Programowanie w modelu przesyłania komunikatów – specyfikacja MPI

Środowisko przesyłania komunikatów MPI

- Rodzaje procedur:
  - blokujące
  - nieblokujące
- Tryby przesyłania komunikatów
  - standardowy – implementacja MPI decyduje o szczegółach realizacji przesłania komunikatu
  - specjalny – jawnie wskazany przez programistę
- Wybór trybu specyficznego oznacza określenie możliwych dodatkowych mechanizmów przesyłania oraz definicję zakończenia operacji wysyłania (co z kolei wskazuje na możliwość ponownego użycia bufora danych)

Środowisko przesyłania komunikatów MPI

- Procedury dwupunktowego przesyłania komunikatów:
  - Przesyłanie nieblokujące – procedura natychmiast przekazuje sterowanie dalszym instrukcjom programu
    - int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *req)
    - int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *req)
  - Rola zmiennej *req
  - W ramach par send-receive można łączyć dowolne kombinacje procedur blokujących i nieblokujących

Środowisko przesyłania komunikatów MPI

- Procedury związane z przesyłaniem nieblokującym
  - int MPI_Wait(MPI_Request *preq, MPI_Status *pstat)
  - int MPI_Test(MPI_Request *preq, int *pflag, MPI_Status *pstat) (wynik testu w zmiennej *pflag)
  - dodatkowe warianty powyższych: MPI_Waitany, MPI_Waitall, MPI_Testany, MPI_Testall
- Procedury testowania przybycia komunikatów (bez odbierania) – dwie wersje blokująca i nieblokująca
  - int MPI_Probe(int src, int tag, MPI_Comm comm, MPI_Status *stat)
  - int MPI_Iprobe(int src, int tag, MPI_Comm comm, int* flag, MPI_Status *stat)
- I wiele innych (MPI_Send_init, MPI_Start, MPI_Sendrecv, MPI_Cancel, itd.)

Środowisko przesyłania komunikatów MPI

- Istotą przykładu jest nakładanie się obliczeń i komunikacji, co w pewnych przypadkach może znacznie podnieść wydajność programu

```
MPI_Request request1, request2;
MPI_Status status1, status2;
MPI_Irecv( &datarecv, num, MPI_INT, source, tag, MPI_COMM_WORLD, &request1 );
// obliczenia nie wymagające danych odbieranych, przygotowanie danych do wysłania
MPI_Isend( &datasent, num, MPI_INT, dest, tag, MPI_COMM_WORLD, &request2 );
// obliczenia nie zmieniające danych wysyłanych
MPI_Wait( &request1, &status1 );
printf("Dane od procesu o randze: %d (%d)\n", source, status.MPI_SOURCE );
MPI_Wait( &request2, &status2 );
printf("Wysłano dane do procesu o randze: %d \n", dest );
```

Przykład konkretny – symulacja w czasie

```
for(;;){
  for(i...) A^{t+1}[i] = A^t[i-1] + A^t[i] + A^t[i+1]; // tzw. schemat różnicowy
  for(i...) A^t[i] = A^{t+1}[i];
}
```

- Obszary pamięci dla procesu: prywatne aktualizowane, wysyłane aktualizowane, otrzymywane
- Działanie najprostsze: wysyłamy, odbieramy, aktualizujemy prywatne i wysyłane – okazuje się, że musimy czekać
- Modyfikacja: odbieramy najpierw, potem wysyłamy i aktualizujemy -> zakleszczenie
- Modyfikacja – nakładanie obliczeń i komunikacji: odbieramy nieblokująco, wysyłamy, aktualizujemy.
- Implementacje na następnych slajdach posługują się globalnymi indeksami w tablicy A
  - w rzeczywistości każdy proces powinien pracować na swoim fragmencie tablicy z indeksami od 0 do rozmiar_lokalny

Implementacja blokująca

```
for(i=0;i<N;i++) A[i]=...;                  // zainicjuj tablicę wartości w chwili t=0
my_first = N/size*rank; if(my_first==0) my_first=1;          // określ zakres ...
my_last = N/size*(rank+1)-1; if(my_last==N-1) my_last=N-2;  // ... swojej podtablicy
                // elementy A[0] i A[N-1] są obliczane z zadanych warunków brzegowych
for(iter=0;iter<10000;iter++){      // w pętli po kolejnych iteracjach – chwilach czasu
  if(rank>0) MPI_Send( &A[my_first],1,MPI_INT,rank-1,...);       // wyślij swoje dane ...
                                                    // ... z lewego krańca do lewego sąsiada
  if(rank<size-1) // odbierz dane od prawego sąsiada (wysłane w poprzedniej linii)
      MPI_Recv( &A[my_last+1],1,MPI_INT,rank+1,...,&status);
  if(rank<size-1) MPI_Send( &A[my_last],1,MPI_INT,rank+1,...); // wyślij swoje dane ...
                                                    // ... z prawego krańca do prawego sąsiada
  if(rank>0) // odbierz dane od lewego sąsiada (wysłane w poprzedniej linii)
      MPI_Recv( &A[my_first-1],1,MPI_INT,rank-1,...,&status);
  for(i=my_first;i<=my_last;i++) B[i] = A[i-1]+A[i]+A[i+1]; // zaktualizuj wartości
  for(i=my_first;i<=my_last;i++) A[i]=B[i];// przepisz uzyskane wartości jako dane ...
}                                                   // ... początkowe dla następnej chwili czasu
```

Implementacja nieblokująca

```
for(i=0;i<N;i++) A[i]=...;                  // zainicjuj tablicę wartości w chwili t=0
my_first = N/size*rank; if(my_first==0) my_first=1;                  // określ zakres ...
my_last = N/size*(rank+1)-1; if(my_last==N-1) my_last=N-2;  // ... swojej podtablicy
for(iter=0;iter<10000;iter++){       // w pętli po kolejnych iteracjach – chwilach czasu
  if(rank>0)                                                         // zainicjuj odbieranie ...
    MPI_Irecv( &A[my_first-1],1,MPI_INT,rank-1,...,&req_1);   // ... wartości na krańcach ...
  if(rank<size-1)                                                    // ... swojej podtatblicy ...
    MPI_Irecv( &A[my_last+1],1,MPI_INT,rank+1,...,&req_2); // ... od sąsiadów
  if(rank>0) MPI_Send( &A[my_first],1,MPI_INT,rank-1,...);   // wyślij swoje dane ...
  if(rank<size-1) MPI_Send( &A[my_last],1,MPI_INT,rank+1,..);  // ... do sąsiadów
  for(i=my_first+1;i<=my_last-1;i++) B[i]=A[i-1]+A[i]+A[i+1];// zaktualizuj ...
  if(rank>0) MPI_Wait(&req_1, &status);                  // ^... wartości w swojej podtablicy
  if(rank<size-1) MPI_Wait(&req_2, &status); // poczekaj na dane przysłane przez sąsiadów
  i=my_first; B[i]=A[i-1] + A[i] + A[i+1]; // posługując się danymi od sąsiadów zaktualizuj
  i=my_last; B[i]=A[i-1] + A[i] + A[i+1]; // ... wartości na krańcach swojej podtablicy
  for(i=my_first;i<my_last;i++) A[i]=B[i];    // przepisz uzyskane wartości jako dane ...
}                                                                    // ... początkowe dla następnej chwili czasu
```

MPI – tryby komunikacji

- Tryby komunikacji
  - buforowany (MPI_Bsend, MPI_Ibsend) – istnieje jawnie określony bufor używany do przesyłania komunikatów; wysyłanie jest zakończone w momencie skopiowania komunikatu do bufora
  - synchroniczny (MPI_Ssend, MPI_Issend) – wysyłanie jest zakończone po otrzymaniu informacji, że została wywołana procedura odbierania
  - gotowości (MPI_Rsend, MPI_Irsend) – system gwarantuje, że procedura odbierania jest gotowa, tak więc wysyłanie jest kończone natychmiast

MPI – Typy danych

- podobnie jak w językach programowania istnieją elementarne typy danych (predefiniowane obiekty typu MPI_Datatype), takie jak MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_BYTE
- podobnie jak w językach programowania istnieją utworzone typy danych, które dają możliwość prostego odnoszenia się do całości złożonych z wielu zmiennych elementarnych
- definicja typu danych określa sposób przechowywania w pamięci zmiennych typów elementarnych tworzących zmienną danego typu
- definicję typu można przedstawić w postaci tzw. mapy typu postaci: $\{(typ_1, odstęp_1), ..., (typ_n, odstęp_n)\}$

MPI – typy danych

- int MPI_Type_contiguous(int count,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
  - zmienna złożona z count zmiennych typu oldtype występujacych w pamięci bezpośrednio po sobie

- int MPI_Type_vector(int count, int blocklength, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
  - zmienna złożona z count bloków zmiennych typu oldtype, każdy z bloków o długości blocklength i o początku w pamięci w odległości stride zmiennych typu oldtype od początku bloku poprzedzającego

MPI – typy danych

- Procedura przekazania do użycia utworzonego typu:
  - int MPI_Type_commit(MPI_Datatype *newtype)
  - typy danych określone w procedurach wysyłania i przyjmowania komunikatu nie muszą być takie same (ważne, aby odnosiły sie do tej samej mapy w pamięci, tzn. odczytywały te same zmienne typów elementarnych)

```
MPI_Type_contiguous( 4, MPI_FLOAT, &nowy_typ );
MPI_Type_commit ( &nowy_typ );
MPI_Send( &a, 4, MPI_FLOAT, ..... ); // proces wysyłający
MPI_Recv( &a, 1, nowy_typ, ..... ); // proces odbierający
```

Typy danych MPI – przykład 1

- Transpozycja macierzy (tablicy dwuwymiarowej) w trakcie komunikacji:

```
int l_kol, l_wier;   double A[l_wier][l_kol]; double B[l_kol][l_wier];
MPI_Datatype typ_kolumna;
MPI_Type_vector( l_wier, 1, l_kol, MPI_DOUBLE, &typ_kolumna);
MPI_Type_commit( &typ_kolumna );
for( i=0; i<l_kol; i++ ){
    MPI_Send( &A[0][i], 1, typ_kolumna, dest, tag,
              MPI_COMM_WORLD);
}
for( i=0; i<l_kol; i++ ){
    MPI_Recv( &B[i][0], l_wier, MPI_DOUBLE, source, tag,
              MPI_COMM_WORLD, &status);
}
```

MPI – typy danych

- int MPI_Type_indexed(int count, int* tablica_długości_bloków, int* tablica_odstępów, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - zmienna złożona z count bloków zmiennych typu oldtype, każdy z kolejnych bloków o długości zapisanej w tablicy tablica_długości_bloków i o początku w pamięci w odległości zapisanej w tablicy tablica_odstępów wyrażonej w liczbie zmiennych typu oldtype
- int MPI_Type_create_struct(int count, int* tablica_długości_bloków, MPI_Aint* tablica_odstępów, MPI_Datatype* tablica_typów, MPI_Datatype *newtype)
  - jak wyżej, ale teraz każdy blok może być złożony ze zmiennych innego typu, zgodnie z zawartością tablicy tablica_typów, a odstępy wyrażone są w bajtach

MPI – typy danych

- Przy tworzeniu nowych typów przydatne są operacje (MPI-2):
  - int MPI_Get_address(void* location, MPI_Aint* address) - zwracająca adresy zmiennych w pamięci oraz
  - int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint* lb, MPI_Aint* extent) - zwracająca zasięg zmiennej danego typu
  - int MPI_Type_size(MPI_Datatype datatype, int* size) - zwracająca rozmiar (w bajtach) zmiennej danego typu
  - int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype* newtype) - rozszerzająca definicję typu o możliwe wyrównanie w pamięci (funkcja może być konieczna jeśli chce się przesłać np. wiele struktur jednym poleceniem send lub gdy chce się utworzyć typ złożony z wielu struktur)

Typy danych MPI – przykład 2

```
struct rekord{   double skalar;  char znak;  float wektor[3];   };
struct rekord   baza[20000000];
MPI_Datatype rekord1_typ, rekord2_typ;
int tab_dlug_blokow[3] = {1, 1, 3 };
MPI_Datatype tab_typow[3] = { MPI_DOUBLE, MPI_CHAR, MPI_FLOAT };
MPI_Aint  lb, zasieg, podstawa, tab_odstepow[3];
MPI_Get_address( &baza[0].skalar, &tab_odstepow[0] );
MPI_Get_address( &baza[0].znak, &tab_odstepow[1] );
MPI_Get_address( &baza[0].wektor[0], &tab_odstepow[2] );
podstawa =  tab_odstepow[0] ;
for( i=0; i<3; i++ ) tab_odstepow[i] -= podstawa ;
MPI_Type_create_struct( 3, tab_dlug_blokow, tab_odstepow, tab_typow,
   &rekord1_typ );
MPI_Type_get_extent( rekord1_typ, &lb, &zasieg );
MPI_Type_create_resized( rekord1_typ, lb, zasieg, &rekord2_typ);
```

MPI – typ danych spakowanych

- int MPI_Pack(void* buf_dane, int count, MPI_Datatype typ, void* buf_send, int buf_send_size, int* pozycja, MPI_Comm comm) - pakowanie count zmiennych o typie typ i o początku w pamięci buf_dane do bufora buf_send o rozmiarze buf_send_size; pozycja jest pozycją końca danych w buforze wysyłania, i jednocześnie rozmiarem spakowanej paczki
- int MPI_Unpack(void* buf_recv, int buf_recv_size, int* pozycja, void* buf_dane, int count, MPI_Datatype typ, MPI_Comm comm) - rozpakowanie paczki
- typ MPI_PACKED stosuje się tak jak predefiniowane typy elementarne i nowo tworzone typy, z tym że liczba zmiennych jest teraz rozmiarem paczki w bajtach (rozmiar można uzyskać używając MPI_Pack_size)

Typ MPI_PACKED - przykład

```
struct rekord{   double skalar;  char znak;  float wektor[3];   };
struct rekord   baza[20000000];
int rozm, rozm_pakietu, pozycja; void* bufor; // lub char bufor[10000000000];
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &rozmiar);
rozm_pakietu = rozm;
MPI_Pack_size(1, MPI_CHAR, MPI_COMM_WORLD, &rozmiar);
rozm_pakietu += rozm;
MPI_Pack_size(3, MPI_FLOAT, MPI_COMM_WORLD, &rozmiar);
rozm_pakietu += rozm;
bufor = (void *)malloc(3*rozm_pakietu);  pozycja = 0;
for( i=0; i<3; i++ ) {
      MPI_Pack(&baza[i].skalar,1,MPI_DOUBLE,bufor,3*rozm_pakietu,&pozycja,MCW);
      MPI_Pack(&baza[i].znak, 1, MPI_CHAR, bufor, 3*rozm_pakietu, &pozycja, MCW);
     MPI_Pack(&baza[i].wektor[0],3,MPI_FLOAT,bufor,3*rozm_pakietu,&pozycja,MCW);
}
MPI_Send( bufor, pozycja, MPI_PACKED, 1, 0, MPI_COMM_WORLD );
```

MPI - dynamiczne zarządzanie procesami

- MPI-2 umożliwia dynamiczne zarządzanie procesami, choć nie umożliwia dynamicznego zarządzania zasobami
- Tworzenie nowych procesów w trakcie działania programu:

  int MPI_Comm_spawn(char* command, char* argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm* intercomm, int array_of_errcodes[])

  - command - oznacza plik wykonywalny, argv - listę argumentów, maxprocs - jest liczbą tworzonych procesów (system może utworzyć mniej procesów), info - informacja dla konkretnego środowiska wykonania, root, comm - identyfikacja procesu realizującego operację, intercomm - zwracany uchwyt do *interkomunikatora*, array_of_errcodes - tablica z kodami błędów

MPI - dynamiczne zarządzanie procesami

- MPI_Comm_spawn jest operacją grupową dla procesów z komunikatora comm i kończy działanie kiedy wywołane zostanie MPI_Init dla nowo utworzonych procesów
- Po utworzeniu nowych procesów (z nowym komunikatorem MPI_COMM_WORLD) można nawiązać komunikację z nimi za pomocą zwróconego interkomunikatora intercomm
- Wersja MPI_Comm_spawn_multiple umożliwia rozpoczęcie wielu nowych procesów (za pomocą tego samego pliku binarnego z różnymi argumentami lub za pomocą różnych plików binarnych) w ramach pojedynczego nowego komunikatora

MPI - dynamiczne zarządzanie procesami

```
int main( int argc, char *argv[] ) {
    int np = NUM_SPAWNS;  int errcodes[NUM_SPAWNS];
    MPI_Comm parentcomm, intercomm;
    MPI_Init( &argc, &argv );
    MPI_Comm_get_parent( &parentcomm );
    if (parentcomm == MPI_COMM_NULL) {
        MPI_Comm_spawn( "spawn_example.exe",  MPI_ARGV_NULL, np,
            MPI_INFO_NULL, 0, MPI_COMM_WORLD, &intercomm, errcodes );
        printf("Proces rodzic\n");
    }   else   {
        printf("Proces dziecko\n");
    }
    fflush(stdout);
    MPI_Finalize();
}
```

Interkomunikatory

- Interkomunikator jest środkiem do realizacji komunikacji pomiędzy dwoma niezależnymi (rozłącznymi) grupami procesów (zwanych umownie lewą i prawą grupą procesów)
- Dla procesu uczestniczącego w komunikacji za pomocą interkomunikatora grupa procesów, do której on należy, jest grupą lokalną, a pozostała grupa jest grupą odległą
- MPI-2 definiuje znaczenie (realizację) operacji grupowych w przypadku, gdy komunikatorem jest interkomunikator, a nie komunikator standardowy - intra-komunikator

Interkomunikatory

- Komunikacja dwupunktowa z wykorzystaniem interkomunikatora przebiega podobnie jak w przypadku standardowym (intra-komunikatora)
- Składnia wywołania procedur wysłania i odbioru jest identyczna
- Dla procesu wysyłającego, argument określający cel komunikatu jest rangą procesu odbierającego w ramach jego grupy, czyli grupy odległej
- Dla procesu odbierającego, podobnie, źródło komunikatu oznacza rangę procesu wysyłającego w ramach grupy odległej

Zarządzanie procesami i interkomunikatory

- Procesom realizującym procedurę MPI_Comm_spawn (procesom rodzicom) uchwyt do interkomunikatora jest zwracany przez tę procedurę
- Procesy utworzone za pomocą MPI_Comm_spawn (procesy dzieci) mogą uzyskać uchwyt do interkomunikatora za pomocą polecenia MPI_Comm_get_parent
- Dla procesów rodziców procesami odległymi są dzieci, dla dzieci odległymi są rodzice

Przykład stosowania interkomunikatora

```
strcpy(slave,"slave.exe") ; num = 3;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
printf("MASTER : spawning 3 slaves ... \n");
MPI_Comm_spawn(slave, MPI_ARGV_NULL, num, MPI_INFO_NULL,0,
    MPI_COMM_WORLD, &inter_comm, array_of_errcodes);
printf("MASTER : send a message to master of slaves ...\n");
MPI_Send(message_0, 50, MPI_CHAR,0 , tag, inter_comm);
MPI_Recv(message_1, 50, MPI_CHAR, 0, tag, inter_comm, &status);
printf("MASTER : message received : %s\n", message_1);
MPI_Send(master_data, 50, MPI_CHAR,0 , tag, inter_comm);
MPI_Finalize();
exit(0);
}
```

Przykład stosowania interkomunikatora

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
MPI_Comm_get_parent(&inter_comm);
if ( my_rank == 0 )   {
    MPI_Recv(message_0, 50, MPI_CHAR, 0, tag, inter_comm, &status);
    printf("SLAVE (master) : message received : %s\n", message_0);
    MPI_Send(message_1, 50, MPI_CHAR,0 , tag, inter_comm);
    MPI_Recv(master_data, 50, MPI_CHAR, 0, tag, inter_comm, &status);
    printf("SLAVE (master) : master data received : %s\n", master_data);
    strcpy(slaves_data, master_data);
}
MPI_Bcast(slaves_data, 50, MPI_CHAR, 0, MPI_COMM_WORLD);
printf("SLAVE - %d : slaves data  received : %s\n", my_rank, slaves_data);
MPI_Finalize();
```

Grupy procesów, komunikatory i interkomunikatory

- Poza funkcjonowaniem w ramach dynamicznego zarządzania procesami, grupy procesów, komunikatory i interkomunikatory mogą być tworzone, modyfikowane, niszczone za pomocą specjalnych niezależnych procedur (m.in. MPI_Comm_create, MPI_Comm_split, MPI_Intercomm_create, MPI_Comm_group)

- Procesom w ramach komunikatorów może być przyporządkowana konkretna topologia (relacja sąsiedztwa) w celu efektywniejszej realizacji operacji grupowych dla specjalnych algorytmów i specyficznych topologii połączeń międzyprocesorowych (np. topologia kartezjańska dla połączeń o topologii kraty i algorytmów macierzowych)

Równoległe wejście/wyjście w MPI-2

- MPI-2 udostępnia interfejs umożliwiający procesom równoległą realizację plikowych operacji wejścia/wyjścia
- Interfejs oparty jest o pojęcia typu elementarnego (*etype*) typu plikowego (*filetype*) i widoku pliku (*view*)
  - typ elementarny oznacza najczęściej albo bajt albo dowolny typ MPI (predefiniowany lub skonstruowany przez użytkownika)
  - typ plikowy jest sekwencją zmiennych typu elementarnego z ewentualnymi miejscami pustymi; typ plikowy określa wzorzec dostępu do pliku
  - widok pliku jest złożony z: przesunięcia (liczby bajtów od początku pliku), typu elementarnego (pojedynczy plik jest związany z pojedynczym typem elementarnym) i typu plikowego
- Każdy proces posiada własny widok otwartego pliku

Równoległe wejście/wyjście w MPI-2

- Odczyt z otwartego pliku dokonywany jest w miejscu wskazywanym przez offset lub wskaźnik pliku (*file pointer*)
  - offset oznacza odstęp od początku pliku wyrażany w liczbie zmiennych typu elementarnego
  - wskaźnik pliku oznacza niejawny offset zarządzany przez implementację MPI
- Wskaźniki pliku mogą być indywidualne (każdy proces posiada własny wskaźnik pliku) lub wspólne (wszystkie procesy uczestniczące w operacjach we/wy na danym pliku posiadają jeden wskaźnik)

Równoległe wejście/wyjście w MPI-2

- Podstawowe operacje na plikach obejmują:
  - MPI_File_open, MPI_File_close - otwarcie i zamknięcie pliku przez wszystkie procesy należące do komunikatora będącego jednym z argumentów procedur; procedura otwarcia przyjmuje dodatkowe argumentów określające sposób dostępu do pliku i zwraca uchwyt do pliku używany w operacjach odczytu i zapisu
  - MPI_File_set_view - ustanowienie widoku pliku
  - MPI_File_read_at, MPI_File_write_at - odczyt i zapis w pozycji określonej przez offset
  - MPI_File_read, MPI_File_write - odczyt i zapis w pozycji określonej przez indywidualne wskaźniki pliku (uaktualniane automatycznie, choć mogą też być jawnie ustawiane przez MPI_File_seek)
- Podane procedury odczytu i zapisu są blokujące, istnieją także nieblokujące wersje tych procedur

Równoległe wejście/wyjście w MPI-2

```
#define FILESIZE (1024 * 1024)
int main(int argc, char **argv) {
    // definicje zmiennych, inicjowanie środowiska MPI
    MPI_File fh;     MPI_Status status;
    bufsize = FILESIZE/nprocs;
    buf = (int *) malloc(bufsize);
    nints = bufsize/sizeof(int);
    MPI_File_open(MPI_COMM_WORLD, "/tmp/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);
    free(buf);
    MPI_Finalize();
}
```

MPI-2

- Poza dotychczas omówionymi elementami, MPI-2 wprowadza także kilka innych mechanizmów:
  - mechanizm dostępu do pamięci odległej (komunikacja jednostronna – *one-sided communication*, RMA – *remote memory access)* – podobnie jak w OpenMP (operacje inicjacji bloku pamięci jako wspólnego: MPI_Win_create, operacje zapisu/odczytu: MPI_Put, MPI_Get, MPI_Accumulate, operacje synchronizacji: MPI_Win_fence, MPI_Win_lock, MPI_Win_unlock i inne)
  - mechanizm ustanawiania i realizacji komunikacji między niezależnymi procesami - podobnie jak gniazda Unixa, ale z użyciem interkomunikatorów (operacje: MPI_Open_port, MPI_Comm_accept, MPI_Comm_connect i inne; za pomocą MPI_Comm_join można korzystać z gniazd Unixowych)
  - i szereg innych drobniejszych zmian i uzupełnień