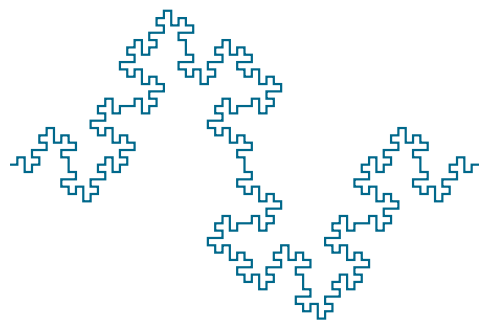
Programowanie współbieżne
Wykład 12
MPI c.d.

Rafał Skinderowicz



Komunikacja grupowa

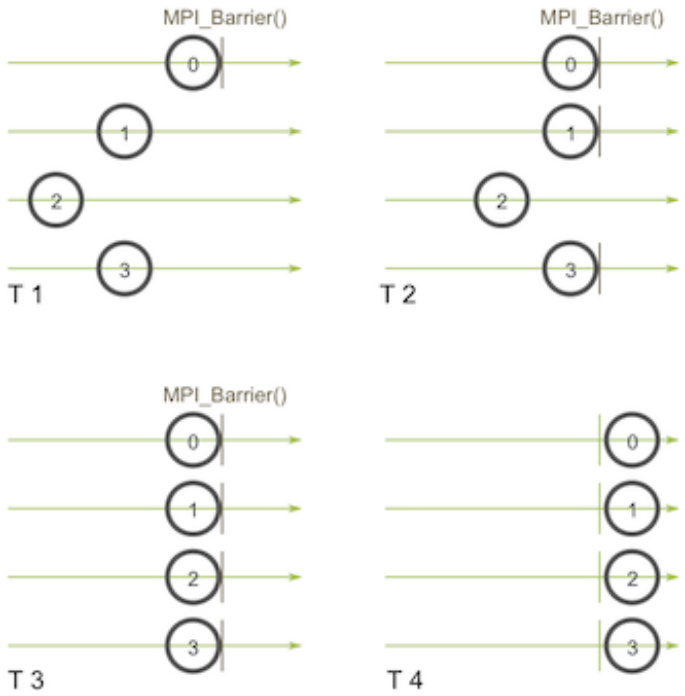
- Poprzednio rozważaliśmy komunikację między parami procesów
  - synchroniczna (blokująca) – np. `MPI_Recv`, `MPI_Send`
  - asynchroniczna (nieblokująca) – np. `MPI_Irecv`, `MPI_Isend`
- Często zachodzi konieczność wymiany komunikatów między wszystkimi procesami, na przykład:
  - proces główny wysyła do pozostałych procesów dane do obliczeń – rozgłaszanie (ang. broadcast)
  - procesy po zakończeniu obliczeń muszą wysłać częściowe wyniki do procesu głównego – „gromadzenie" (ang. gather) danych z wszystkich procesów tylko w jednym procesie
  - każdy proces potrzebuje dane z wszystkich pozostałych procesów – połączenie rozgłaszania z gromadzeniem

Komunikacja grupowa

- MPI udostępnia funkcje do wykonywania komunikacji między wszystkimi procesami wewnątrz *komunikatora*
  - wygodniejsze w użyciu, niż komunikacja między parami procesów
  - większa efektywność komunikacji – niekoniecznie programu
- Komunikacja grupowa wymaga synchronizacji wszystkich procesów
  - wszystkie procesy muszą osiągnąć ten sam punkt synchronizacji zanim będą mogły kontynuować
  - punktem synchronizacji musi być we wszystkich procesach ta sama operacja (funkcja), by obliczenia mogły być kontynuowane
  - w MPI 3.0 planowane jest dodanie funkcji nieblokujących (asynchronicznych) do komunikacji grupowej

MPI_Barrier

Do synchronizacji procesów w MPI służy funkcja:
`MPI_Barrier(MPI_Comm komunikator)`

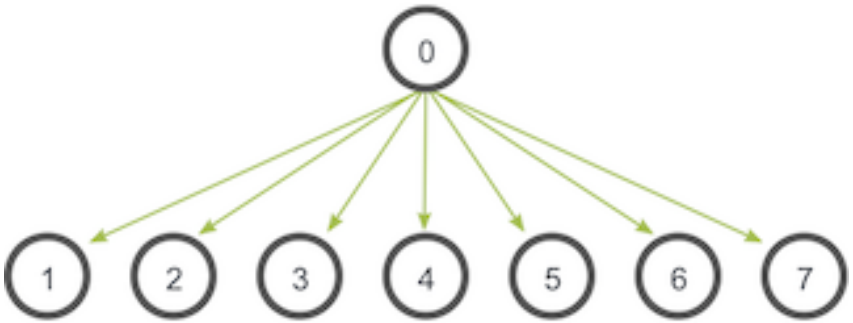


Źródło: http://www.mpitutorial.com/

Jawna synchronizacja stosowana jest zazwyczaj do podziału wykonania programu na sekcje.

Rozgłaszanie

Rozgłaszanie – proces wysyła kopie danych do pozostałych procesów w obrębie komunikatora



Źródło: http://www.mpitutorial.com/

- Przydatne przy wysyłaniu przez proces główny parametrów obliczeń lub danych wejściowych do pozostałych procesów

MPI_Bcast

W MPI rozgłaszanie realizowane jest za pomocą funkcji:
`MPI_Bcast(void *data, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

Zarówno proces nadawca, jak i procesy odbiorcy wywołują tę samą metodę, różnica jest w znaczeniu parametrów

- `data`
  - w procesie–nadawcy wsk. bufora z danymi do wysłania
  - w procesie–odbiorcy wsk. bufora na rozgłaszane dane
- `count` – rozmiar danych do wysłania
- `datatype` – typ wysyłanych danych, np. MPI_DOUBLE
- `root` – identyfikator (ranga) procesu rozgłaszającego
- `comm` – komunikator, może być `MPI_COMM_WORLD`

Wykonanie rozgłaszania powoduje niejawną synchronizację procesów

MPI_Bcast – uwagi

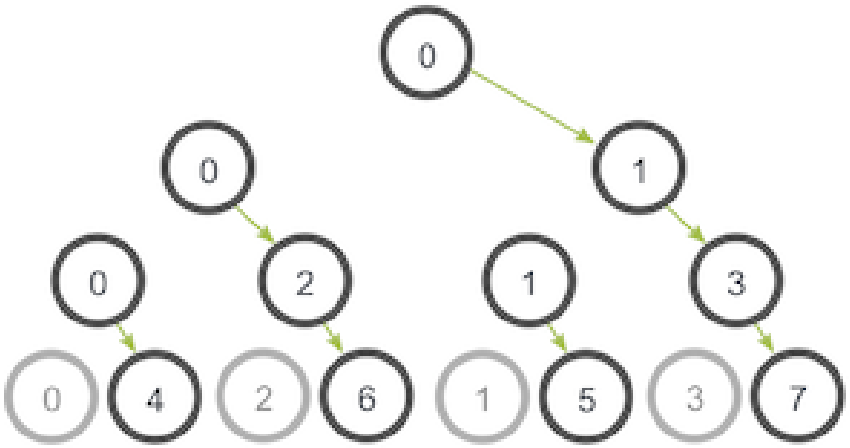
Wykonanie funkcji MPI_Bcast jest „w przybliżeniu" równoważne wykonaniu poniższej „naiwnej" implementacji:

```
void my_bcast(void* data, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm) {
 int proc_rank, proc_num;
 MPI_Comm_rank(comm, &proc_rank);
 MPI_Comm_size(comm, &proc_num);

 if (proc_rank == root) { // proces rozgłaszający
   for (int i = 0; i < proc_num; i++) {
     if (i != proc_rank) {
       MPI_Send(data, count, datatype, i, 0, communicator);
     }
   }
 } else { // proces odbiorca, odbierz dane od nadawcy (root)
   MPI_Recv(data, count, datatype, root, 0, communicator,
            MPI_STATUS_IGNORE);
 }
}
```

MPI_Bcast – uwagi

- Przedstawiona implementacja jest bardzo nieefektywna – dla n procesów proces–nadawca musi wysłać $n-1$ komunikatów
- Na szczęście można zrobić to znacznie szybciej, zgodnie z porządkiem „drzewiastym”



Źródło: http://www.mpitutorial.com/

- Teraz pojedynczy proces musi wysłać, co najwyżej $O(\log n)$ komunikatów

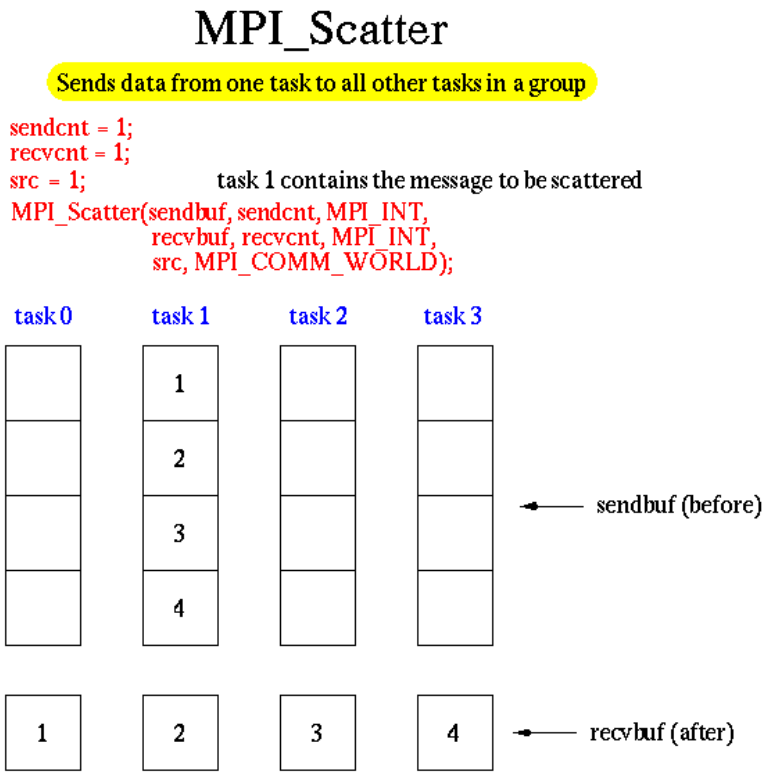
MPI_Bcast – przykład

Rozgłaszanie dla macierzy dynamicznej najłatwiej wykonać umieszczając ją w spójnym obszarze pamięci.

```
const int N = 10;
int *bufor = new int[N*N];
if (proc_id == 0) {
   int **m = new int*[N]; // inicjuj macierz
   for (int i = 0; i < N; i++) {
       m[i] = bufor + i * N;
       for (int j = 0; j < N; j++)
           m[i][j] = (i+1) * (j+1); // przykł. zawartość
   }
}
MPI_Bcast(bufor, N*N, MPI_INT, 0, MPI_COMM_WORLD);
if (proc_id != 0) {
   int **m = new int*[N];
   // poskładaj macierz
   for (int i = 0; i < N; i++)
       m[i] = bufor + i * N;
   // teraz można wykonywać operacje na elementach m[i][j]
}
```

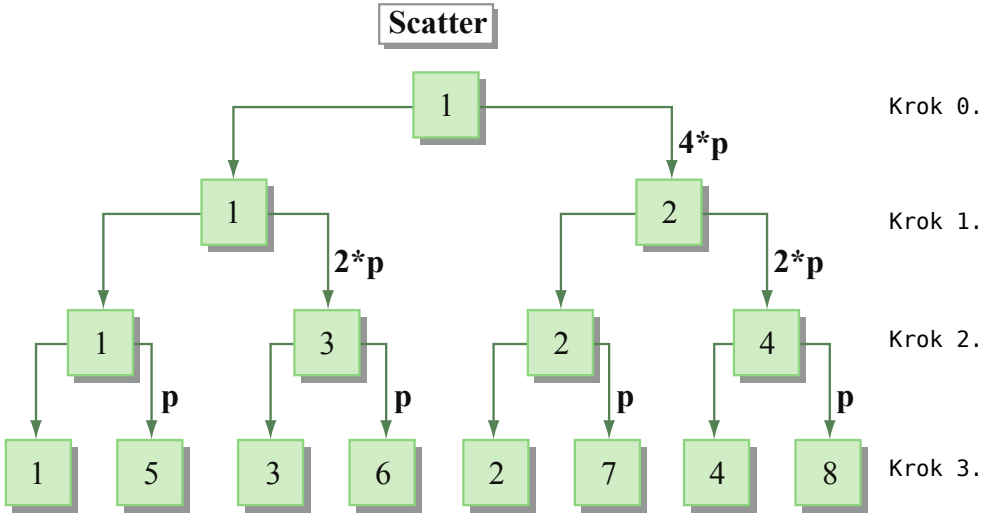
MPI_Scatter

- Czasem zachodzi potrzeba wysłania do wszystkich procesów różnych porcji danych tego samego rodzaju, np. wierszy macierzy – służy do tego funkcja `MPI_Scatter`



Źródło: https://computing.llnl.gov/

MPI_Scatter – efektywność

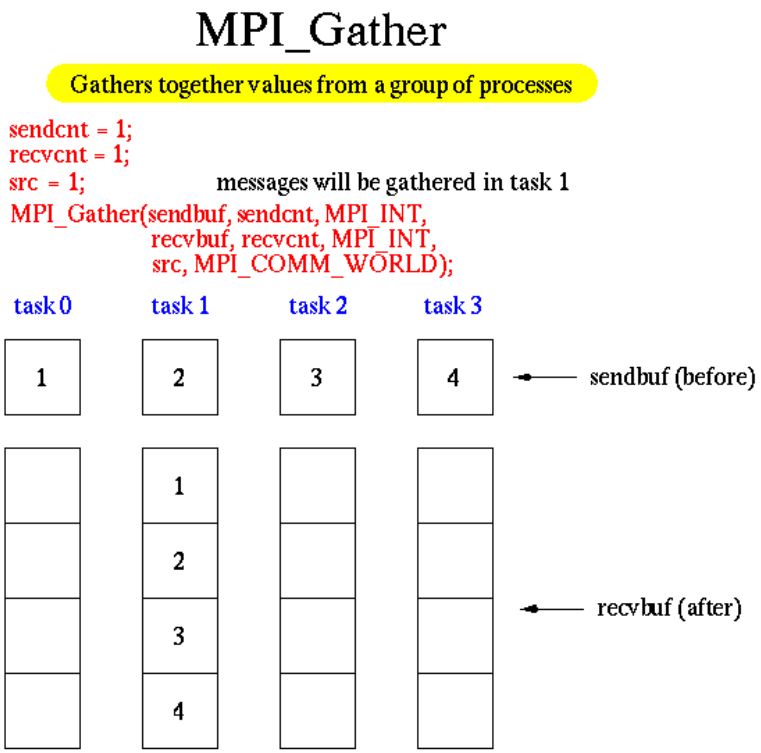


Źródło: MIT OpenCourseWare

Podobnie jak `MPI_Bcast` operacja `MPI_Scatter` wykonywana jest efektywnie, tak by zaangażować w rozsyłanie danych jak najwięcej procesów.

MPI_Gather

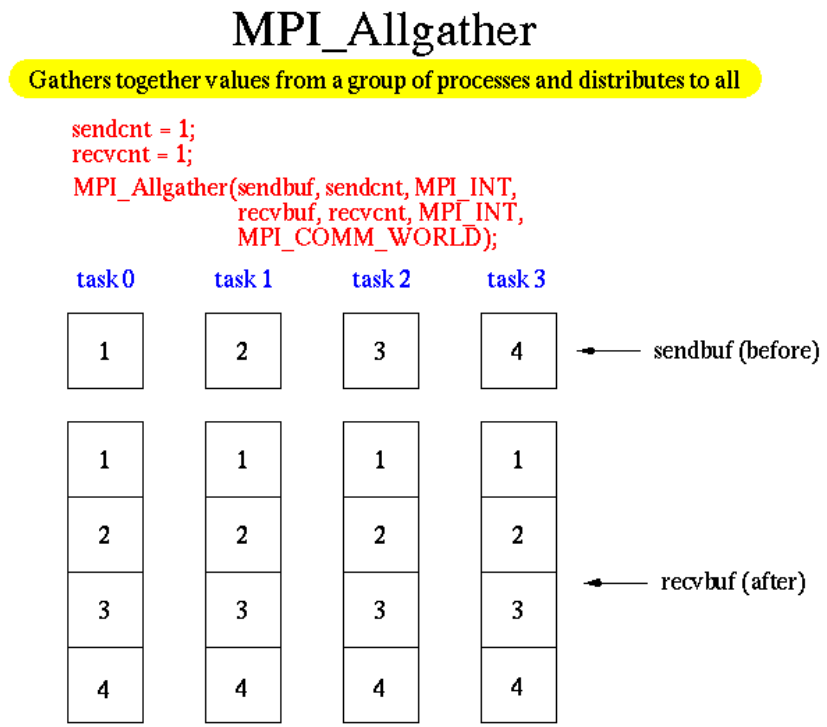
- Odwrotnością funkcji `MPI_Scatter` jest funkcja `MPI_Gather`, która polega na *zebraniu* przez jeden proces danych od pozostałych procesów



Źródło: https://computing.llnl.gov/

MPI_Allgather

- Jeżeli *każdy* proces ma uzyskać dostęp do danych wszystkich *procesów*, to odpowiednią metodą jest `MPI_Allgather`



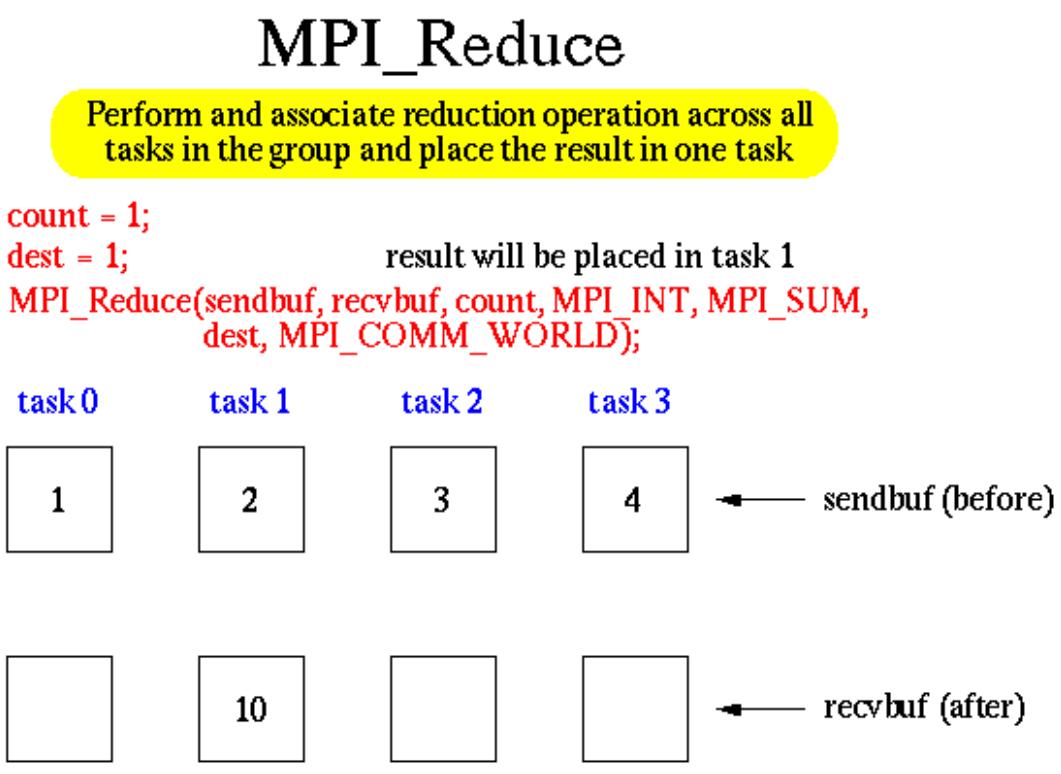
Źródło: https://computing.llnl.gov/

Operacje redukcji

- MPI udostępnia możliwość efektywnego wykonywania operacji redukcji, tj. łącznej operacji dwuargumentowej na zbiorze danych rozproszonych między procesami w komunikatorze: `MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
- Zdefiniowane są następujące rodzaje operacji (MPI_Op):
  - MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC
  - MPI_SUM, MPI_PROD
  - MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR
- MPI_MINLOC i MPI_MAXLOC zwracają odpowiednio minimaklną i maksymalną wartość odraz indeks (rangę) procesu, który je posiada

MPI_Reduce

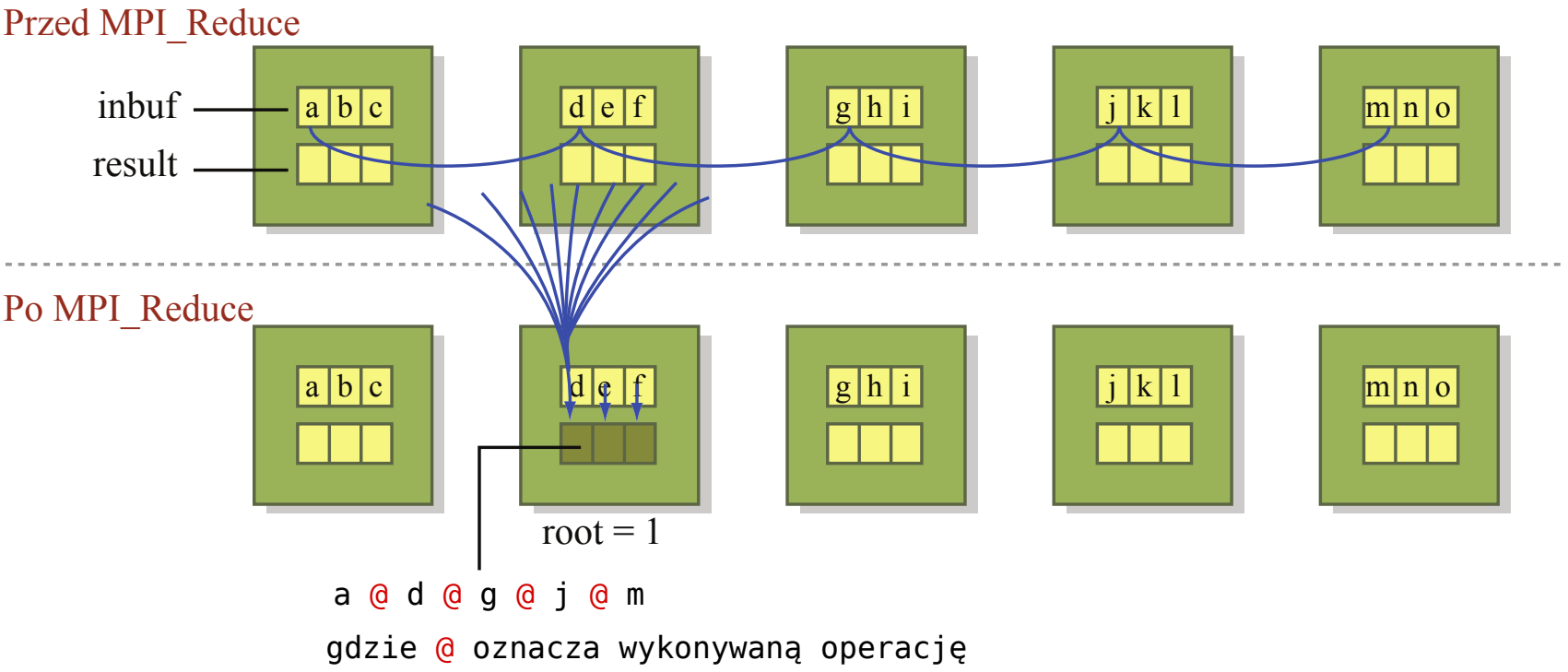
Prosty przykład sumowania liczb:



Źródło: https://computing.llnl.gov/

MPI_Reduce

W ogólnym przypadku zadana operacja wykonywana jest na wszystkich elementach podanych tablic.



Źródło: MIT OpenCourseWare

Analogicznie do operacji rozgłaszania, wykonanie redukcji wymaga $O(\log n)$ etapów komunikacji

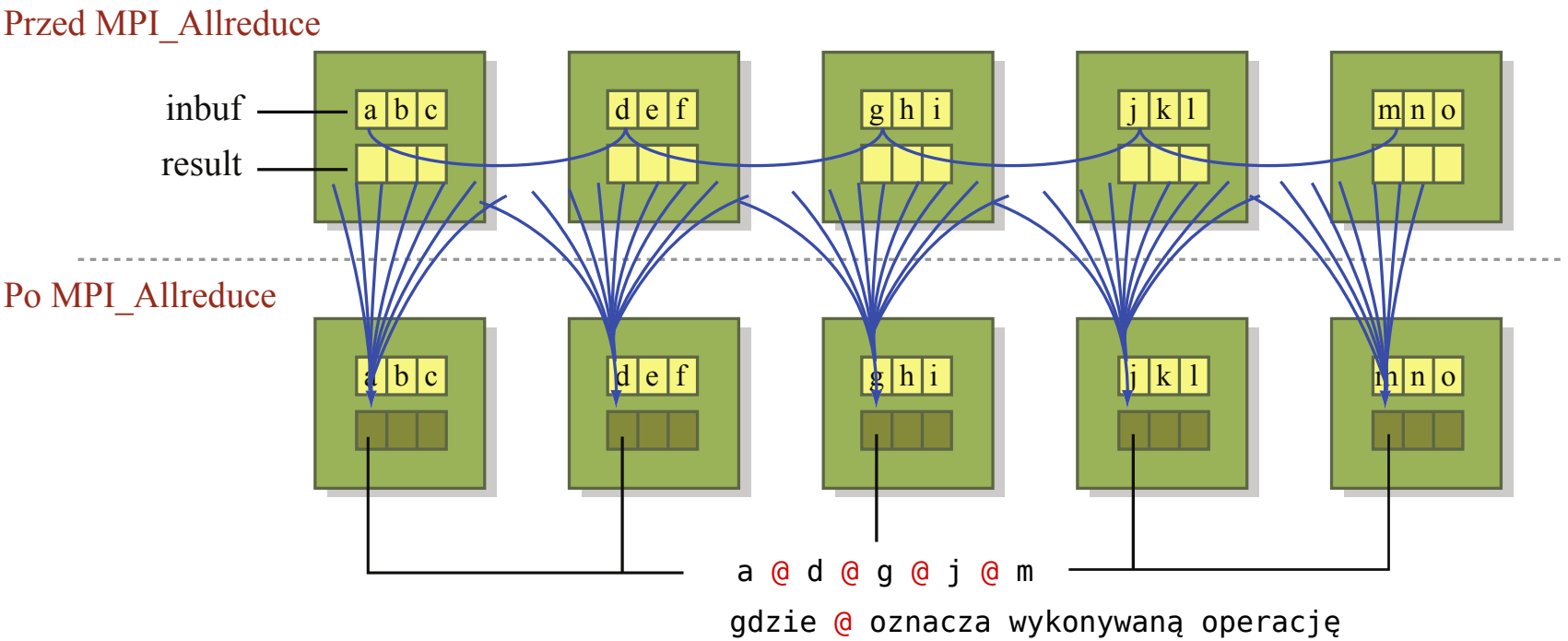
Przykład – zliczanie liczb pierwszych

```
int n, proc_id, proc_num;
MPI_Comm_size(MPI_COMM_WORLD,&proc_num);
MPI_Comm_rank(MPI_COMM_WORLD,&proc_id);
if (proc_id == 0)
   printf("Podaj n: "); scanf("%d",&n);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
double start = MPI_Wtime();
int sum = proc_id == 0 ? 1 : 0;
for (int i = 2*(proc_id + 1)+1; i <= n; i += 2*proc_num) {
   if (is_prime(i))
       sum++;
}
double koniec = MPI_Wtime();
int wynik = 0;
MPI_Reduce(&sum, &wynik, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (proc_id == 0) {
   printf("Jest %d liczb pierwszych <= %d", wynik, n);
   printf("Czas obliczen: %.2lf sek.", koniec - start);
}
```

MPI_Allreduce

Możliwe jest również wykonywanie operacji redukcji dla wszystkich procesów jednocześnie za pomocą:

```
MPI_Allreduce(void *sendbuf, void *recvbuff, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

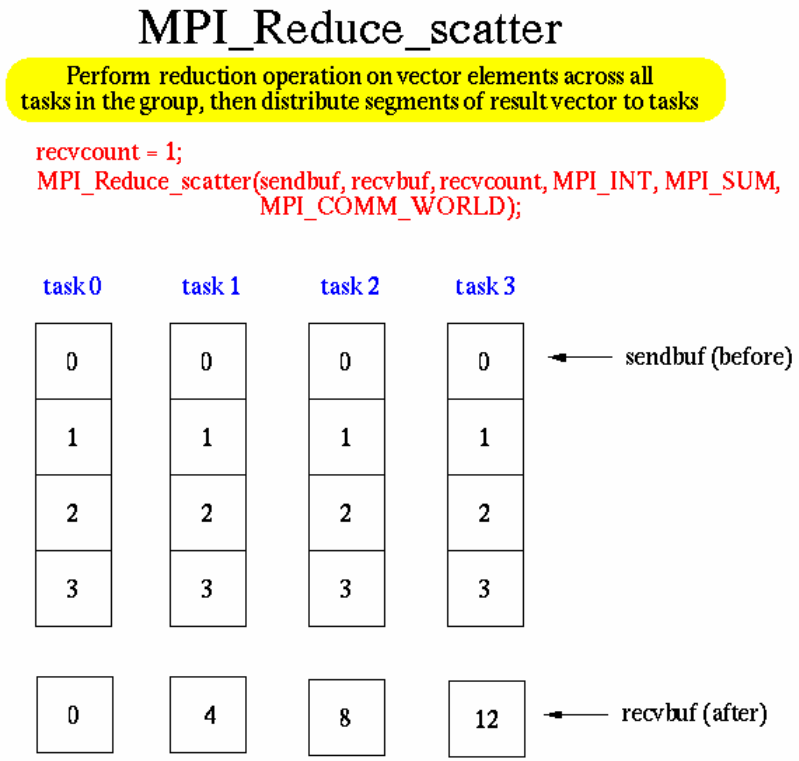


Źródło: MIT OpenCourseWare

MPI_Reduce_scatter

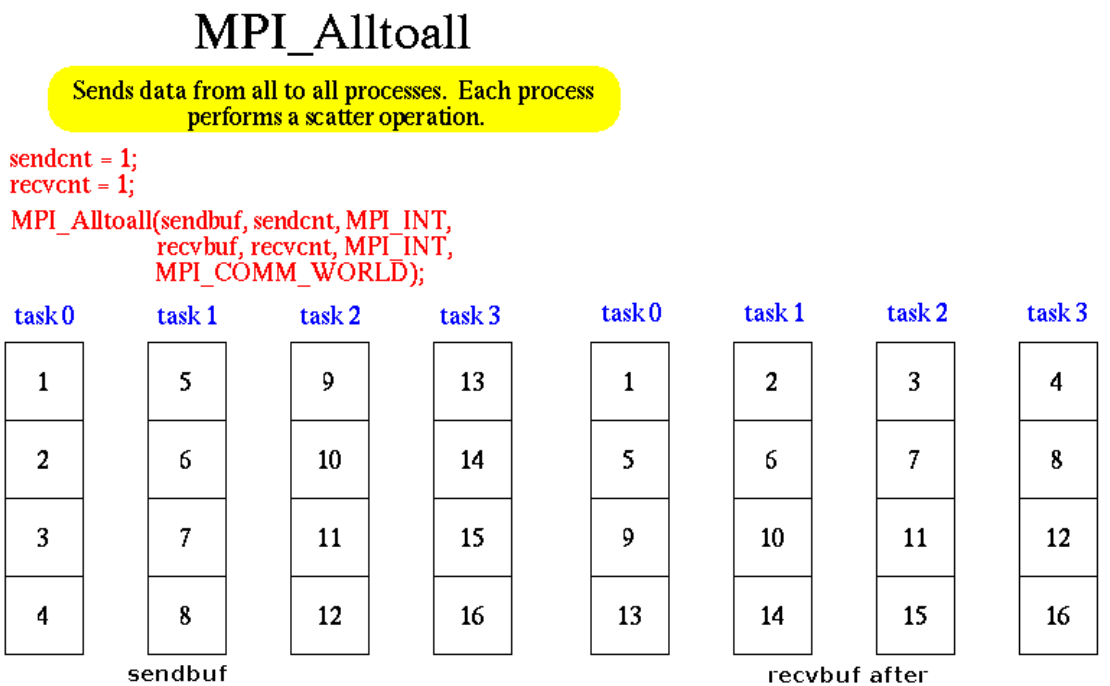
Jak nazwa wskazuje polega na wykonaniu:

- operacji redukcji na tablicach (wektorach) wszystkich procesów w komunikatorze (reduce)
- rezultat dzielony jest na rozłączne fragmenty rozsyłane do poszczególnych procesów (scatter)



MPI_Alltoall

- Ciekawą operacją jest `MPI_Alltoall`, która polega na wysłaniu przez każdy proces unikalnej porcji danych do wszystkich pozostałych procesów
- $n(n-1)$ operacji wysyłania komunikatów



Źródło: https://computing.llnl.gov/

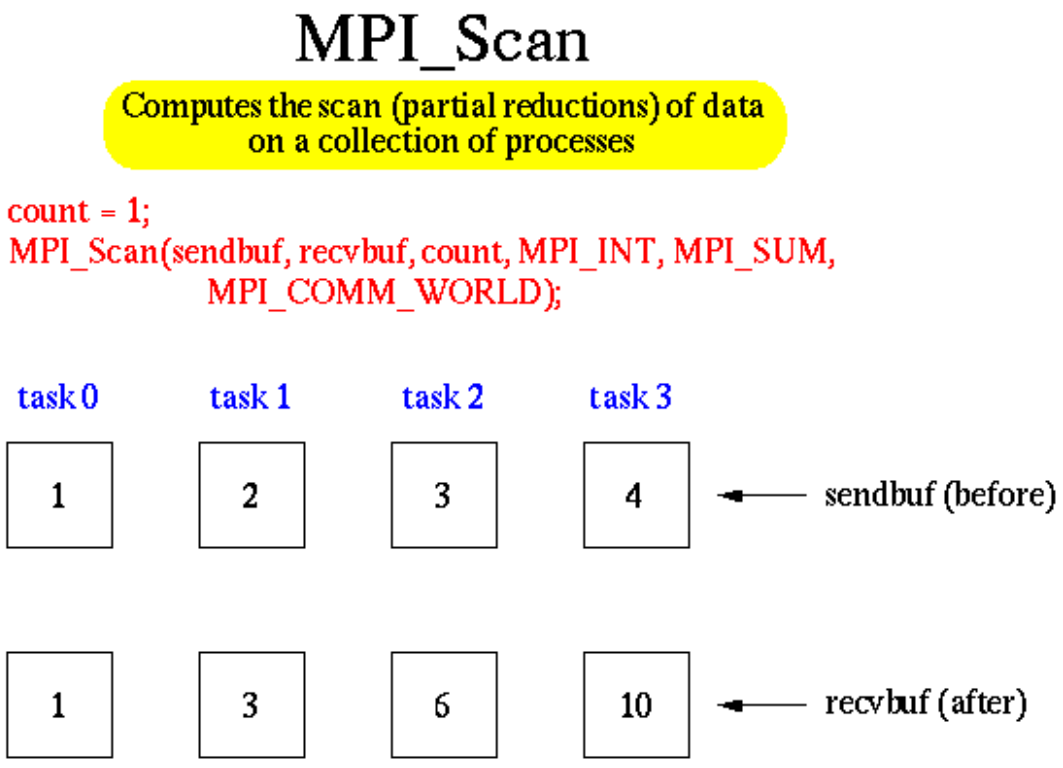
Operacje prefiksowe

- Operacje prefiksowe realizowane są przez funkcję

```
MPI_Scan(void *sendbuf, void *recvbuf, int count,
MPI_Datatype type, MPI_Op op, MPI_Comm comm)
```

- Polegają na wykonaniu częściowej redukcji
- Proces *i* otrzymuje wynik redukcji dla danych procesów o indeksach (rangach) od 0 do *i*
- Proces o najwyższej randze ma wynik dla wszystkich danych
- Funkcja obsługuje takie operatory, jak `MPI_Reduce`

MPI_Scan – przykład



Źródło: https://computing.llnl.gov/

Trwała komunikacja

- W przypadku częstej komunikacji, np. w pętli, można uniknąć narzutu czasowego związanego z inicjowaniem operacji komunikacji za pomocą utworzenia trwałego żądania komunikacyjnego (ang. persistent communication request) – „półkanał" komunikacyjny
- Oszczędność wynika z przypisania parametrów operacji na stałe do żądania
- `MPI_Send_init(buf, count, datatype, dest, tag, comm, request)` tworzy trwałe żądanie wysłania
- `MPI_Recv_init(buf, count, datatype, source, tag, comm, request)` tworzy trwałe żądanie odbierania
- Parameter `request` pełni podobną rolę jak w operacjach `MPI_Irecv` oraz `MPI_Isend`

Trwała komunikacja – scenariusz zastosowania

1. Utworzenie trwałego żądania za pomocą `MPI_Send_init` lub `MPI_Recv_init`
2. Zainicjowanie operacji wysłania / odbioru za pomocą `MPI_Start`
3. Czekanie na zakończenie zainicjowanej operacji za pomocą `MPI_Wait` lub `MPI_Waitall`
4. Usunięcie żądania za pomocą `MPI_Request_free`

Kroki 2. i 3. powtarzane są tyle razy, ile komunikatów ma zostać wysłanych / odebranych

Trwała komunikacja – przykład

```
MPI_Request req[2];
MPI_Recv_init(buf1, count,type,src,tag,comm,&req[0]);
MPI_Send_init(buf2, count,type,src,tag,comm,&req[1]);
const int N = 100000;
for (i=1; i < N; i++)
{
   MPI_Start(&req[0]);
   MPI_Start(&req[1]);
   MPI_Waitall(2,req,status);
   obliczenia(buf1, buf2);
}
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```

Definiowanie nowych typów danych

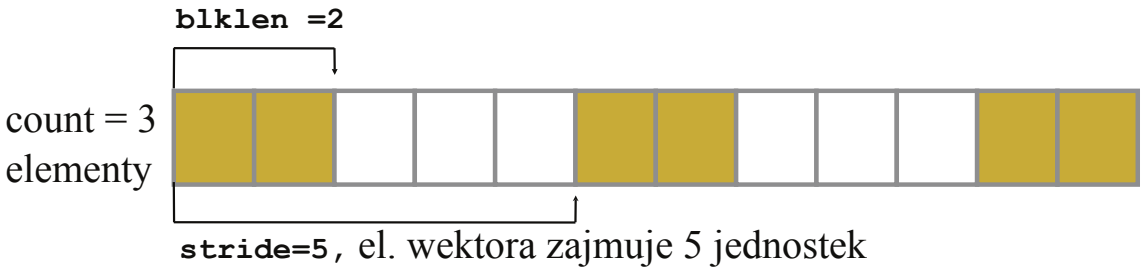
- Typy danych zdefiniowane w MPI pozwalają na łatwą wymianę tablic złożonych z elementów jednego z podstawowych typów, np. MPI_INT
- W programach często zachodzi konieczność wymiany złożonych danych różnego typu
  1. Można wysyłać kolejno składowe w osobnych operacjach – niewygodne i wolne
  2. Można zdefiniować własny typ dla przesyłanych danych – znacznie wygodniejsze
  3. Można stosować „pakowanie" i „rozpakowywanie" do i z bufora

Definiowanie nowych typów danych c.d.

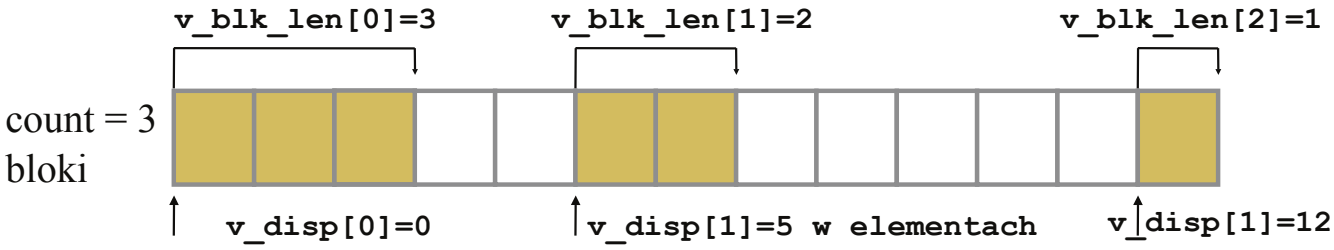
MPI udostępnia funkcje umożliwiające zdefiniowanie nowych typów danych należących do jednej z kategorii:

- Elementarne – typy danych zdefiniowane w języku programowania
- Ciągłe – tablice
- Wektory – tablice, w których kolejne elementy oddzielone są separatorem o określonym rozmiarze
- Indeksowane – podobnie jak wektor, ale adresy kolejnych elementów podawane są jawnie
- Strukturalne – najbardziej ogólne, umożliwia przesyłanie struktur (`struct`) z języka C

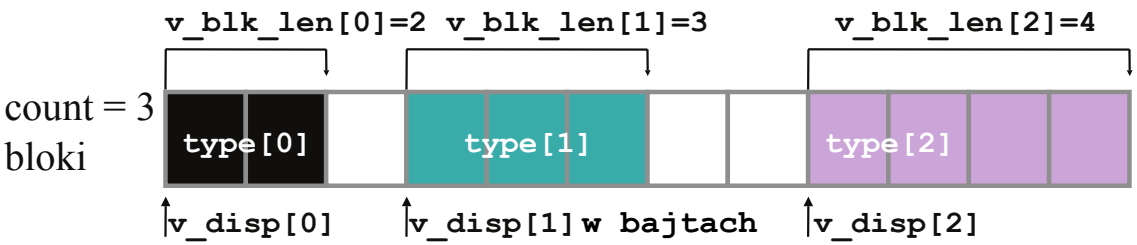
Definiowanie nowych typów danych c.d.



Wektor
(z separatorami)



T. indeksowany



Strukturalny

Definiowanie nowych typów – przykład

```
typedef struct {
   float wartosc;
   char id;
} Pomiar;

int v_blk_len[2] = { 1, 1 }; // dł. bloków
MPI_Datatype v_types[2] = { MPI_FLOAT, MPI_CHAR };
Pomiar pom;
// ustal adresy kolejnych pól struktury
MPI_Aint v_addr[3];
MPI_Address(&pom, &v_addr[0]);
MPI_Address(&(pom.wartosc), &v_addr[1]);
MPI_Address(&(pom.id), &v_addr[2]);
// na ich podstawie wyznacz przesunięcia pól
MPI_Aint v_disp[2] = { v_addr[1] - v_addr[0],
                       v_addr[2] - v_addr[1] };
// zdefiniuj typ
MPI_Datatype pomiar_type;
MPI_Type_struct(2, v_blk_len, v_disp, v_types, &pomiar_type);
MPI_Type_commit(&pomiar_type);
```

Definiowanie nowych typów – przykład, c.d.

```
const int N = 10;
Pomiar pomiary[N];
if (proc_id == 0) {
   for (int i = 0; i < N; ++i) {
       pomiary[i].wartosc = 1 + i;
       pomiary[i].id = 'A' + i;
   }
}
MPI_Bcast(pomiary, N, pomiar_type, 0, MPI_COMM_WORLD);
if (proc_id != 0) {
   for (int i = 0; i < N; i++) {
       cout << pomiary[i].wartosc
            << ", " << pomiary[i].id << endl;
   }
}
```

Grupowanie danych

- Definiowanie własnych typów danych wymaga dodatkowego nakładu pracy, który warto ponieść w przypadku częstej wymiany komunikatów o tej samej strukturze
- W przypadku, gdy wymiana danych złożonych pojawia się w jednym miejscu programu można skorzystać z możliwości przesyłania danych zgrupowanych (spakowanych)
- Wszystkie dane muszą zostać zapisane w ciągłym obszarze pamięci

Grupowanie danych – funkcje

- `int MPI_Pack(void *pack_data, int in_count, MPI_Datatype datatype, void *buffer, int size, int *position_ptr, MPI_Comm comm)`
  - Bufor pack_data powinien zawierać in_count elementów typu datatype.
  - Parametr position_ptr zawiera adres początkowy w buforze wyjściowym buffer i po wykonaniu funkcji jego wartość jest uaktualniana (dodawana jest długość zapisanych danych)
  - Parametr size określa rozmiar obszaru pamięci wskazywanego przez buffer.
- `int MPI_Unpack(void buffer, int size, int position_ptr, void *unpack_data, int count, MPI_Datatype datatype, MPI_Comm comm)`
  - Funkcja rozpakowuje count elementów typu datatype ze zmiennej buffer począwszy od pozycji position do obszaru pamięci wskazywanego przez unpack_data i odpowiednio uaktualnia zmienną position.

Grupowanie danych – przykład

```
float a, b;
int l;
char buffer[100];
int pos;
// Proces wysyłający dane
pos=0;
MPI_Pack(&a, 1, MPI_FLOAT, buffer, 100, &pos, MPI_COMM_WORLD);
MPI_Pack(&b, 1, MPI_FLOAT, buffer, 100, &pos, MPI_COMM_WORLD);
MPI_Pack(&l, 1, MPI_INT, buffer, 100, &pos, MPI_COMM_WORLD);
MPI_Bcast(buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);
...
// Proces odbierający dane

MPI_Bcast(buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);
pos=0;
MPI_Unpack(buffer, 100, &pos, &a, 1, MPI_FLOAT, MPI_COMM_WORLD);
MPI_Unpack(buffer, 100, &pos, &b, 1, MPI_FLOAT, MPI_COMM_WORLD);
MPI_Unpack(buffer, 100, &pos, &l, 1, MPI_INT, MPI_COMM_WORLD);
```

Równoległe I/O

- MPI udostępnia również szereg funkcji do wykonywania operacji wejścia/wyjścia w sposób równoległy (i rozproszony)
  - Program może jednocześnie zapisywać i odczytywać dany plik za pośrednictwem funkcji MPI-IO
  - Wymaga równoległego systemu plików (ang. parallel file system) – modułowy, rozproszony; Przykłady: Lustre, IBM GPFS, PanFS
- Alternatywy dla MPI-IO:
  - Wyznaczony proces zajmuje się odczytem / zapisem do pliku i rozsyłaniem oraz gromadzeniem danych
  - Każdy proces korzysta z odrębnego pliku

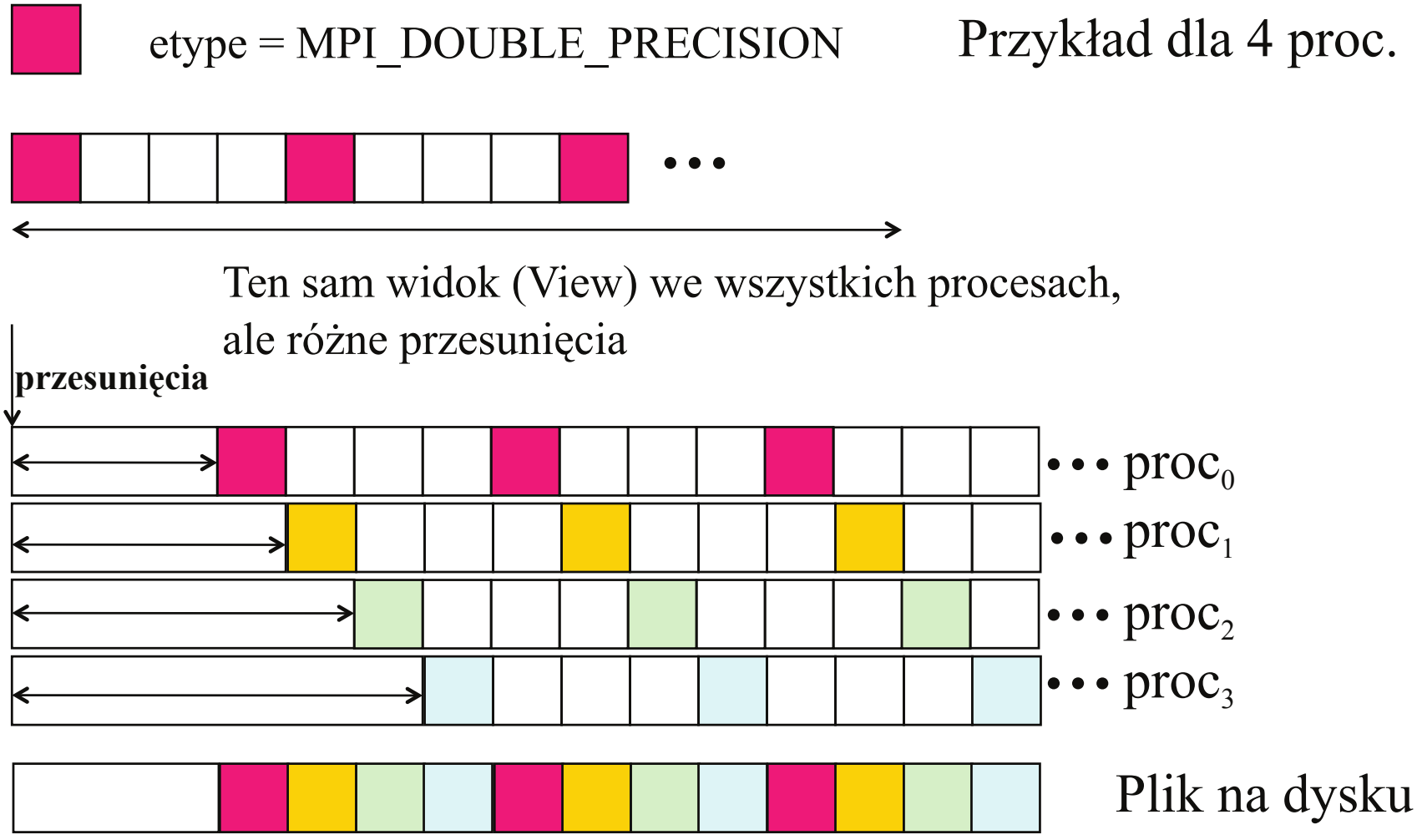
Równoległe I/O c.d.

- Funkcje do obsługi I/O zostały zaimplementowane dopiero w MPI 2.0
- Zalety równoległego I/O, to:
  - zwiększona wydajność
  - zunifikowany dostęp do plików, podobnie jak w przypadku programu sekwencyjnego – ułatwia przetwarzanie plików z wynikami obliczeń
- Stosowanie sekwencyjnego I/O wymaga dodatkowo kopiowania plików między węzłami obliczeniowymi – każdy proces zapisuje do lokalnego systemu plików
- MPI-IO udostępnia mechanizmy do:
  - synchronizacji plików między serwerami oraz ich kopiowania
  - „nieciągłego" sposobu wykonywania operacji na pliku za pomocą widoków (ang. file views)

Równoległe I/O – przykład

```
MPI_File fh;
MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);
MPI_File_open(MPI_COMM_WORLD, "/home/datafile",
                MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek( fh, rank*bufsize, MPI_SEEK_SET);
MPI_File_read( fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

Równoległe I/O – File View



Równoległe I/O – File View

```
#define N 100

// nowy typ danych - tablicowy
MPI_Datatype arraytype;
MPI_Offset disp;
MPI_Type_contiguous(N, MPI_INT, &arraytype);
MPI_Type_commit(&arraytype);

disp = rank * sizeof(int) * N; // przesunięcie
etype = MPI_INT;
MPI_File_open(MPI_COMM_WORLD, "/home/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, arraytype, "native",
                  MPI_INFO_NULL);
MPI_File_write( fh, buf, N, MPI_INT, MPI_STATUS_IGNORE);
```