREE(MY_MPI_TYPE, IERROR)
```

Kolejne bloki, każdy o długości N są kolumnami podmacierzy. Mamy M bloków. Kolumny zaczynają się co NN elementów – deklarowany wymiar kolumny macierzy zawierającej podmacierz. Po podaniu definicji typ jest zgłaszany w systemie (commit). Po wysłaniu danych (jedna porcja) typ zostaje zwolniony.

Typy definiowane, przykłady

```
LENA(1) = 1   !  długość składowej definiowanego typu
CALL MPI_ADDRESS(X(1), LOCA(1), IERROR)
TYPA(1) = MPI_DOUBLE    !  typ składowy
LENA(2) = 1   !  długość składowej drugiej definiowanego typu
CALL MPI_ADDRESS(Y(1), LOCA(2), IERROR)
TYPA(2) = MPI_DOUBLE   ! typ składowy drugiej komponenty definiowanego typu
LENA(3) = 1   !  długość trzeciej składowej
CALL MPI_ADDRESS(Z(1), LOCA(3), IERROR)
TYPA(3) = MPI_DOUBLE    ! ...
LENA(4) = 1
CALL MPI_ADDRESS(X(1), LOCA(4), IERROR)
TYPA(4) = MPI_LB
LENA(5) = 1
CALL MPI_ADDRESS(X(2), LOCA(5), IERROR)
TYPA(5) = MPI_UB
CALL MPI_TYPE_STRUCT(5, LENA, LOCA, TYPA, MY_TYPE, IERROR)
CALL MPI_TYPE_COMMIT(MY_TYPE,IERROR)
CALL MPI_SEND(MPI_BOTTOM, N, MY_TYPE, DEST, TAG, MPI_COMM_WORLD,IERROR)
CALL MPI_TYPE_FREE(MY_TYPE,IERROR)
```

Problemy...?

Komunikatory

- MPI pozwala tworzyć zbiory procesorów – grupy, komunikatory, zależnie od potrzeb ...
- Cel tworzenia grup procesów
  - pozwalają organizować zadania
  - umożliwiają komunikację kolektywną w podzbiorach zadań powiązanych ze sobą
  - dostarczają bazy do wykorzystania szczególnej topologii zadań
  - zapewniają bezpieczną komunikację

Komunikatory

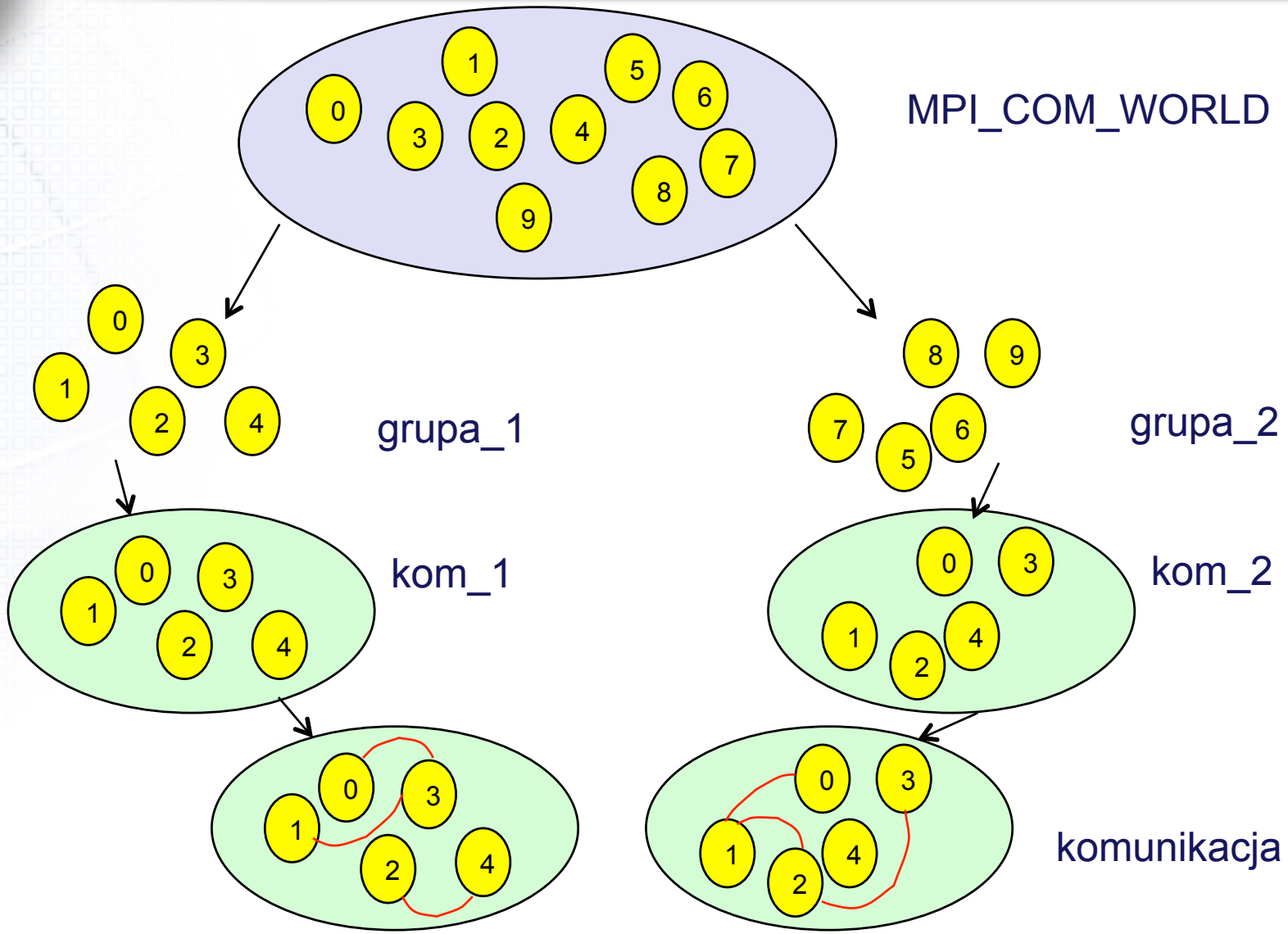
CECHY

- Grupy/komunikatory są obiektami dynamicznymi; są tworzone i rozwiązywane w czasie działania programu (zależnie od sytuacji)
- Procesy mogą należeć do różnych grup; posiadają jednoznaczny identyfikator w grupie
- MPI dostarcza ok. 40 procedur do pracy związanych z grupowaniem procesów

Komunikatory

- Typowa procedura
  - Utworzenie uchwytu grupy (handle) z komunikatora **`MPI_COMM_WORLD`** z użyciem **`MPI_Comm_group`**
  - Utworzenie nowej grupy procesów w postaci podzbioru grupy globalnej z pomocą **`MPI_Group_incl`**
  - Utworzenie nowego komunikatora nowej grupy z pomocą **`MPI_Comm_create`**
  - Wyznaczenie nowej rangi w nowym komunikatorze przez wywołanie **`MPI_Comm_rank`**
  - Komunikacja w nowym komunikatorze z użyciem dowolnych procedur MPI
  - Opcjonalne zwalnienie nowego komunikatora i grupy przez wywołanie **`MPI_Comm_free`** and **`MPI_Group_free`**

Komunikatory



Komunikatory, przykład

```
program group  ! L
  include 'mpif.h'
  integer, parameter :: NPROCS=8
  integer rank, new_rank, sendbuf, recvbuf, numtasks
  integer ranks1(4), ranks2(4), ierr
  integer orig_group, new_group, new_comm
  data ranks1 /0, 1, 2, 3/, ranks2 /4, 5, 6, 7/
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
  if (numtasks .ne. NPROCS) then
     print *, 'Nalezy podac MPROCS= ',NPROCS,' Koniec.'
     call MPI_FINALIZE(ierr)
     stop
  endif
  sendbuf = rank
  ! Wydzielenie uchwytu grupy
  call MPI_COMM_GROUP(MPI_COMM_WORLD, orig_group, ierr)
  ! Podział zadania na dwie grupy wg. rangi
  if (rank .lt. NPROCS/2) then
     call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks1, new_group, ierr)
  else
     call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks2, new_group, ierr)
  endif
  call MPI_COMM_CREATE(MPI_COMM_WORLD, new_group, new_comm, ierr)
  call MPI_ALLREDUCE(sendbuf, recvbuf, 1, MPI_INTEGER, MPI_SUM, new_comm, ierr)
  call MPI_GROUP_RANK(new_group, new_rank, ierr)
  print *, 'rank= ',rank,' newrank= ',new_rank,' recvbuf= ', recvbuf
  call MPI_FINALIZE(ierr)
end program group
```

Komunikatory. Zadanie.

- Uruchomić program `group z` poprzedniej strony

Komunikatory

- Inne procedury
  - MPI_COMM_GROUP
  - MPI_GROUP_INCL
  - MPI_GROUP_EXCL
  - MPI_GROUP_RANK
  - MPI_GROUP_FREE
  - MPI_COMM_CREATE
  - MPI_COMM_SPLIT

Komunikatory

- Topologie wirtualne ...
  - MPI_CART_CREATE
  - MPI_CART_COORDS
  - MPI_CART_RANK
  - MPI_CART_SHIFT
  - MPI_CART_SUB
  - MPI_CARTDIM_GET
  - MPI_CART_GET
  - MPI_CART_SHIFT

Problemy...?