Operacje kolektywne MPI

Operacje kolektywne

- Do tej pory w operacje przesyłania komunikatu miały charakter punkt-punkt (najczęściej pomiędzy nadawcą i odbiorcą).
- ***W operacjach grupowych udział biorą wszystkie procesy w komunikatorze; wszystkie muszą wywołać daną funkcję.***
  - Jeżeli używamy `MPI_COMM_WORLD` oznacza to wszystkie procesy aplikacji.
- Wszystkie operacje grupowe są blokujące. Niektóre są synchronizujące.
- Komunikaty nie mają etykiet.
- Procesy odbierające i nadające używają takiej samej długości komunikatu.

Operacja barriery

```
int MPI_Barrier(MPI_Comm com);
```

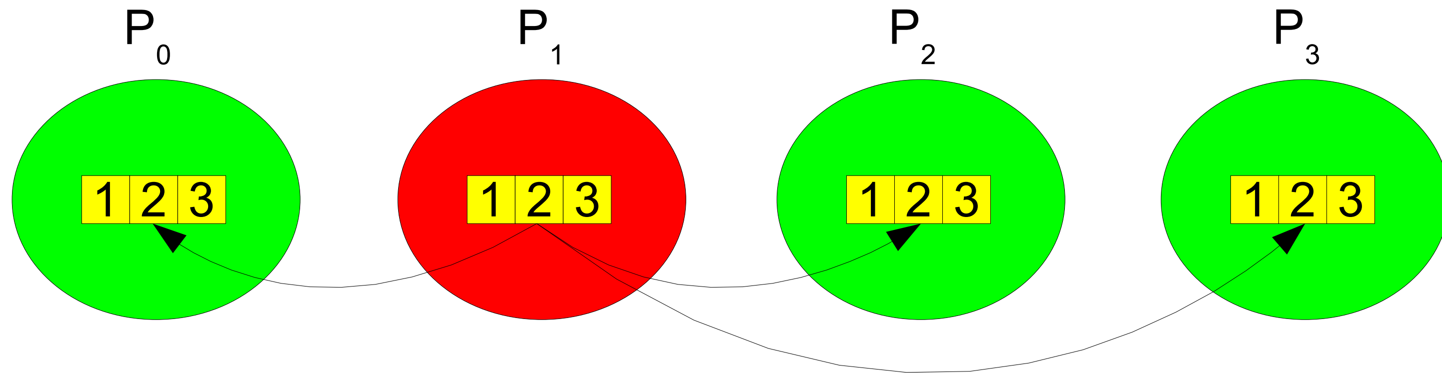
- Jest to operacja tylko synchronizująca. Nie przesyła i nie przetwarza żadnych danych. Przypominam, że muszą wywołać wszystkie procesy w komunikatorze.
- Definicja jest następująca

„Proces może opuścić operację MPI_Barrier dopiero gdy wszystkie inne procesy w komunikatorze wywołają tę operację".

- Była wykorzystana w programie Mandelbrot przy pomiarze czasu.
  - Proces 0 mierzył czas swojej funkcji Master (`MPI_Wtime`)
  - Ale co się wydarzy gdy proces 0 wystartuje na maszynie jako pierwszy, a pozostałe znacznie później -> do czasu obliczeń zostanie dodany czas oczekiwania na pozostałe procesy (przy pierwszej operacji Send albo Receive).
  - Wyjście: wszystkie procesy aplikacji po zainicjowaniu MPI, przed pomiarem czasu wykonują kod:

```
MPI_Barrier(MPI_COMM_WORLD);
```

Operacja rozgłaszania (ang. broadcast).



- Jeden proces (w przykładzie o numerze 1, zwany root proces) przesyła komunikat, który otrzymują pozostałe procesy.

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

- Przypominam, że funkcje muszą wywołać wszystkie procesy w komunikatorze !!!
- Parametr `root`, to numer procesu który jest nadawcą, pozostałe są odbiorcami.
  - Wszystkie procesy muszą dostarczyć ten sam numer root.
- Wszystkie procesy muszą dostarczyć takie same wartości parametrów `count` oraz `datatype`.

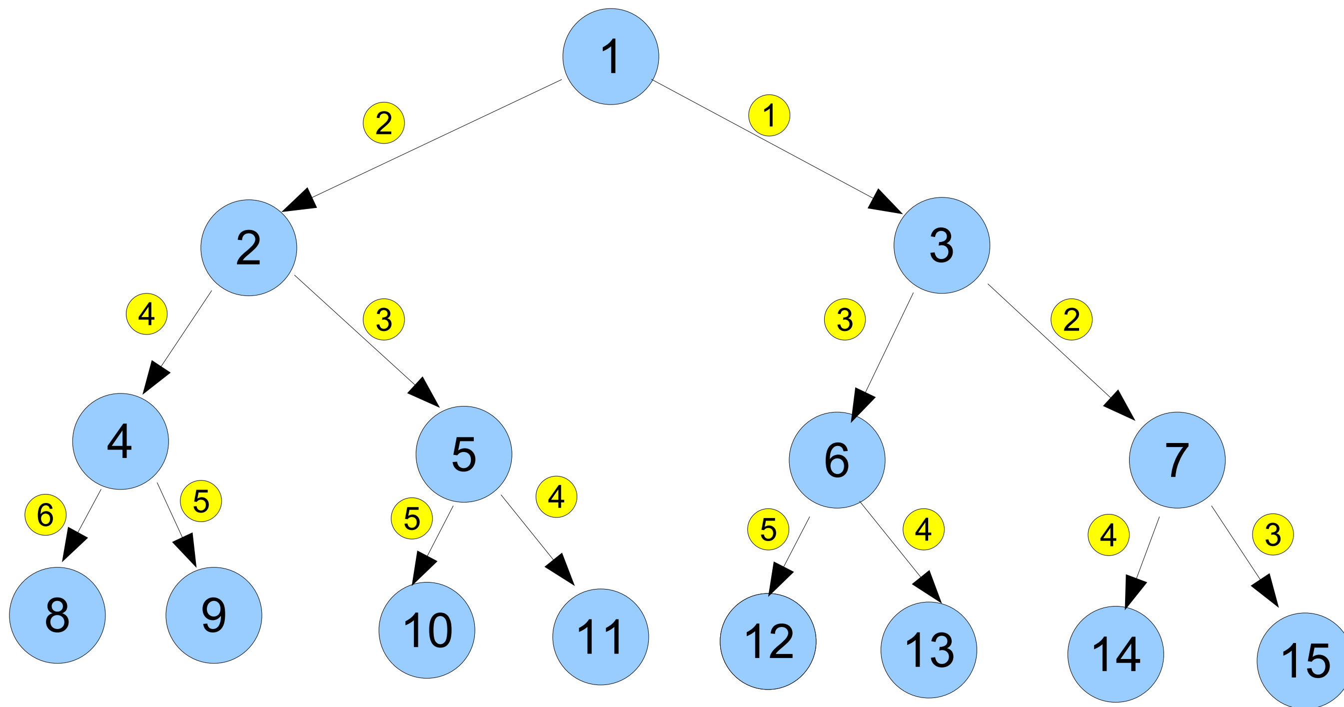
Broadcast – złożoność obliczeniowa

- Najprostsza implementacja operacji broadcast dla p procesów wysyła p komunikatów ze źródła. Złożoność jest O(p). Pseudokod:

```
if (myrank == root) {
   for (int i=0;i<N;i++)
      if (i != myrank)
         MPI_Send(buf,count,datatype,i,...,comm);
} else
   MPI_Recv(buf, count,datatype, root,...,comm,status);
```

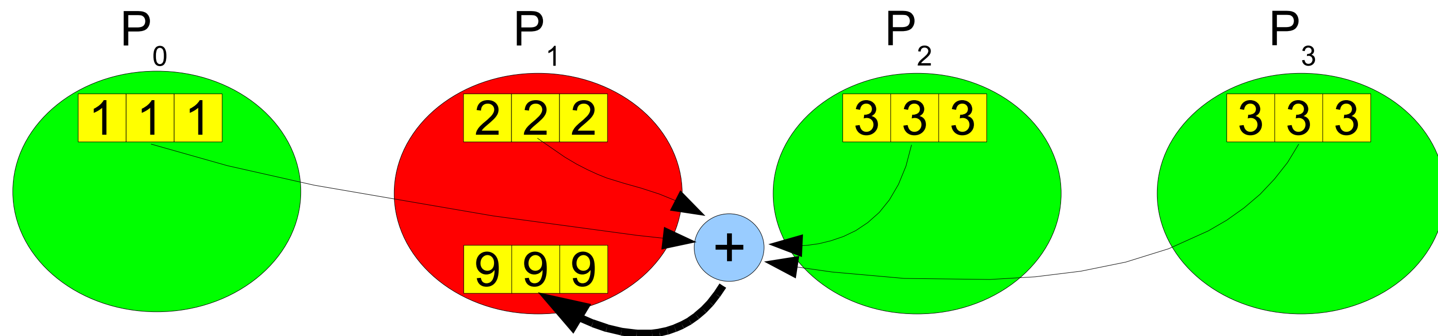
- Istnieją lepsze algorytmy, oparte na różnych strukturach drzewiastych.
  - Realna złożoność jest O(log p).

Broadcast po drzewie binarnym - demo



- Istnieją jeszcze lepsze algorytmy (np. teraz w krokach >2 proces 1 jest bezczynny)
  - Drzewo rozpinające hipersześcian (4 kroki).
- Dla wysokiej wydajności sieć połączeń musi wspierać równoległe przesyłanie komunikatów

Operacja redukcji (ang. reduce).



- Wszystkie procesy wykonują przemienną operację na danych (np. Sumowanie) a wynik jest przesyłany do jednego procesu (root)

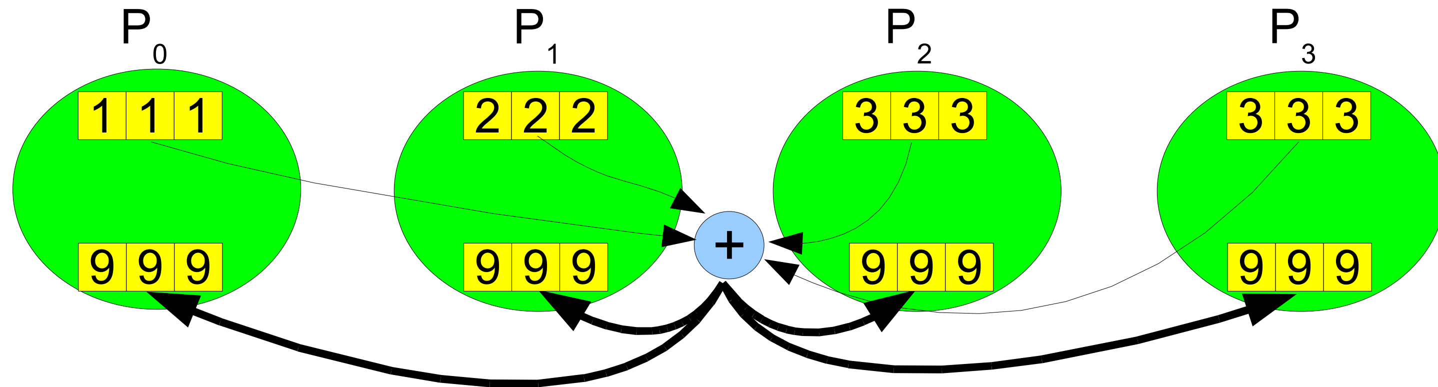
```
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
comm )
```

- Proces `root`, również przesyła dane do redukcji (jest nadawcą i odbiorcą), pozostałe są nadawcami.
- Operacja redukcji jest odwrotnością operacji rozgłaszania i może być wykonana w czasie O(log p) przy pomocy wydajnych algorytmów drzewiastych.

Operacje redukcji

Operacja	Znaczenie	Typy danych
`MPI_MAX`	Maksimum	C całkowite i zmiennoprzec.
`MPI_MIN`	Minimum	C całkowite i zmiennoprzec.
`MPI_SUM`	Suma	C całkowite i zmiennoprzec.
`MPI_PROD`	Iloczyn	C całkowite i zmiennoprzec.
`MPI_LAND`	Logiczne AND	C całkowite
`MPI_BAND`	Bitowe AND	C całkowite i byte
`MPI_LOR`	Logiczne OR	C całkowite
`MPI_BOR`	Bitowe OR	C całkowite i byte
`MPI_LXOR`	Logical XOR	C całkowite
`MPI_BXOR`	Bitowe XOR	C całkowite i byte
`MPI_MAXLOC`	Lokacja maksimum	Pary danych
`MPI_MINLOC`	Lokacja minimum	Pary danych

Operacja allreduce.

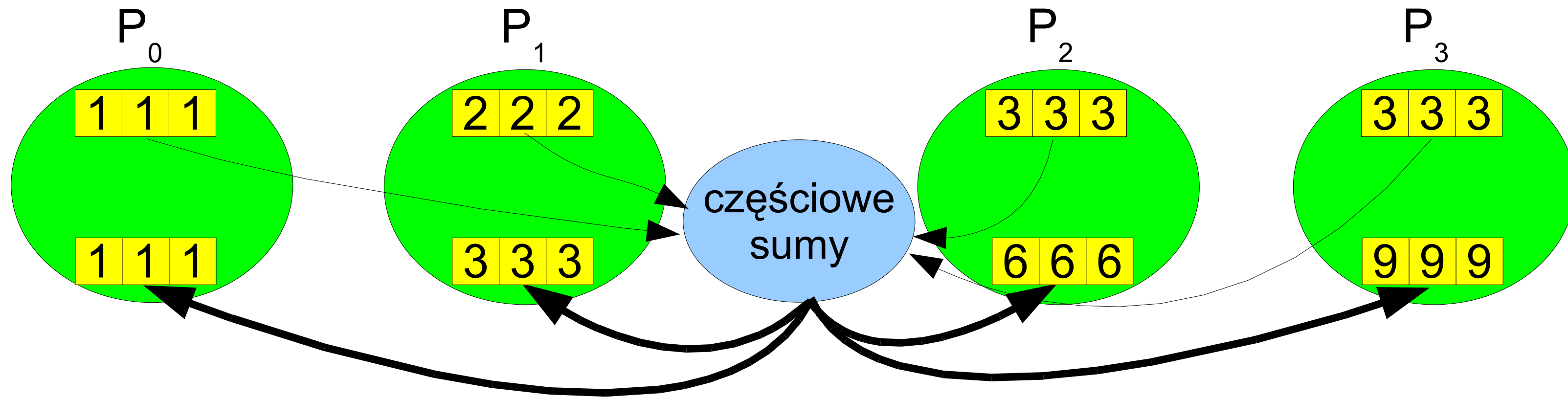


- Operacja redukcji, której wynik jest wysyłany wszystkim procesom, a nie tylko jednemu.

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

- Brak argumentu `root`, ponieważ wynik otrzymują wszystkie procesy.
- Operacja allreduce koncepcyjnie jest sekwencją operacji redukcji i operacji rozgłaszania wyniku i może być zaimplementowana w czasie O(log p) przy pomocy wydajnych algorytmów drzewiastych.

Operacja scan.



- Jest to częściowa operacja redukcji, w której proces o numerze *i* otrzymuje w swoim buforze odbiorczym `recvbuf` redukcję danych wysłanych przez procesy 0,1,...,*i*.

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

- Brak argumentu `root`, ponieważ wynik otrzymują wszystkie procesy.
- Operacja allreduce koncepcyjnie jest sekwencją operacji redukcji i operacji rozgłaszania wyniku i może być zaimplementowana w czasie O(log p) przy pomocy wydajnych algorytmów drzewiastych.

Definiowanie własnych operacji

- Własną operację redukcji możemy zdefiniować przy pomocy funkcji:

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

Parametr `commute` powinien być równy 1 jeżeli operacja jest przemienna. Pod adresem `op` zostanie zapisany identyfikator operacji do posłużenia się jako argument operacji redukcji.

- Parametr `function` jest adresem funkcji o prototypie:

```
void MPI_User_function( void * a, void * b, int * len, MPI_Datatype
*type );
```

która przeprowadza operację `b[i]=a[i] op b[i]`, dla `i=0..len-1`.

- `a` i `b` są wektorami wartości typu `type` o długości `len`.
- Stworzoną operację, gdy nie jest już potrzebna należy zwolnić przy pomocy funkcji

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

Przykład: całkowanie metodą Monte-Carlo

- Całka jest przybliżana poprzez losowanie wielokrotne losowanie wartości x_r z przedziału $[x_1, x_2]$ i sumowanie $f(x_r)$:

$$\int_{x_1}^{x_2} f(x)\,dx = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} f(x_r)(x_2 - x_1)$$

- Zrównoleglenie jest bardzo proste (problem „żenująco równoległy”): Niech każdy proces przeprowadza losowanie i oblicza własną sumę wartości funkcji.
- Następnie proces 0 zsumuje (***operacja redukcji***) sumy cząstkowe i obliczy wartość całki mnożąc sumę przez $(x_2 - x_1)$ i dzieląc przez N.
- Aby zademonstrować operację rozgłaszania poprosimy użytkownika o podanie liczby prób.
- Uwaga: należy zadbać aby ***generator liczb losowych w każdym procesie był zainicjalizowany innym ziarnem***. W przeciwnym wypadku każdy proces będzie losowałe te same liczby i wyniki nie będą statystycznie poprawne.
- Uwaga: ***istnieją o wiele wydajniejsze algorytmy całkowania !***

Obliczanie sum cząstkowych

```
int rank,size,N; // N to całkowita liczba prób
// Całka tej funkcji na przedziale [0,1] to pi
double f(double a) {
    return (4.0 / (1.0 + a*a));
}

double PartialSum() {
   int Trials=N/size,i;
   double sum=0.0;
// Korekta, aby łączna liczba prób była równa N
   if (rank<N%size) Trials++;

   for(i=0;i<Trials;i++) {
      double x=drand48();
      sum+=f(x);
   }
   return sum;
}
```

Www

- Www
- Www
-

Funkcja main (1)

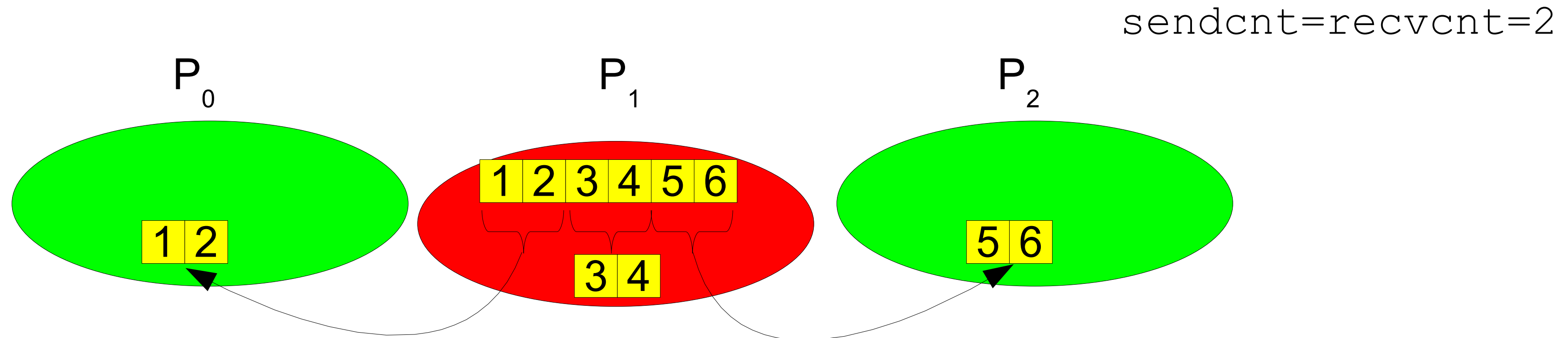
```
int main(int argc,char *argv[])
{
   double t1, t2;
   double globalsum,sum;
   const double PI25DT = 3.141592653589793238462643;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   // Inicjalizacja przy pomocy numeru procesu, każdy proces ma inne
   // ziarno generatora liczb pseudolosowych
   srand48((rank+1)*111);
   // Czekamy aż wszystkie wystartują
   MPI_Barrier(MPI_COMM_WORLD);
   if (rank == 0) {
      // Proces 0 pyta o liczbę prób
      printf("Podaj liczbę prób:");
      fflush(stdout);
      scanf("%d",&N);
      // mierzy czas
      t1=MPI_Wtime();
   }
   // Proces 0 rozgłasza liczbę prób pozostałym procesom.
   MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Funkcja main (2)

```
   // Wszystkie procesy obliczają sumę częściową
   sum=PartialSum();
   // sumy częściowe wszystkich procesów są sumowane a wynik
   // przesyłany do procesu 0 do zmiennej global sum.
   MPI_Reduce(&sum,&globalsum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
   // Proces 0 drukuje wyniki
   if (rank == 0) {
      printf("Pi jest w przybliżeniu %.16f, Błąd %.16f\n",
                  globalsum/N, fabs(globalsum/N - PI25DT));
      t2=MPI_Wtime();
      printf("Czas obliczeń = %f\n", t2-t1);
   }
   MPI_Finalize();
   return 0;
}
```

- Wszystkie procesy (w tym proces o numerze 0) przeprowadzają obliczenia
- Proces 0 jest wyróżniony poprzez
  - Wczytywanie danych (liczba prób – trzeba ją przesłać innym)
  - Drukowanie wyników (trzeba zsumować sumy cząstkowe)
  - Pomiar czasu

Operacja scatter

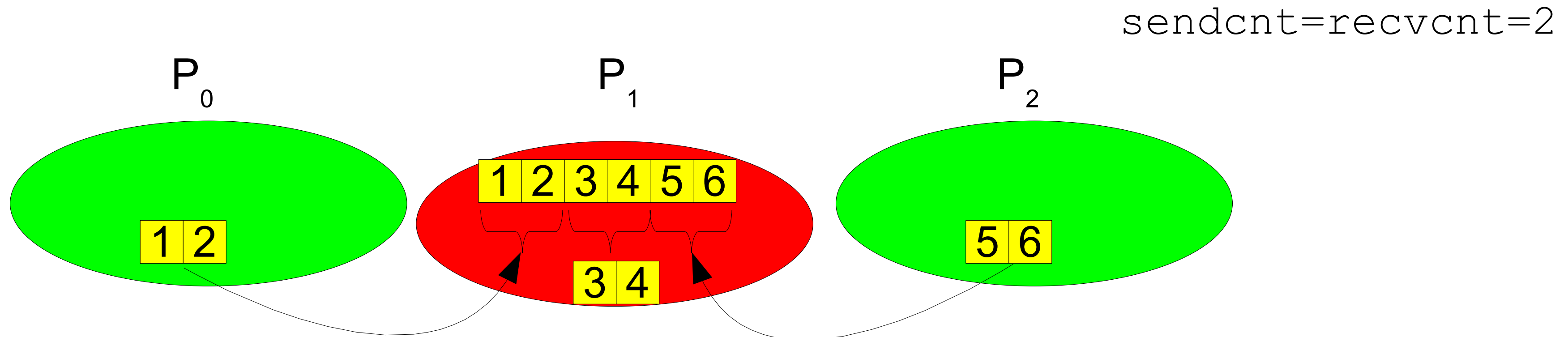


- Jest wykonywana przy pomocy funkcji

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

- Proces źródłowy `root` wysyła innym procesom (i sobie) części wektora o adresie `sendbuf` i o długości `sendcnt` każda.
- Każdy proces otrzymuje tyle samo (`recvcnt`) elementów do bufora odbiorczego pod adresem `recvbuf`. Wszystkie procesy muszą dostarczyć identyczną wartość `recvcnt`.
- Proces o numerze i odbiera elementy począwszy od numeru i*`sendcnt`.

Operacja gather

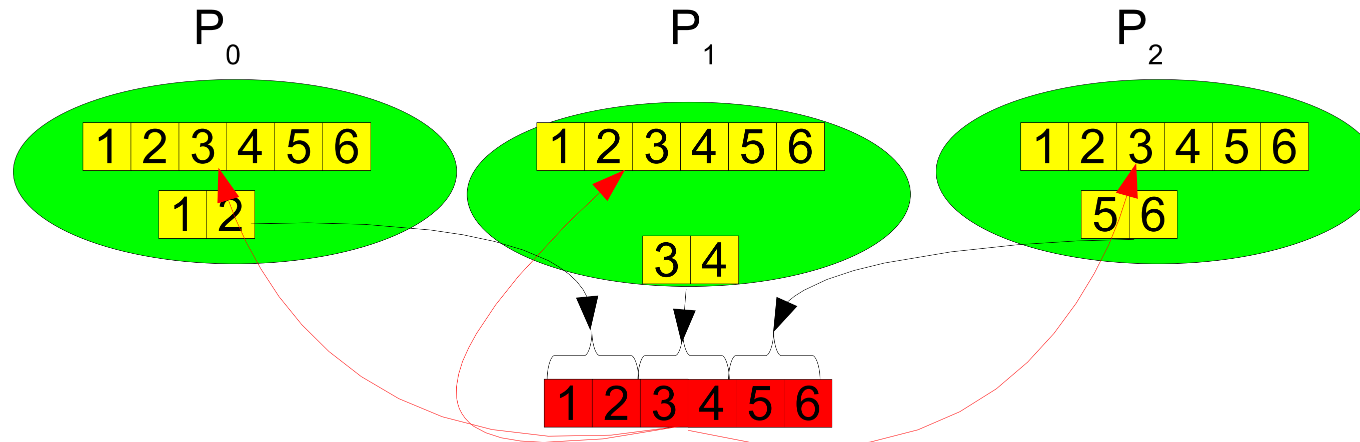


- Odwrotność operacji scatter. Jest wykonywana przy pomocy funkcji

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

- Proces źródłowy `root` odbiera od każdego innego procesu (i od siebie) wektor o adresie `sendbuf` i o długości `sendcnt` każda.
- Argument `recvcnt` specyfikuje liczbę elementów odebranych od pojedynczego nadawcy a nie całkowitą liczbę odebranych elementów. Elementy zostaną zapisane do bufora odbiorczego pod adresem `recvbuf`. Wszystkie procesy muszą dostarczyć identyczną wartość `recvcnt`.
- Proces o numerze i odbiera elementy począwszy od numeru i*`sendcnt`.

Operacja allgather



- Jest to operacja gather, której wynik przesyłany jest wszystkim procesom – brak procesu źródłowego `root`.

```
int MPI_Allgather(void *sendbuf,int sendcnt,MPI_Datatype sendtype,
void *recvbuf,int recvcnt,MPI_Datatype recvtype,MPI_Comm comm);
```

- Proces źródłowy `root` odbiera od każdego innego procesu (i od siebie) wektor o adresie `sendbuf` i o długości `sendcnt` każda.
- Argument `recvcnt` specyfikuje liczbę elementów odebranych od pojedynczego nadawcy a nie całkowitą liczbę odebranych elementów. Elementy zostaną zapisane do bufora odbiorczego pod adresem `recvbuf`. Wszystkie procesy muszą dostarczyć identyczną wartość `recvcnt`.
- Proces o numerze i odbiera elementy począwszy od numeru i*`sendcnt`.

Operacje gather i scatter ze zmienną liczbą elementów.

- Do tej pory zakładaliśmy, że każdy proces wysyła identyczną liczbę elementów. MPI dostarcza również operacje, w których możemy podać różną liczbę elementów dla różnych procesów. Ma ona prototyp:

```
int MPI_Gatherv (void *sendbuf, int sendcnt,MPI_Datatype
sendtype, void *recvbuf, int *recvcnts, int *displs,
MPI_Datatype recvtype,int root, MPI_Comm comm)
```

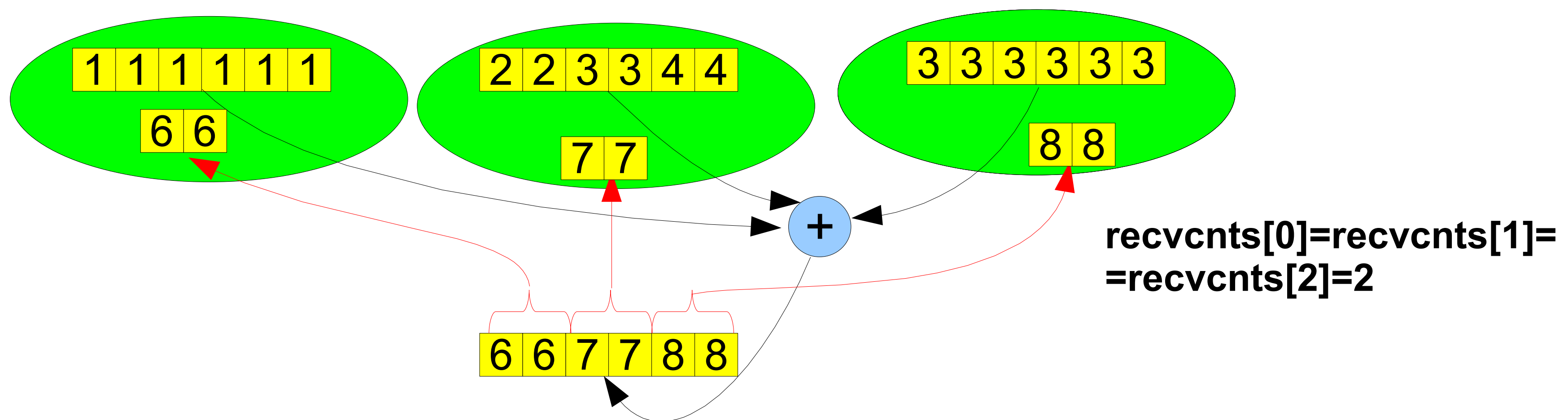
- Liczba wysyłanych elementów przez proces i to parametry `sendcount` oraz `recvcnts[i]` (tablica !!!). Mogą one być różne dla różnych procesów.

- Wartość `displs[i]` (tablica !!!) oznacza miejsce, w które w buforze recvbuf będą zapisane elementy wysłane przez proces i.

- Dostępna jest również operacja `MPI_Allgatherv` dla operacji allgather oraz operacja `MPI_Scatterv` o prototypie:

```
int MPI_Scatterv (void *sendbuf,int *sendcnts,int *displs,
MPI_Datatype sendtype,void *recvbuf,int recvcnt,
MPI_Datatype recvtype, int root,MPI_Comm comm )
```

Operacja reduce-scatter

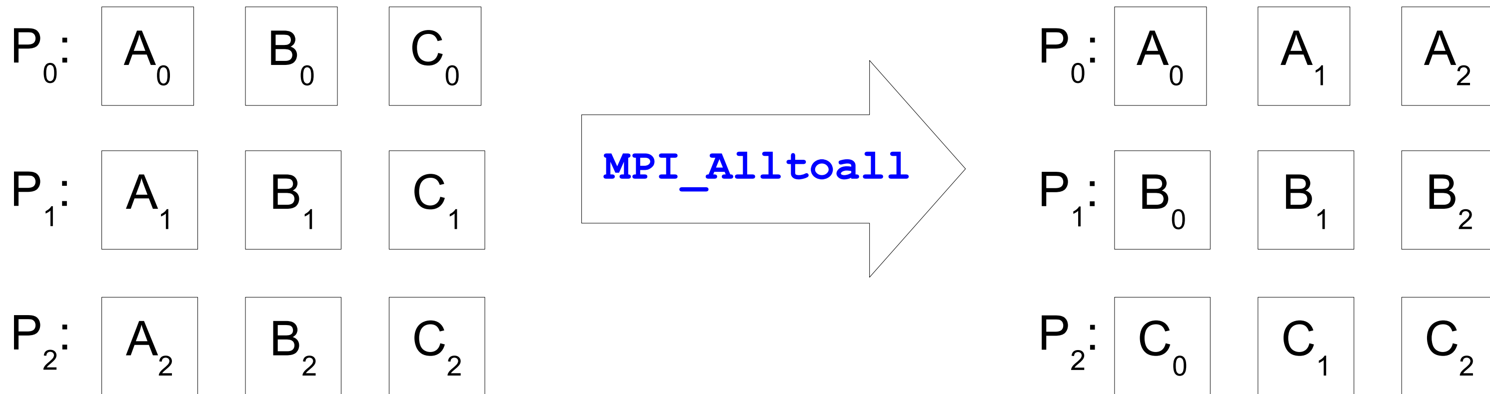


- Połączenie funkcji `MPI_Reduce` z `MPI_Scatterv`. Jest wykonywana przy pomocy funkcji:

```
int MPI_Reduce_scatter ( void *sendbuf, void *recvbuf, int *recvcnts,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

- Operacja redukcji jest wykonywana na buforze o liczbie elementów równej sumie liczb z tablicy `recvcnts`.
- Proces i otrzymuje `recvcnts[i]` elementów.
- Wszystkie procesy muszą dostarczyć te same wartości parametrów `datatype`, `op`, `comm` oraz identyczną tablicę `recvcnts`.

Operacja All-to-All



- Każdy proces wykonuje jednocześnie operacje scatter/gather wysyłając części swojego bufora nadawczego `sendbuf` innym procesom i odbierając do bufora odbiorczego `recvbuf` części wysłane przez inne procesy.

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

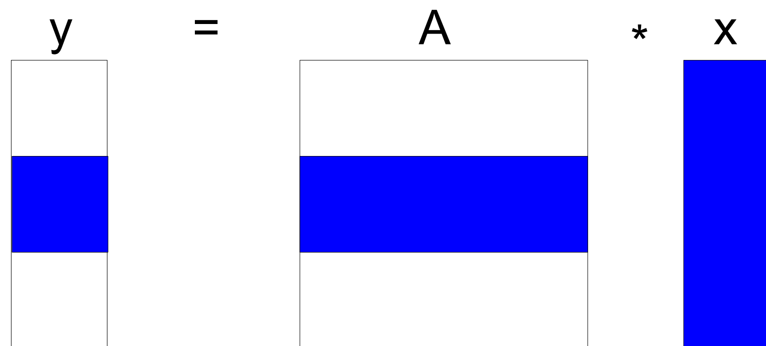
- W powyższym przykładzie `sendcount=recvcnt=1`. Istnieje wersja (`MPI_Alltoallv`) pozwalająca na wysłanie różnej liczby elementów różnym procesom.

Przykład: mnożenie macierzy i wektora

- Kwadratowa macierz double A[N][N], Wektory double x[N], y[N].
- Operacja macierzowa y=A*x;
- Kod szeregowy:

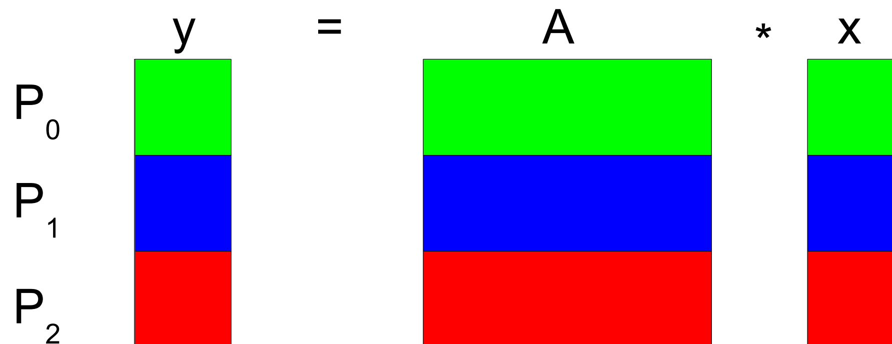
```
for(int i=0;i<N;i++) {
   y[i]=0;
   for(int j=0;j<N;j++)
      y[i]=y[i]+A[i][j]*x[j];
}
```

- i - ty element wektora y to wynik przemnożenia i-tego wiersza macierzy A przez wektor x.



Dystrybucja wierszami

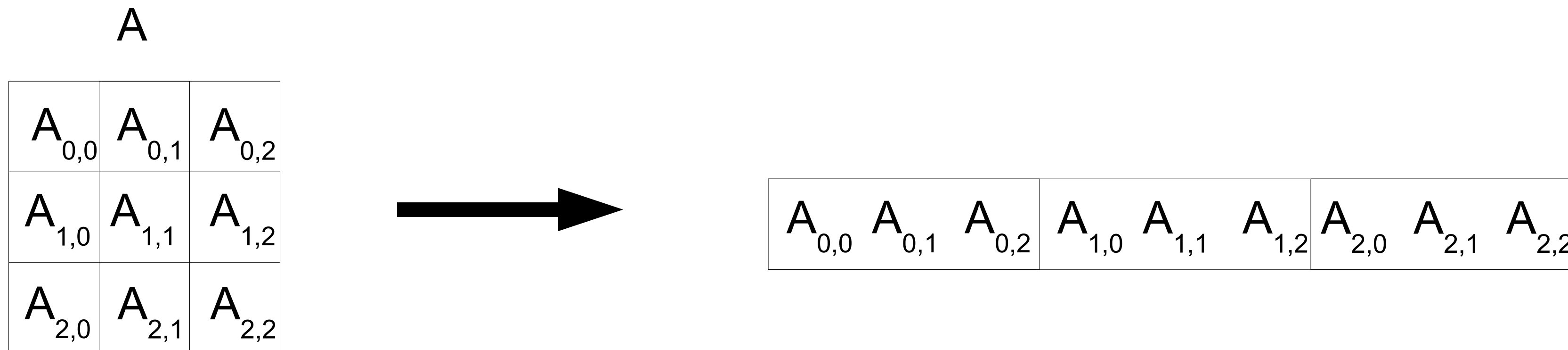
- Każdy proces otrzymuje i przechowuje N/p (p jest liczbą procesów) wierszy macierzy A.
- Ponadto każdy proces otrzymuje N/p elementów wektora x oraz odpowiada za obliczenie N/p elementów wektora wynikowego y;
- Dystrybucja danych i wyników dla 3 procesów.



- Proces P1 mógłby obliczyć wartość swoich (niebieskich) elementów wektora y, ale potrzebuje do tego całego wektora x.
- Podobnie w przypadku procesów P_0 oraz P_2.
- Idea: skorzystać z operacji `MPI_Allgather`, umożliwiając zgromadzenie przez wszystkie procesy całego wektora x.

Reprezentacja macierzy A w pamięci

- Pamięć jest tablicą jednowymiarową, zaś macierz A tablicą dwuwymiarową. Jak odwzorować ją w jednowymiarowej pamięci ?



- Deklaracja macierzy A:

```
double *A;
```

- Alokacja pamięci macierzy NxN:

```
A=malloc(sizeof(double)*N*N);
```

Adresowanie elementu $A_{i,j}$: (numeracja indeksów od 0)

A[i*N+j]

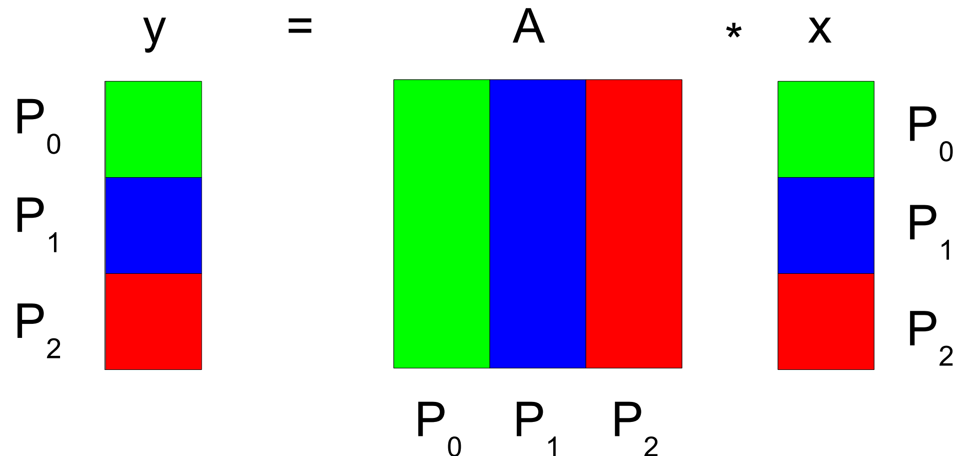
Kod funkcji mnożącej (Grama i wsp.)

```
void MultiplyRowwise(int N, double *A,double *x,double *y)
// A część macierzy przechowywana przez proces
// x oraz y części wektorów
   int np; // liczba procesów
   double fullx[N]; // pełny wektor x; rozszerzenie gnu języka C i C++

   MPI_Comm_size(MPI_COMM_WORLD,&npes);
// Liczba wierszy macierzy A oraz elementów wektorów x i y
// przechowywanych przez proces
   int nlocal=N/np; // Zakładamy, że N dzieli się przez np
// Gromadzimy wszyskie elementy wektora x w fullx;
   MPI_Allgather(x,nlocal,MPI_DOUBLE,fullx,nlocal,MPI_DOUBLE,
   MPI_COMM_WORLD);
// Obliczamy elementy wektora y za które odpowiada proces
   for(int i=0;i<nlocal;i++) {
      y[i]=0.0;
      for(int j=0;j<N;j++)
         y[i]+=A[i*N+j]*fullx[j];
   }
}
```

Alternatywna dystrybucja macierzy kolumnami

- Każdy proces otrzymuje N/p kolumn macierzy A i odpowiada za N/p elementów wektorów x oraz y.



- Każdy proces oblicza częściowy iloczyn skalarny dla swoich kolumn macierzy A.
- Częściowe iloczyny skalarne są sumowane funkcją `MPI_Reduce` dając wektor y
- Wektor y jest rozpraszany pomiędzy procesy funkcją `MPI_Scatter`.
- Kod zostawiam jako ***pracę domową***.