MPI dla zaawansowanych

Interdyscyplinarne Centrum Modelowania

Matematycznego i Komputerowego

Uniwersytet Warszawski

http://www.icm.edu.pl

Maciej Cytowski

m.cytowski@icm.edu.pl

Maciej Szpindler

m.szpindler@icm.edu.pl

Wersja 1.PL, 20 Październik 2009

Program

- **Standard MPI – przypomnienie**
- Co nowego w MPI-2
- Dynamiczne zarządzanie procesami w MPI-2
- Komunikacja jednostronna w MPI-2
- Równoległe I/O
- Paraver – diagnostyka aplikacji równoległych

Podstawy: Biblioteka MPI

- MPI - **Message Passing Interface**
  - Biblioteka (standard) do tworzenia programów równoległych
  - Pracuje natywnie z C,C++,Fortranem
  - Faktyczny standard programowania w oparciu o wymianę komunikatów
    - Wersja 1.2 (MPI-1) – wprowadza około 150 funkcji
    - Wersja 2.1 (MPI-2) – rozszerzona o nową funkcjonalność, razem ponad 500 funkcji
- Składa się z zestawu procedur i funkcji
- Różne implementacje
  - MPICH / MPICH2 - *http://www-unix.mcs.anl.gov/mpi/mpich/*
  - **OpenMPI** - *http://www.open-mpi.org/*
  - Intel MPI ($$$)
  - HP MPI ($$$)

Podstawy: model równoległości

- **Założenia**
  - Pamięć jest rozproszona = model pamięci rozproszonej – procesy wymieniają się komunikatami
  - Jawna równoległość
- Proces odpowiada fizycznej jednostce obliczeniowej
- Program składa się z procesów – podprogramów działających na poszczególnych procesorach
- Stworzona dla komputerów masywnie równoległych
  - Potrafi obsłużyć setki procesów / procesów
  - Wiele implementacji dobrze pracuje na architekturach z pamięcią współdzieloną
- SPMD – **Single Program Multiple Data**
  - ten sam podprogram pracuje na każdym procesorze (wiele instancji tego samego programu)

Podstawy: Komunikator

- Podstawowa struktura programu równoległego używającego MPI
- **Komunikator** – zbiór procesów mogących się wzajemnie komunikować
- W ramach komunikatora każdy proces otrzymuje unikalny identyfikator (numer) tzw. **rank**
- Informacja o komunikatorze zawarta jest w zmiennej typu **`MPI_COMM`**
- W czasie inicjacji programu biblioteka tworzy domyślny komunikator o nazwie **`MPI_COMM_WORLD`** zawierający wszystkie dostępne procesy
- Programista może definiować własne komunikatory zawierające podzbiory procesów

MPI: Pobieranie informacji z komunikatora

- Liczba (np) dostępnych procesów w ramach komunikatora

```
MPI_Comm_size(MPI_COMM_WORLD, &np)
```

- Numer (rank) mojego procesu

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank)
```

  - (Uwaga Fortran) rank = 0, ..., np – 1

```
C:
 int MPI_Comm_size ( MPI_Comm comm, int *np );
 int MPI_Comm_rank ( MPI_Comm comm, int *rank );

Fortran:
 CALL MPI_COMM_SIZE ( integer comm, integer np, integer ierror )
 CALL MPI_COMM_RANK ( integer comm, integer rank, integer ierror )
```

Podstawy: Startowanie i zamykanie sesji

- Wywołania wszelkich funkcji i procedur biblioteki MPI muszą znajdować się pomiędzy instrukcjami:

  `MPI_Init(&argc, &argv)`

  - Inicjuje działanie biblioteki i domyślnego komunikatora (sesję)

  `MPI_Finalize(void)`

  - Kończy działanie sesji MPI
- Funkcje `Init` i `Finalize` muszą być wywołane przez wszystkie procesy w ramach sesji

```
C:
 int MPI_Init ( int *argc, int **argv  );
 int MPI_Finalize ( );

Fortran:
 CALL MPI_INIT( integer ierror )
 CALL MPI_FINALIZE ( integer ierror )
```

Podstawy: przesyłanie komunikatów

- Podstawowa metoda przesłania komunikatu pomiędzy dwoma procesami
  - Nadawca wywołuje funkcję **Send**
  - Odbiorca wywołuje funkcję **Receive**
- Poszczególne komunikaty odróżnia etykieta (**tag**)
- `MPI_Send` / `MPI_Recv` to funkcje blokujące tzn. blokują dalsze wykonanie programu do czasu zakończenia komunikacji

Komunikacja nieblokująca

- Komunikacja realizowana w 3 krokach:
  - inicjalizacja nieblokującej komunikacji
    - natychmiast zwraca wykonanie do procesu
    - nazwa funkcji rozpoczyna się od MPI_I…
  - wykonanie pracy, obliczenia, inna komunikacja
  - czekanie na zakończenie komunikacji nieblokującej

- Nieblokujące wysłanie:

```
MPI_Isend(…)
//wykonanie pracy
MPI_Wait(…)
```

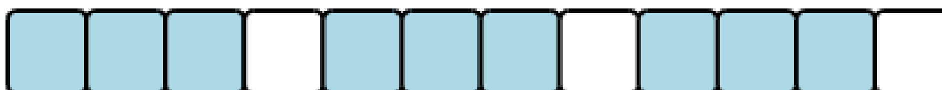
- Nieblokujący odbiór:

```
MPI_Irecv(…)
//wykonanie pracy
MPI_Wait(…)
```

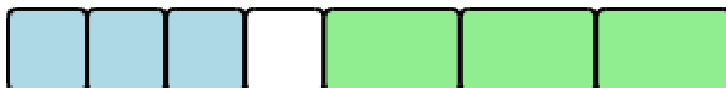
Definiowalne typy danych MPI

- MPI umożliwia nam definiowanie 3 rodzajów własnych typów:
  - typy reprezentujące ciągły fragment pamięci zawierający elementy tego samego typu – `MPI_Type_Contiguous`

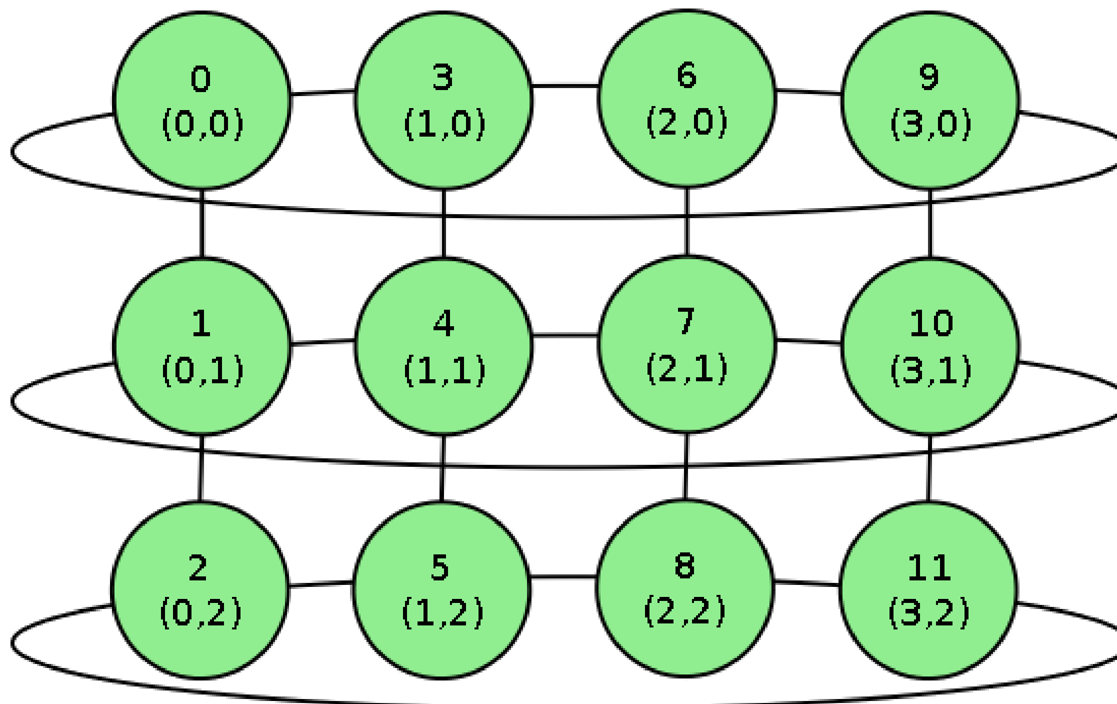
  - typy będące wektorami elementów danego typu, które nie reprezentują ciągłego fragmentu pamięci – `MPI_Type_Vector`



  - typy będące strukturami zawierającymi elementy różnych typów, które niekoniecznie reprezentują ciągły fragment pamięci – `MPI_Type_Struct`



- Wirtualne topologie to mechanizm pozwalający nazywać procesy w komunikatorze w sposób, który lepiej pasuje do schematu komunikacji pomiędzy procesami
- Wirtualnych topologii użyjemy na przykład jeśli obliczenia wykonywane są na dwu-wymiarowej siatce i każdy proces komunikuje się tylko i wyłącznie ze swoimi najbliższymi sąsiadami w siatce



Komunikacja grupowa

- Komunikacja grupowa, czyli taka, w której uczestniczy grupa procesów MPI
- Dana funkcja wywoływana jest przez wszystkie procesy w komunikatorze
- W MPI wyróżniamy następujące typy komunikacji grupowej:
  - Bariera synchronizacyjna
  - operacja Broadcast – nadaj wiadomość wielu procesom
  - operacja typu rozrzuć (z ang. scatter)
  - operacja typu zbierz (z ang. gather)
  - operacja wszyscy do wszystkich
  - operacja redukcji (np. globalna suma, globalne maksimum,..)

Program

- Standard MPI – przypomnienie
- **Co nowego w MPI-2**
- Dynamiczne zarządzanie procesami w MPI-2
- Komunikacja jednostronna w MPI-2
- Równoległe I/O
- Paraver – diagnostyka aplikacji równoległych

Co nowego w MPI-2?

- Uściślenia standardu MPI 1.1
- MPI I/O – obsługa równoległego zapisu i odczytu z plików
- Komunikacja jednostronna
- Zarządzanie procesami

Jak sprawdzić wersję MPI?

- W wersji 1.2 standardu MPI wprowadzono możliwość kontroli nad wersją używanych bibliotek
- Podczas kompilacji:

```
#define MPI_VERSION 2
#define MPI_SUBVERSION 2
```

- Podczas uruchomienia:

```
int MPI_Get_version(int *version, int *subversion)
```

`MPI_Get_version` może być uruchomione przed `MPI_Init()` i po `MPI_Finalize()`

Program

- Standard MPI – przypomnienie
- Co nowego w MPI-2
- **Dynamiczne zarządzanie procesami w MPI-2**
- Komunikacja jednostronna w MPI-2
- Równoległe I/O
- Paraver – diagnostyka aplikacji równoległych

Dynamiczne zarządzanie procesami

- Program korzystający z MPI ma formę wielu procesów działających w ramach komunikatora
  - Standard MPI-1 zakłada statyczny zbiór procesów
  - Wszystkie procesy są tworzone przy starcie programu
  - Statyczny komunikator
    - Tworzony przy wywołaniu funkcji **`MPI_Init`**
    - Istnieje do wywołania funkcji **`MPI_Finalize`**
- MPI-2 wprowadza nowe możliwości
  - Tworzenie nowych procesów MPI
  - Łączenie niezależnie wystartowanych procesów
  - Przekazywanie pomiędzy procesami zmiennych typu **`MPI_Info`**

Rozmnażanie procesów

- Rodzic
  - Inicjuje tworzenie nowych procesów

  `MPI_Info_create, MPI_Info_set`

  - Tworzy informacje o procesie tworzącym

  `MPI_Comm_spawn`

  - Tworzy wspólny komunikator rodzica i potomków

  `MPI_Comm_spawn`

- Potomek
  - Posiada własny komunikator
  - Uzyskuje informacje o procesie tworzącym - rodzicu

  `MPI_Get_parent`

Rozmnażanie identycznych procesów

- Procesy rozmnaża funkcja

```
MPI_Comm_spawn (command, argv[], maxprocs, info,
                root, comm, intercomm,
                errcodes[])
```

  - Z poziomu procesu **root** inicjuje identyczne nowe procesy komendą **command** z parametrami **argv[]**
  - Nowe procesy pracują w komunikatorze **comm**
  - Jest ich maksymalnie **maxprocs**
  - Funkcja zwraca tzw. inter-komunikator **intercomm**, pomiędzy pierwotną grupą procesów (rodzica) i nową grupą procesów (potomków)
  - Struktura **info** zawiera zbiór par klucz-wartość, które proces rodzic przekazuje procesowi potomnemu

Rozmnażanie procesów

C:

```
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[]);
```

Fortran:

```
 CALL MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM,
                     INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

CHARACTER*(*) COMMAND, ARGV(*)
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES(*), IERROR
```

Tworzenie wielu różnych procesów

- Różne procesy rozmnaża funkcja

```
MPI_Comm_spawn_multiple (count, array_of_commands[],
array_of_argv[],varray_of_maxprocs[], array_of_info[],
int root, comm, intercomm, array_of_errcodes[])
```

  - Z poziomu procesu **root** inicjuje **count** nowych procesów
  - Tablice **array_of_commands** i **array_of_argv**[] zawierają nazwy programów i parametry dla nowych procesów
  - Nowe procesy pracują w komunikatorze **comm**
  - Funkcja zwraca inter-komunikator **intercomm**, pomiędzy pierwotną grupą procesów (rodzica) i nową grupą procesów (potomków)
  - Struktura **info** zawiera zbiór par klucz-wartość, które proces rodzic przekazuje procesom potomnym

Rozmnażanie różnych procesów

C:

```
int MPI_Comm_spawn_multiple(int count, \
                            char *array_of_commands[], \
                            char* *array_of_argv[],    \
                            int array_of_maxprocs[],   \
                            MPI_Info array_of_info[],  \
                            int root, MPI_Comm comm,   \
                            MPI_Comm *intercomm,       \
                            int array_of_errcodes[]);
```

Komunikatory potomne

- **MPI_Comm_ get_parent(MPI_Comm *parent)**
  - Zwraca dowiązanie do komunikatora rodzica
  - Jeżeli proces nie został stworzony przy pomocy jednej z funkcji
    - `MPI_Comm_spawn`
    - `MPI_Comm_spawn_multiple`
  - lub proces rodzica został
    - odłączony
    - zakończony
  - zwraca wartość `MPI_COMM_NULL`

Komunikatory potomne

C:

```
int MPI_Comm_get_parent(MPI_Comm *parent);
```

Fortran:

```
 CALL MPI_COMM_GET_PARENT(PARENT, IERROR)

 INTEGER PARENT, IERROR
```

Obiekty typu MPI_Info

- Pusty obiekt typu **`MPI_Info`** tworzy
  - **`MPI_Info_create( info )`**
- Dodawanie zawartość klucz-wartość obiektu
  - **`MPI_Info_set(info, key, value)`**
- Usuwanie zawartości klucz-wartość
  - **`MPI_Info_delete(info, key)`**
- Dodatkowo do zarządzanie obiektami typu **`MPI_Info`** służą
  - **`MPI_Info_dup`** – duplikuje obiekt
  - **`MPI_Info_free`** – zwalnia obiekt
  - **`MPI_Info_get`** – zwraca wartość odpowiadająca kluczowi
  - **`MPI_Info_get_nkeys`** – zwraca liczbę zdefiniowanych kluczy

Obiekty typu MPI_Info

C:

```
int MPI_Info_create(MPI_Info *info);

int MPI_Info_set(MPI_Info info, char *key, char *value);

int MPI_Info_delete(MPI_Info info, char *key);
```

Fortran:

```
 CALL MPI_INFO_CREATE(INFO, IERROR)

 CALL MPI_INFO_SET(INFO, KEY, VALUE, IERROR)

 CALL MPI_INFO_DELETE(INFO, KEY, IERROR)

 INTEGER INFO, IERROR
 CHARACTER*(*) KEY, VALUE
```

Łączenie procesów

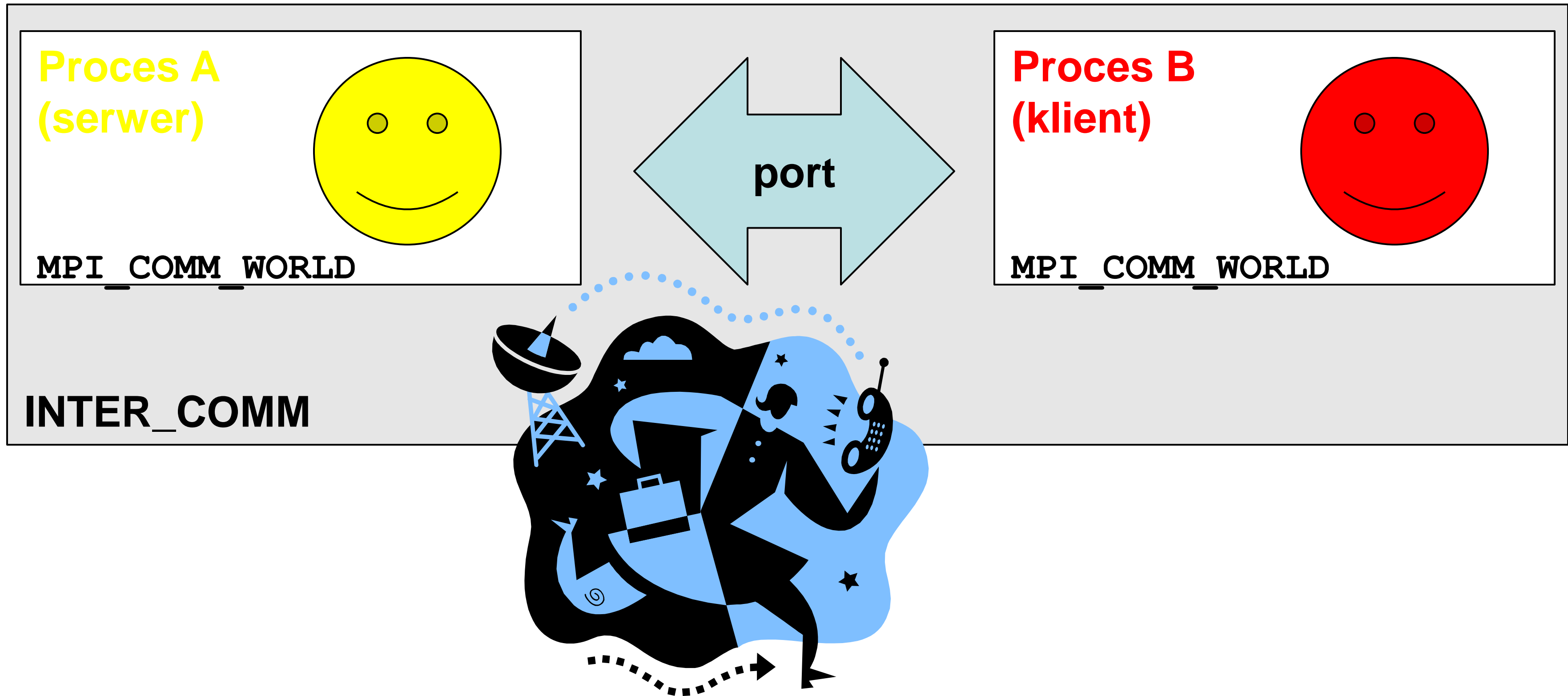
- Inne metody ustanowienia komunikacji między dwoma niezależnymi procesami MPI
  - `MPI_Comm_join(fd, intercomm)`
    - Tworzy nowy inter-komunikator **intercomm** dla procesów połączonych socketem **fd**
    - **fd** musi być socketem typu `SOCK_STREAM`
    - brak funkcji MPI do manipulacji socketami, programista musi użyć funkcji systemowych
- Łączenie procesów za pośrednictwem portu
  - `MPI_Comm_connect(port, info, root, comm, intercomm)`
    - Wysyła prośbę utworzenia nowego inter-komunikatora
  - `MPI_Comm_accept(port, info, root,comm, intercomm)`
    - Akceptuje utworzenie nowego inter-komunikatora
  - Otwieranie i zamykanie portów MPI
    - `MPI_Open_port(info, port_name)`
    - `MPI_Close_port(port_name)`

Łączenie procesów

C:

```
int MPI_Comm_connect(char *port_name, MPI_Info info, int root,\
                     MPI_Comm comm, MPI_Comm *newcomm);

int MPI_Comm_accept(char *port_name, MPI_Info info, int root, \
                    MPI_Comm comm, MPI_Comm *newcomm);

int MPI_Comm_join(int fd, MPI_Comm *intercomm);

int MPI_Open_port(MPI_Info info, char *port_name);

int MPI_Close_port(char *port_name);
```

Łączenie procesów: przykład



Łączenie procesów: przykład

- Proces A – serwer
  - Otwiera port
  - Wypisuje numer portu
  - Akceptuje połączenie

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;

MPI_Init(&argc, &argv);
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
MPI_Comm_accept(myport, MPI_INFO_NULL, 0,
    MPI_COMM_SELF, &intercomm);
…
MPI_Finalize();
```

- Proces B – klient
  - Pobiera numer portu od użytkownika
  - Wysyła prośbę połączenia za pomocą portu

```
char name[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;

MPI_Init(&argc, &argv);
printf("enter port name: ");
gets(name);

MPI_Comm_connect(name, MPI_INFO_NULL, 0,
    MPI_COMM_SELF, &intercomm);
….
MPI_Finalize();
```

Interakcyjne łączenie procesów za pomocą portów

Łączenie procesów – ciąg dalszy

- Proces komunikujący się za pośrednictwem portów MPI może rozgłaszać swoje porty
  - **`MPI_Publish_name(service_name, port_name)`**
    - Publikuje parę **service_name**, **port_name**
    - Inny proces może odebrać numer portu jeżeli zna konkretną wartość **service_name**
  - **`MPI_Lookup_name(service_name, info, port_name)`**
    - Odbiera numer portu **port_name** opublikowany z daną nazwą (wartością) **service_name**
    - Program musi zapewnić odpowiednio duży bufor do składowania wszystkich potencjalnych nazw portów (rozmiaru **`MPI_MAX_PORT_NAME`**)

Łączenie procesów

```
C:
int MPI_Publish_name(char *service_name, MPI_Info *info,
                     char *port_name);

int MPI_Lookup_name( char *service_name, MPI_Info *info,
                     char *port_name);


Fortran:
 CALL MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)

 CALL MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)

 INTEGER INFO, IERROR
 CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

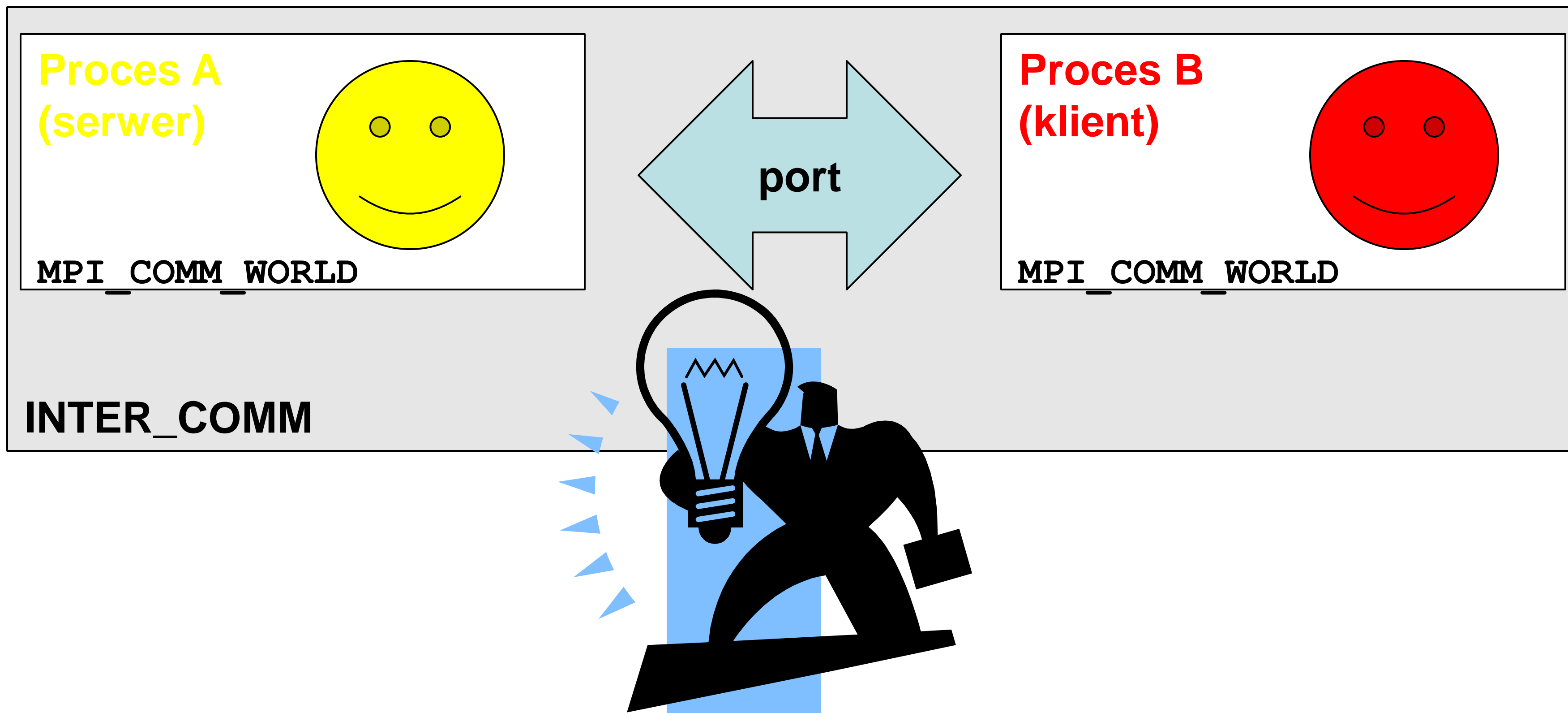
Odłączanie procesu od komunikatora

- Odłączenie procesu od komunikatora następuje poprzez wywołanie
  - **`MPI_Comm_disconnect(comm)`**
    - Procedura czeka na zakończenie wszelkich zainicjowanych operacji komunikacyjnych
    - Następnie odłącza proces od komunikatora **comm** i ustawia komunikator dla procesu na **`MPI_COMM_NULL`**
  - Nie można odłączyć procesu od komunikatorów:
    - **`MPI_COMM_WORLD`**
    - **`MPI_COMM_SELF`**

Odłączanie procesu od komunikatora

```
C:
int MPI_Comm_disconnect(MPI_Comm *comm);

Fortran:
 CALL MPI_COMM_DISCONNECT(COMM, IERROR)

 INTEGER COMM, IERROR
```

Łączenie procesów: przykład nieinteraktywny



Łączenie procesów: przykład nieinteraktywny

- Proces A – serwer
  - Otwiera port
  - Rozgłasza numer portu
  - Akceptuje połączenie

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;

MPI_Init(&argc, &argv);
MPI_Open_port(MPI_INFO_NULL, myport);
MPI_Publish_name("ala", MPI_INFO_NULL,
    myport);
MPI_Comm_accept(myport, MPI_INFO_NULL, 0,
    MPI_COMM_SELF, &intercomm);
…
MPI_Unpublish_name("ala", MPI_INFO_NULL,
    myport);
MPI_Finalize();
```

- Proces B – klient
  - Znajduje numer portu rozgłaszany przez proces A
  - Wysyła prośbę połączenia za pomocą portu

```
char name[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;

MPI_Init(&argc, &argv);
MPI_Lookup_name("ala", MPI_INFO_NULL,
    name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0,
    MPI_COMM_SELF, &intercomm);
….
MPI_Finalize();
```

Łączenie procesów za pomocą portów z rozgłaszaniem

Rozmnażanie procesów: przykład

- Proces rodzic tworzy nowe procesy
- Nowe procesy są kopią rodzica

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define NUM_SPAWNS 2

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) {
        /* Rozmnazamy biezacy proces
        tworzac NUM_SPAWNS nowych
        procesow - program musi nazywac
        sie spawn-self */

        MPI_Comm_spawn( "spawn-self",
         MPI_ARGV_NULL, np,
         MPI_INFO_NULL,     0,
         MPI_COMM_WORLD, &intercomm,
         errcodes );

        printf("I'm the parent.\n");
    }  else {
        printf("I'm the spawned.\n");
    }
    fflush(stdout);
    MPI_Finalize();
    return 0;
}
```

Rozmnażanie procesów z użyciem MPI_SPAWN

Wskazówka autorów standardu MPI-2

- *Advice to users.*

It is possible in MPI to start a static SPMD or MPMD application by starting first one process and having that process start its siblings with MPI_COMM_SPAWN.

This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI-1 application.

Program

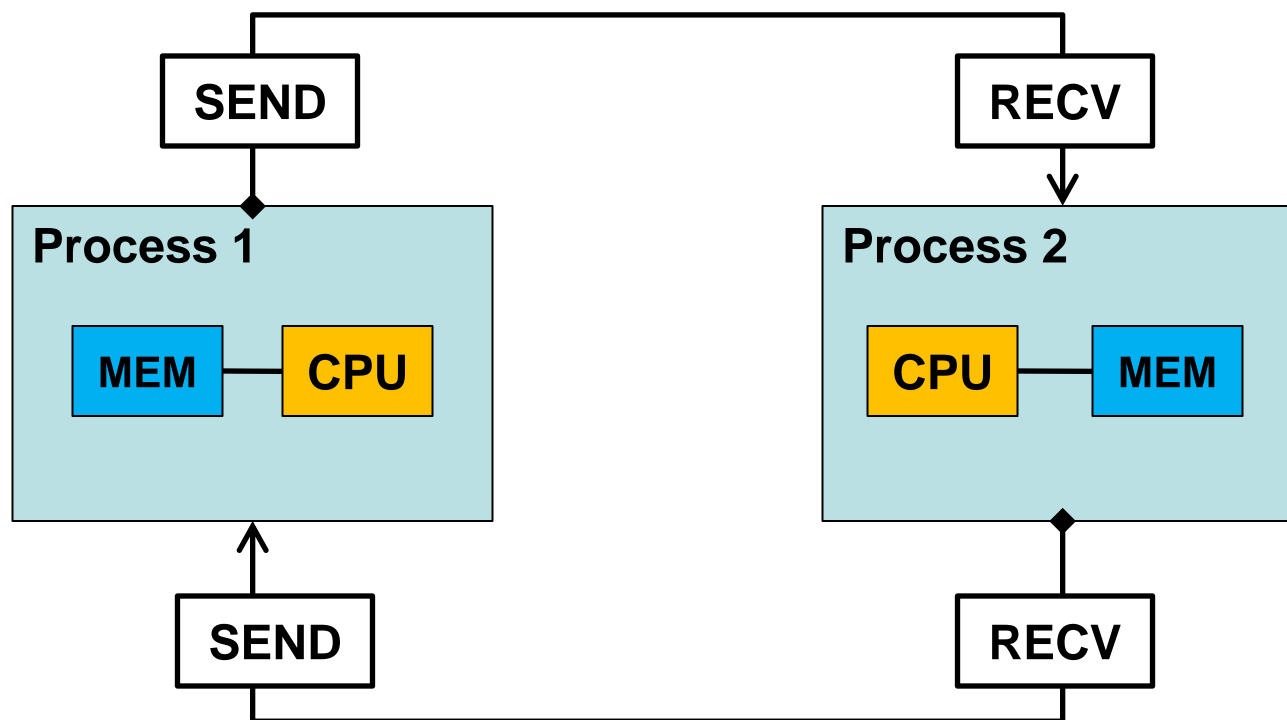
- Standard MPI – przypomnienie
- Co nowego w MPI-2
- Dynamiczne zarządzanie procesami w MPI-2
- **Komunikacja jednostronna w MPI-2**
- Równoległe I/O
- Paraver – diagnostyka aplikacji równoległych

Operacje jednostronne w MPI-2

- Poza operacjami komunikacyjnymi dwu-stronnymi definiowanymi przez standard MPI-1:
  - `MPI_Send` i `MPI_Xsend`
  - `MPI_Recv`
- MPI-2 wprowadza komunikację jednostronną
  - Operacja **PUT**
  - Operacja **GET**
- Odwołanie do pamięci innego procesu (procesora)
  - tzw. Remote Memory Access
- Komunikacja jednostronna oddziela przesyły danych i synchronizację
  - w przypadku `MPI_Send` niejawna synchronizacja
  - Operacje jednostronne wymagają specjalnych, jawnych wywołań funkcji synchronizujących odwołania do pamięci

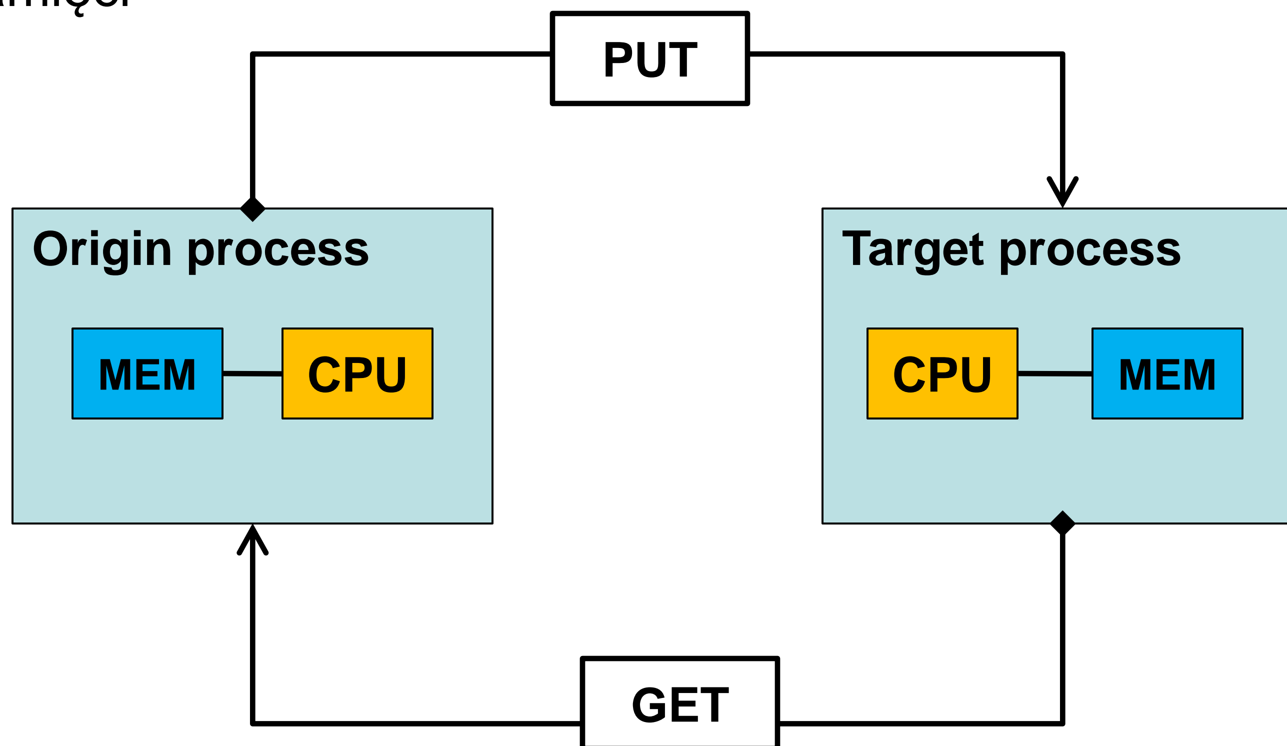
Komunikacja dwustronna MPI-1

- Nadawca i odbiorca muszą jawnie uczestniczyć w komunikacji
- Wywołania odpowiednich funkcji MPI Send/Recv
- W trybie blokującym niejawna synchronizacja



Komunikacja jednostronna

- Komunikacja jest inicjowana tylko przez jeden proces źródłowy (origin)
- Programista musi zapewnić odpowiednią kolejność odwołań do pamięci



- Ukrywa przed programistą subtelności związane z organizacją pamięci

Tworzenie okien dla dostępu do pamięci

- Tworzenie okna – wydzielanie fragmentu w już zaalokowanej pamięci procesu, do którego mogą mieć dostęp inne procesy (w trybie RMA)
- `MPI_WIN_CREATE( base_address, win_size, disp_unit, info, comm, win)`
  - Tworzy okno w pamięci procesu o rozmiarze **win_size** poczynając od **base_address**
  - Można wskazać jednostkę (czynnik skalujący) **disp_unit** dla wyznaczania przesunięcia (w bajtach)
  - Wywoływane dla każdego procesu w ramach inter-komunikatora **comm**
  - Zwraca obiekt **win** typu `MPI_Win`, który będzie stosowany w operacjach komunikacji jednostronnej
- `MPI_WIN_FREE(win)`
  - Zwalnia region pamięci przypisany do okna
  - Może być wykonane tylko po ukończonej synchronizacji

Tworzenie okien w trybie RMA

C:

```
MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
               MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

Fortran:

```
  CALL MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN,
                      IERROR)

<type> BASE(*)
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
```

Zwalnianie pamięci w oknie RMA

C:

```
int MPI_Win_free(MPI_Win *win)
```

Fortran:

```
 CALL MPI_WIN_FREE(WIN, IERROR)


INTEGER WIN, IERROR
```

Operacja MPI_Put

- Operacja odpowiadająca wysłaniu przez proces źródłowy i odpowiedniemu odebraniu przez proces docelowy
  - para: nadawca – MPI_Send, odbiorca – MPI_Recv, model komunikacji MPI-1
- Proces źródłowy ustala wszystkie parametry dla komunikacji
- ```
  MPI_PUT( origin_address, origin_count,
           origin_datatype, target_rank, target_disp,
           target_count, target_datatype, win)
  ```
- Proces źródłowy przesyła do pamięci procesu o numerze **target_rank** dane typu **origin_datatype** i rozmiarze **origin_count** spod adresu **origin_address**
- Dane zostają umieszczone w pamięci procesu **target_rank** pod adresem wyznaczonym przez okno **win** i przesunięcie **target_disp**
- Dane zostaną umieszczone jako typ **target_data_type** rozmiaru **target_count**
  ```

Operacja MPI_Put

C:

```
MPI_Put(void *origin_addr, int origin_count, \
        MPI_Datatype origin_datatype, int target_rank, \
        MPI_Aint target_disp, int target_count, \
        MPI_Datatype target_datatype, MPI_Win win);
```

Fortran:

```
CALL MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN,
IERROR)

<type> ORIGIN_ADDR(*)
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK
INTEGER TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
```

Operacja MPI_Get

- Działa podobnie do operacji **`MPI_Put`**, ale dane są przesyłane w odwrotnym kierunku
- ```
  MPI_GET( origin_address, origin_count,
           origin_datatype, target_rank, target_disp,
           target_count, target_datatype, win)
  ```
  - Pobiera fragment pamięci procesu **target_rank** i umieszcza pod adresem **origin_address**
  - Rozmiary i typy danych opisują **origin_count**, **origin_datatype** i **target_count**, **target_datatype** odpowiednio
  - Adres pobieranej pamięci jest wyznaczany jako przesunięcie **target_disp** względem początku okna **win**
- W celu ukończenia transferu należy wykonać synchronizacji okna **win**
- Lokalny bufor **origin_address** nie powinien być używany przed zakończeniem synchronizacji
  ```

Operacja MPI_Get

C:

```
MPI_Get(void *origin_addr, int origin_count, \
        MPI_Datatype origin_datatype, int target_rank, \
        MPI_Aint target_disp, int target_count, \
        MPI_Datatype target_datatype, MPI_Win win);
```

Fortran:

```
CALL MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN,
IERROR)

<type> ORIGIN_ADDR(*)
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK
INTEGER TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
```

Operacja MPI_Accumulate

- Akumuluje zawartość lokalnego bufora i docelowego bufora za pomocą wybranej operacji
-
```
MPI_ACCUMULATE(origin_address,
               origin_count,origin_datatype,
               target_rank,
               target_disp,target_count,
               target_datatype, op, win)
```
  - Wynik operacji **op** wykonanej na lokalnym buforze i zdalnym jest nadpisywana w pamięci zdalnej
  - Rozmiary i typy danych opisują **origin_count**, **origin_datatype** i **target_count**, **target_datatype** odpowiednio
  - Adres zdalnej pamięci jest wyznaczany jako przesunięcie **target_disp** względem początku okna **win**
  - Nie można definiować własnych operacji
  - Można używać operacje redukcji (`MPI_Reduce`)
  - Akumulacja jest atomowa – można wykonać wiele akumulacji z różnych źródeł dla tego samego zdalnego bufora

Operacja MPI_Accumulate

C:

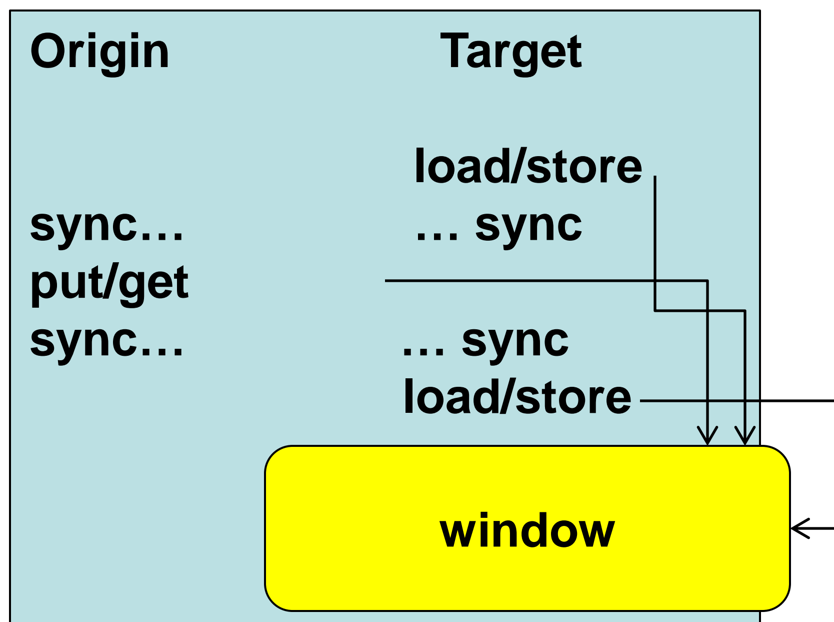
```
int MPI_Accumulate(void *origin_addr, int origin_count,
            MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Op op, MPI_Win
            win);
```

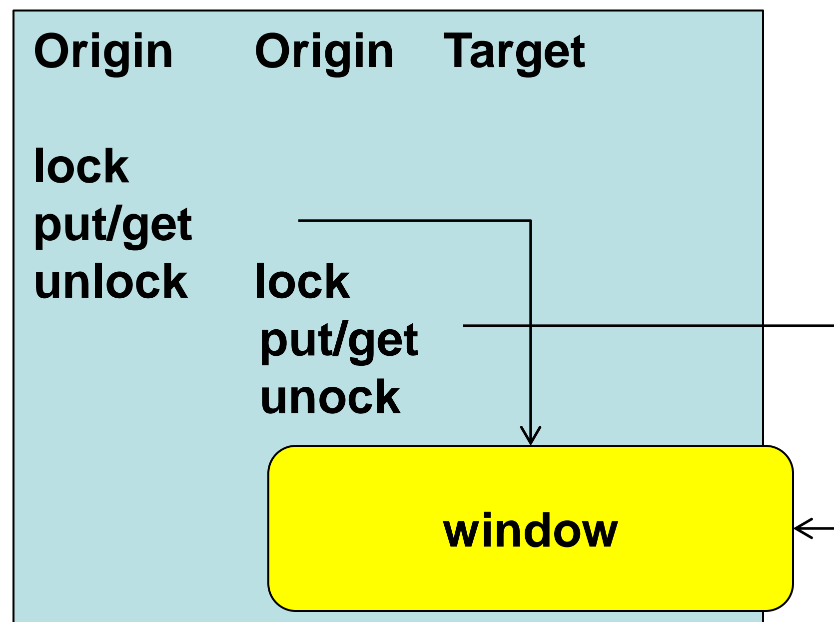
Fortran:

```
CALL MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
      TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE,
      OP, WIN, IERROR)

<type> ORIGIN_ADDR(*)
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
      TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
```

Synchronizacja w komunikacji jednostronnej

- Z udziałem procesu zdalnego (active target)
  - Proces zdalny uczestniczy jedynie w synchronizacji
  - Operacje fence
- Tylko z udziałem procesów źródłowych (passive target)
  - Proces zdalny nie uczestniczy jawnie w synchronizacji
  - Operacje lock/unlock





Synchronizacja okna pamięci

- `MPI_WIN_FENCE(assert, win)`
  - Synchronizuje wszystkie operacje RMA wykonywane na danym oknie pamięci **win**
  - Przypomina barierę
  - Wykonywana kolektywnie na procesach związanych z oknem
  - Powinna być używana do synchronizacji przed i po wywołaniu
    - `MPI_Put`
    - `MPI_Get`
    - `MPI_Accumulate`
  - Argument **assert** służy do podania wskazówek optymalizacyjnych

MPI_Win_fence

C:

```
int MPI_Win_fence(int assert, MPI_Win win)
```

Fortran:

```
 CALL MPI_WIN_FENCE(ASSERT, WIN, IERROR)

INTEGER ASSERT, WIN, IERROR
```

Inne mechanizmy synchronizacji w trybie RMA

- Tryb mniej restrykcyjny od WIN_FENCE
  - **`MPI_Win_start`**
  - **`MPI_Win_wait`**
  - **`MPI_Win_post`**
  - **`MPI_Win_complete`**
- Mechanizm zamków (lock) podobny jak w modelu programowania równoległego w oparciu o pamięć współdzieloną
  - **`MPI_Win_lock`**
  - **`MPI_Win_unlock`**

Program

- Standard MPI – przypomnienie
- Co nowego w MPI-2
- Dynamiczne zarządzanie procesami w MPI-2
- Komunikacja jednostronna w MPI-2
- **Równoległe I/O**
- Paraver – diagnostyka aplikacji równoległych

Równoległe I/O: Motywacja

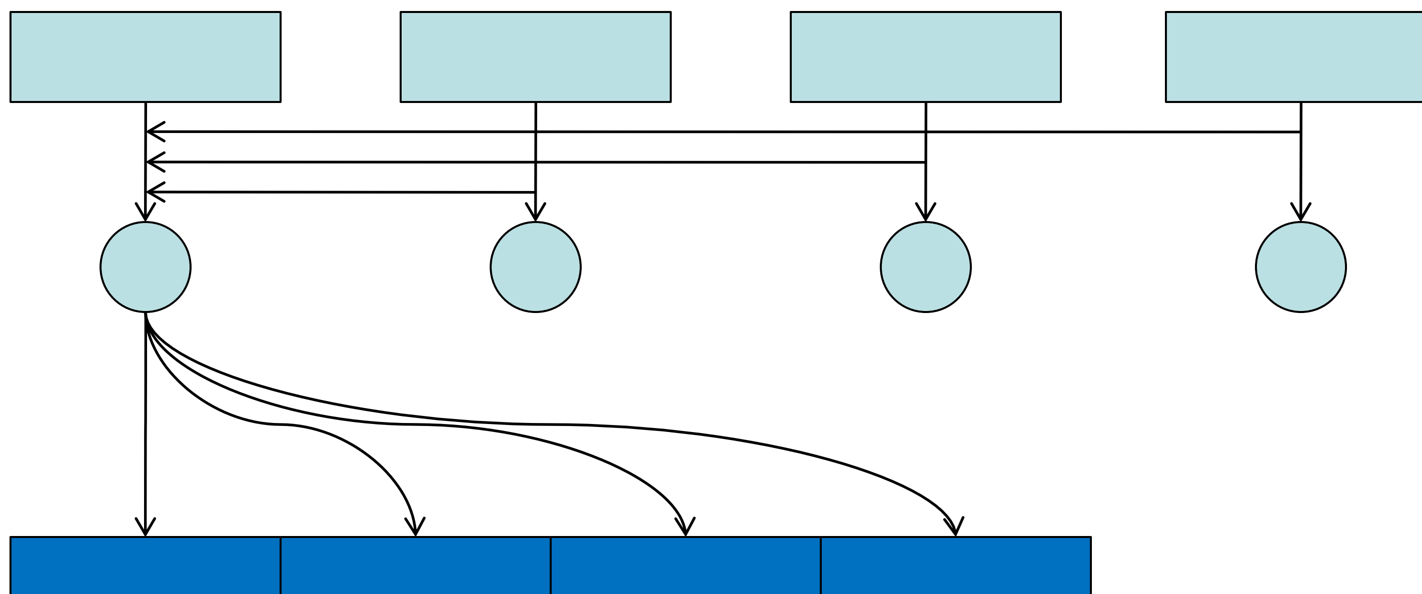
- Wiele aplikacji potrzebuje:
  - równoległego dostępu do pliku przez grupę procesów
  - jednoczesnego dostępu
  - aby procesy mogły czytać/zapisywać wiele (małych) nie-ciągłych fragmentów plików
  - aby procesy mogły czytać te same dane
- Analogia: zapis/czytanie pliku to jak wysłanie/odbieranie wiadomości
- Obsługa równoległego I/O wymaga:
  - Wsparcia dla grup procesorów
  - Wsparcia dla operacji grupowych w ramach komunikatora
  - Wsparcia dla nieblokujących operacji (ukrywanie I/O za obliczeniami)
  - Nieciągłego dostępu

Równoległe I/O

- Wykonywanie równoległego czytania i zapisu do plików
- Podstawowe operacje: open, close, read, write, seek
- Duża funkcjonalność:
  - Operacje blokujące/nie blokujące
  - Operacje atomowe/nie atomowe
  - Zdefiniowane offset-y / prywatne wskaźniki do plików / współdzielone wskaźniki do plików
  - Nieciągły dostęp do pliku i pamięci
- Wszystkie wywołania rozpoczynają się od `MPI_File_` (read, write, seek, close, …)
- Asynchroniczny odpowiednik: `_i`
- Operacje kolektywne: `_all` i ich rozczepianie: `_begin` `_end`
- Prawdziwa pozycja: `_at`
- Współdzielony wskaźnik do pliku: `_shared`

I/O sekwencyjne

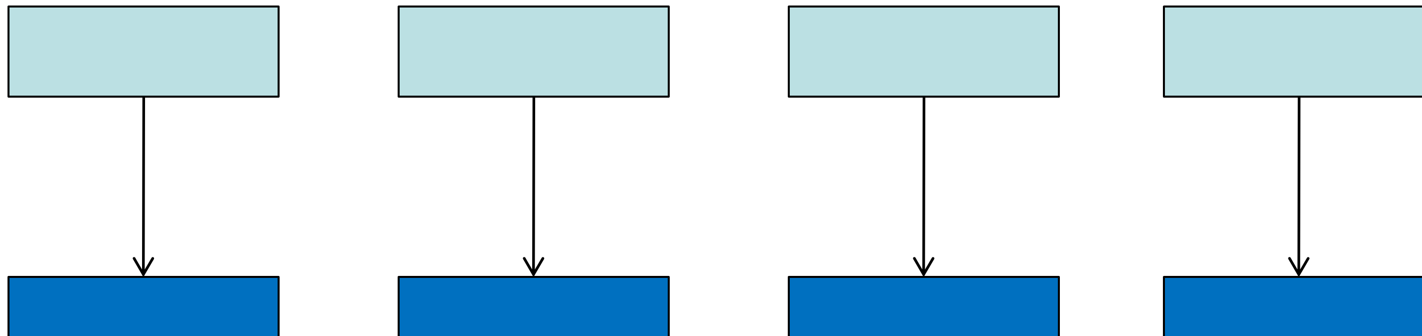
Każdy proces wysyła dane do procesu 0, proces 0 zapisuje do pliku



Przykład: numeryczne prognozy pogody COAMPS

I/O „trywialnie" równoległe

Każdy proces zapisuje do osobnego pliku



- Zalety: równoległość, doskonała wydajność
- Wady: obsługa wielu małych plików, trudności przy wczytywaniu danych dla innej liczby procesów

Co to jest równoległe I/O?

- Wiele procesów programu równoległego czyta/pisze dane z/do wspólnego pliku

Plik

P0 P1 P2 P(n-1)

Dlaczego MPI i równoległe I/O?

- Pisanie to jak wysyłanie wiadomości, a czytanie to jak ich otrzymywanie
- Każda implementacja równoległego I/O potrzebuje:
  - Obsługi operacji grupowych (kolektywnych)
  - Obsługi skomplikowanych, nieciągłych typów danych i nieciągłego ułożenia danych w RAM i plikach
  - Obsługi mechanizmu testowania/czekania na operacje nieblokujące
- Przydaje się tutaj całe mnóstwo funkcjonalności już wcześniej zaimplementowanej w MPI

Użycie MPI dla prostego I/O

Plik

P0 P1 P2 P(n-1)

Każdy proces musi wczytać (podobny) kawałek ze wspólnego pliku

Użycie lokalnych wskaźników do pliku

```
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, "/pfs/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);
```

Użycie jawnych offsetów

```
integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
            MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)

nints = FILESIZE / (nprocs*INTSIZE)
offset = rank * nints * INTSIZE

call MPI_FILE_READ_AT(fh, offset, buf, nints,
                      MPI_INTEGER, status, ierr)

call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
  print *, 'process ', rank, 'read ', count, 'integers'

call MPI_FILE_CLOSE(fh, ierr)
```

Hands-on: równoległe I/O

Zadanie:

- Każdy proces powinien wypisać swój numer (jako jeden znak) dziesięć razy z przesunięciem według wzoru:

  `my_rank + i*nprocs, i=0,..,9`

- Przykładowy wynik dla nprocs=3:
  “`0123012301230123012301230123012301230123`“

Hands-on: równoległe I/O

1. Utwórz prosty program MPI
2. Zadeklaruj nieznane jak dotąd zmienne:

   ```
   MPI_File fh;
   MPI_Offset offset;
   MPI_Status status;
   ```

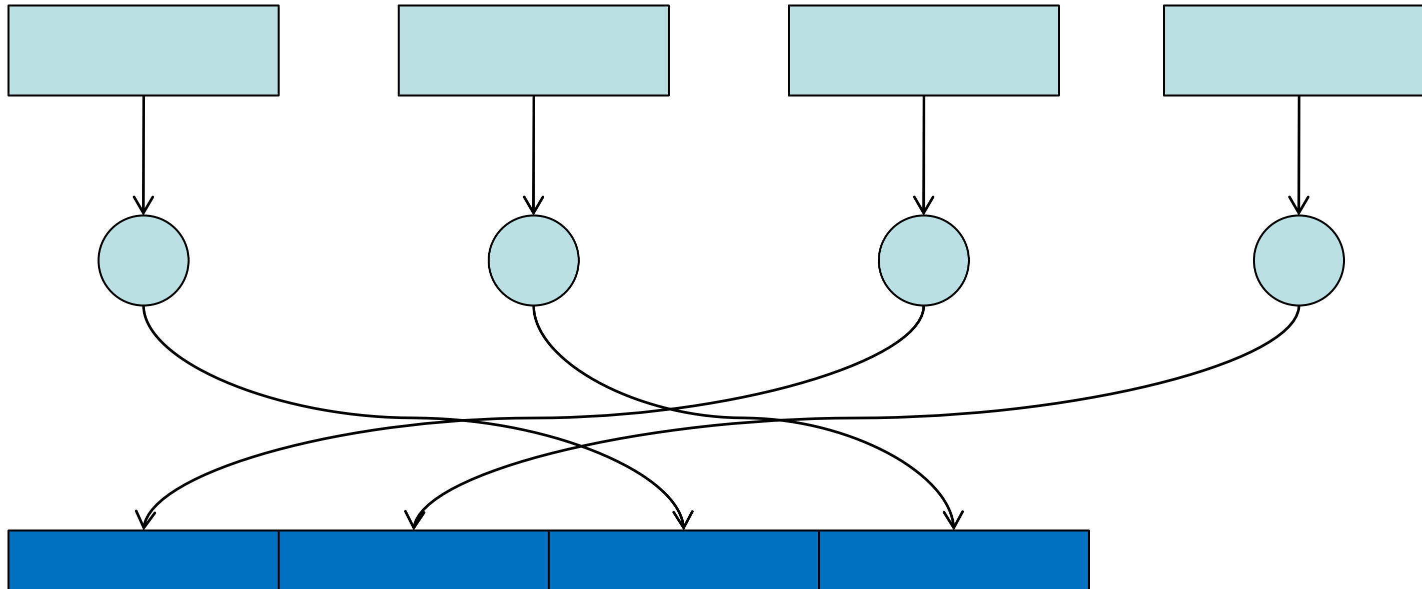
3. Otwórz plik używając `MPI_File_open` z opcjami `MPI_MODE_RDWR | MPI_MODE_CREATE`
4. Utwórz pętle od 0 do 10. Wewnątrz oblicz `offset` oraz `buf` i wypisz do pliku za pomocą `MPI_File_write_at`
5. Zamknij pętlę. Zamknij plik `MPI_File_close`

Zapis do pliku

- Użycie `MPI_File_write` lub `MPI_File_write_at`
- Użycie `MPI_MODE_WRONLY` lub `MPI_MODE_RDWR` jako flag dla `MPI_File_open`
- Jeśli plik nie istnieje to należy dodać flagę `MPI_MODE_CREATE` do `MPI_File_open`
- Możemy podawać wiele flag używając '|' w C, lub '+" w Fortranie
- Inne tryby:
  - MPI_MODE_RDONLY – tylko odczyt
  - MPI_MODE_RDWR – odczyt i zapis
  - MPI_MODE_WRONLY – tylko zapis
  - MPI_MODE_CREATE – utwórz jeśli plik nie istnieje
  - MPI_MODE_EXCL – zwróć błąd jeśli wykonana została próba utworzenia pliku, który istnieje
  - MPI_MODE_DELETE_ON_CLOSE – usuń plik przy zamknięciu
  - MPI_MODE_UNIQUE_OPEN – wymuś pojedynczy dostęp do pliku
  - MPI_MODE_APPEND – ustaw wszystkie wskaźniki na koniec plików

Użycie widoków plików

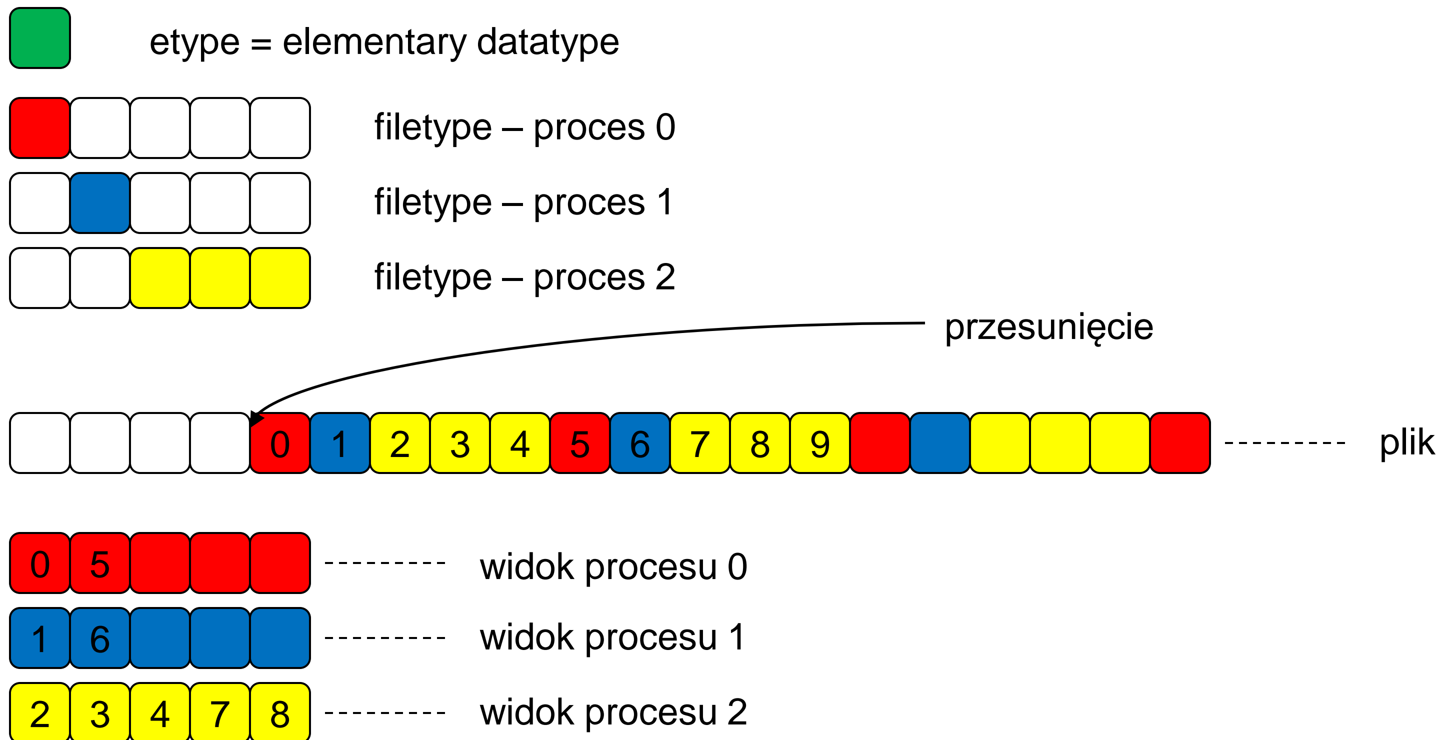
- Procesy zapisują do wspólnego pliku



- `MPI_File_set_view` przypisuje regiony plików dla różnych procesów

Widoki plików

- Tworzone przed trójkę (*przesunięcie*, *etype*, and *filetype*) przekazywanych do **`MPI_File_set_view`**
- *przesunięcie* = ilość bajtów do opuszczenia na początku pliku
- *etype* = podstawowa jednostka pliku – typ elementarny (np. typ MPI)
- *filetype* = określa, która część pliku jest widoczna dla procesu

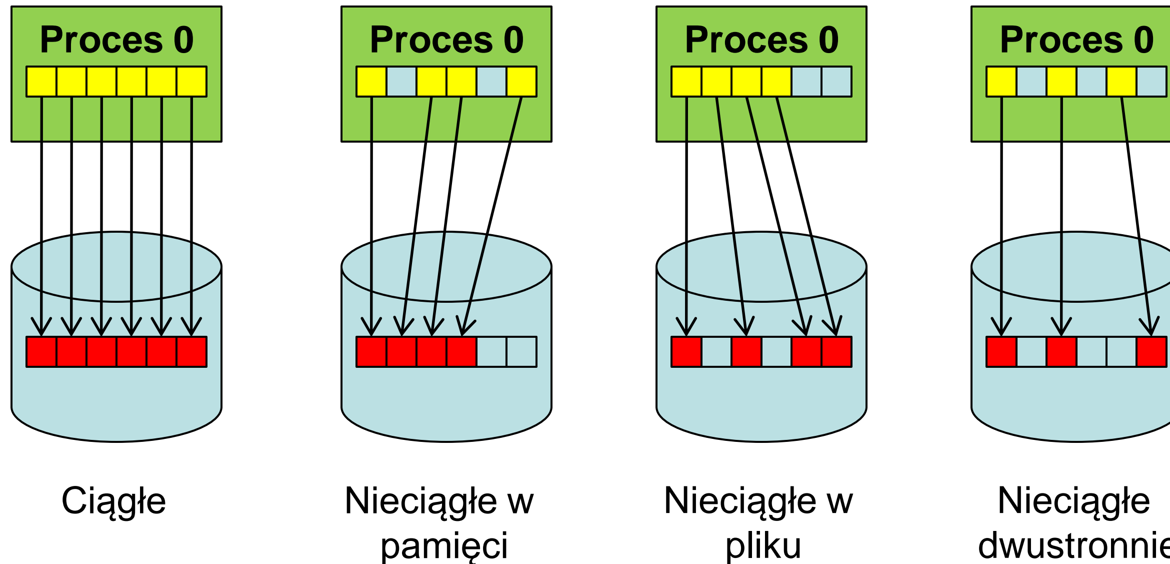


Przykład: widoki plików

- Opisuje część pliku widzialną z konkretnego procesu MPI.
- Argumenty `MPI_File_set_view`: `MPI_File file, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info`

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;

MPI_File_open(MPI_COMM_WORLD, "testfile",
     MPI_MODE_CREATE|MPI_MODE_WRONLY,MPI_INFO_NULL, &thefile);

MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
     MPI_INT, MPI_INT, "native",MPI_INFO_NULL);

MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,MPI_STATUS_IGNORE);

MPI_File_close(&thefile);
```

Nieciągłe I/O



- **Ciągłe I/O** przesuwanie danych z jednego bloku pamięci do jednego bloku w pliku

- **Nieciągłe I/O** ma 3 formy:nieciągłość w pamięci, nieciągłość w pliku, obydwie

Przykład: 2D lub 3D dane blokowe

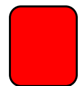
Przykład: nieciągłe I/O

Tablica 2D wczytywana przez 4 procesy

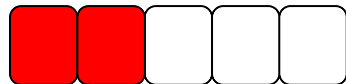
Proces 0	Proces 1
Proces 2	Proces 3

Wspólny plik z zapisem w stylu wierszowym

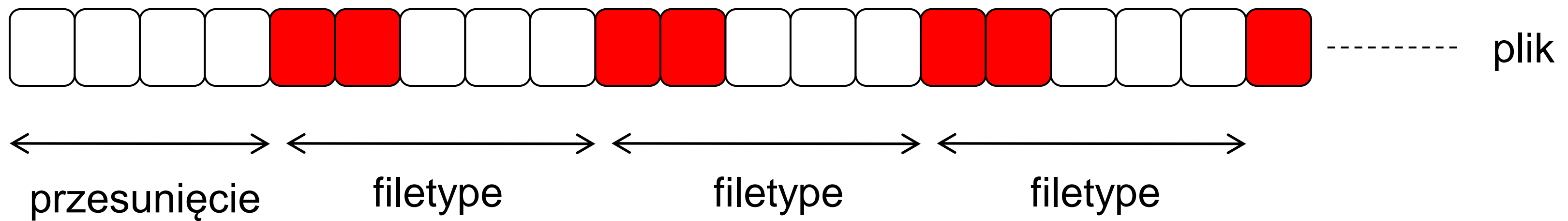
Przykład: nieciągłe I/O



etype = MPI_INT



filetype = 2 x MPI_INT oraz 3 x przerwa wielkości MPI_INT



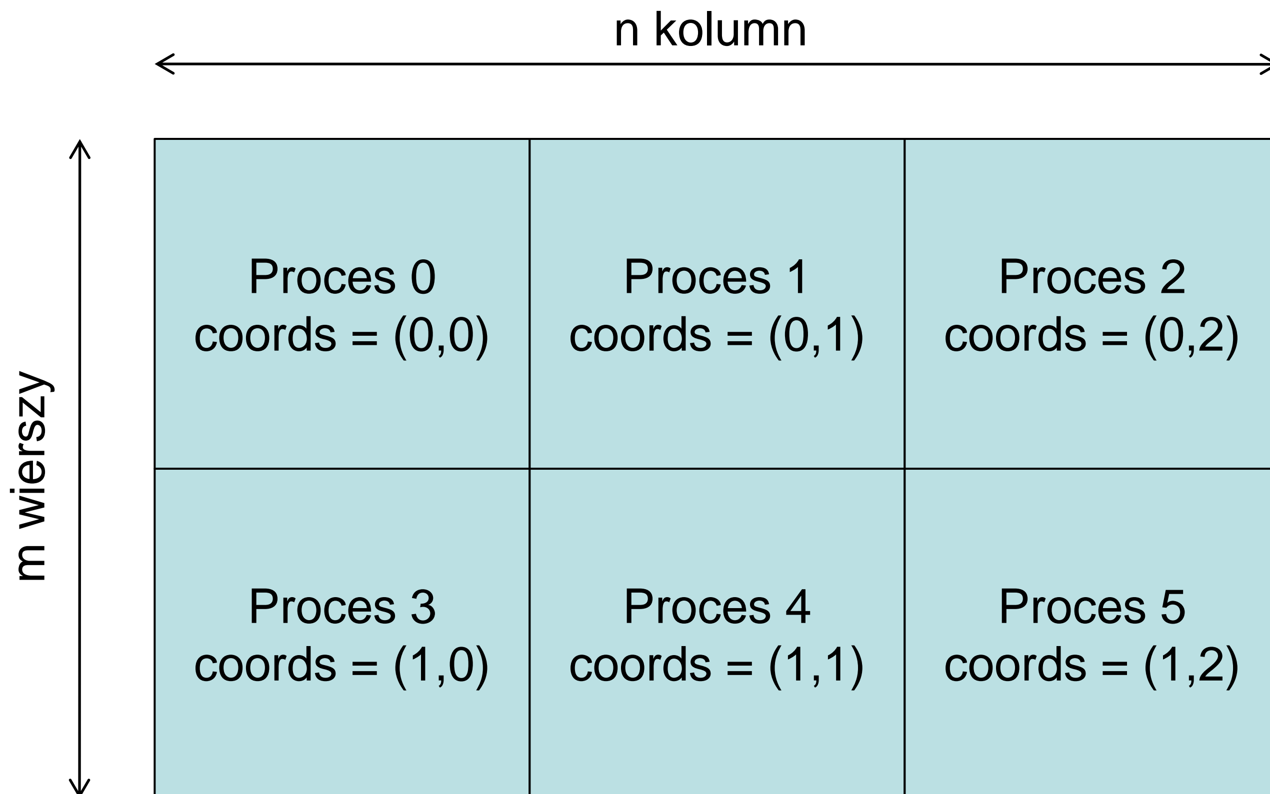
Przykład: widok plików

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);

lb = 0; extent = 6 * sizeof(int);

MPI_Type_create_resized(contig, lb, extent, &filetype);

MPI_Type_commit(&filetype);

disp = 5 * sizeof(int); etype = MPI_INT;


MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);

MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Zbiorowe I/O

- `MPI_File_read_all`, `MPI_File_read_at_all`, ...
- `_all` oznacza, że wszystkie funkcje z komunikatora wywołają funkcję `MPI_File_open`
- każdy proces specyfikuje tylko i wyłącznie swój własny dostęp do pliku – argumenty są dokładnie takie same jak w komunikacji nie-zbiorowej

Wczytywanie tablic z pliku



nproc(1) = 2, nproc(2) = 3

Użycie typu Darray

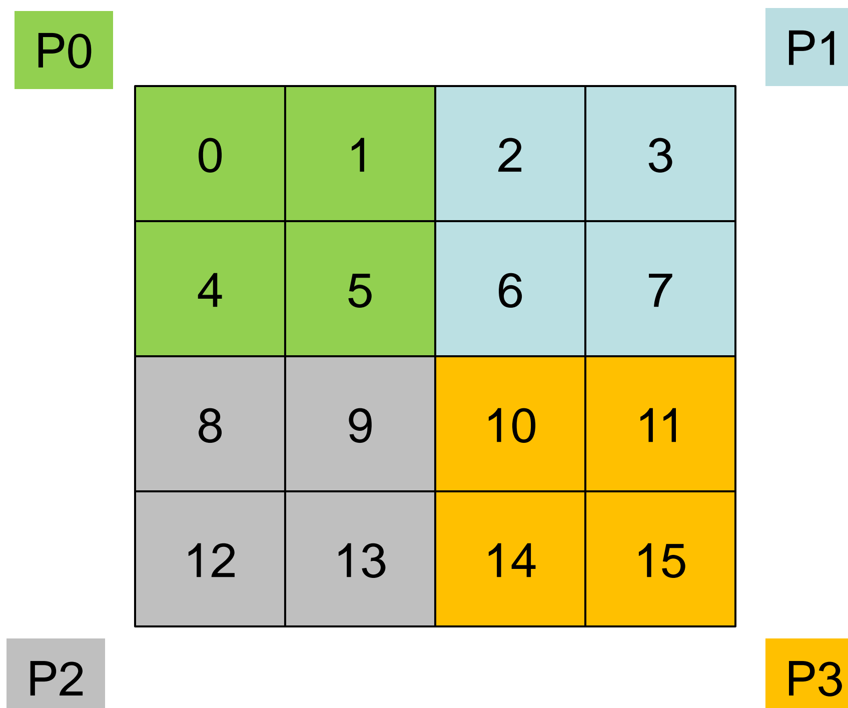
```
int gsizes[2], distribs[2], dargs[2], psizes[2];

gsizes[0] = m;     // ilość wierszy w globalnej tablicy
gsizes[1] = n;     // ilość kolumn w globalnej tablicy

distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;

dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;

psizes[0] = 2; // ilość procesów w pionie
psizes[1] = 3; // ilość procesów w poziomie

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs,
               psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",MPI_MODE_CREATE|MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",MPI_INFO_NULL);

local_array_size = num_local_rows * num_local_cols;
MPI_File_write_all(fh, local_array, local_array_size,MPI_FLOAT, &status);

MPI_File_close(&fh);
```

Uwagi dot. typu darray

- Typ **darray** zakłada bardzo specyficzny rozkład danych
- Jeśli rozmiar nie dzieli się przez liczbę procesorów, **darray** używa zaokrągleń (20/6=4)
- Wszędzie tam, gdzie nie da się użyć **darray** używamy **subarray**

Hands-on: darray

- Napisz program, który uruchamiany na 4 procesach MPI wczytuje równolegle następującą tablicę zawartą w pliku binarnym:



- Skorzystaj ze schematu przedstawionego na slajdzie 78

Typ Subarray

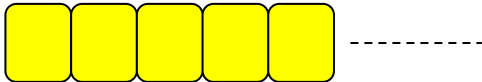
```
gsizes[0] = m;  // ilość wierszy w globalnej tablicy
gsizes[1] = n;  // ilość kolumn w globalnej tablicy

psizes[0] = 2; // ilość procesów w pionie
psizes[1] = 3; // ilość procesów w poziomie

lsizes[0] = m/psizes[0]; // ilość wierszy w lokalnej tablicy
lsizes[1] = n/psizes[1]; // ilość kolumn w lokalnej tablicy

dims[0] = 2; dims[1] = 3;
periods[0] = periods[1] = 1;

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);

MPI_Comm_rank(comm, &rank);

MPI_Cart_coords(comm, rank, 2, coords);
```

Typ Subarray

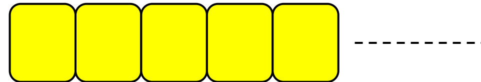
```
// globalne indeksy pierwszego elementu lokalnej tablicy
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                         MPI_ORDER_C, MPI_FLOAT, &filetype);

MPI_Type_commit(&filetype);


MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
              MPI_INFO_NULL);

local_array_size = lsizes[0] * lsizes[1];

MPI_File_write_all(fh, local_array, local_array_size,
              MPI_FLOAT, &status);
```

Nieregularne tablice

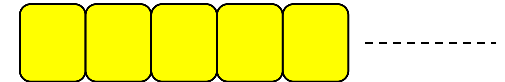
Tablica procesu 0



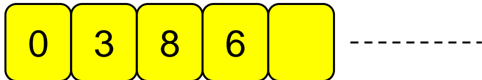
Tablica procesu 1



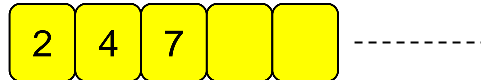
Tablica procesu 2



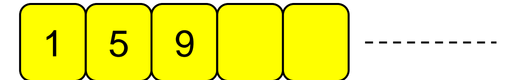
Mapa procesu 0



Mapa procesu 1



Mapa procesu 2



Mapa określa położenie każdego elementu tablicy lokalnej w tablicy globalnej

Nieregularne tablice

```
integer (kind=MPI_OFFSET_KIND) disp

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
                   MPI_MODE_CREATE + MPI_MODE_RDWR, &
                   MPI_INFO_NULL, fh, ierr)

call MPI_TYPE_CREATE_INDEXED_BLOCK(bufsize, 1, map, &
                    MPI_DOUBLE_PRECISION, filetype, ierr)

call MPI_TYPE_COMMIT(filetype, ierr)

disp = 0

call MPI_FILE_SET_VIEW(fh, disp, MPI_DOUBLE_PRECISION, &
                 filetype, 'native', MPI_INFO_NULL, ierr)

call MPI_FILE_WRITE_ALL(fh, buf, bufsize, &
                      MPI_DOUBLE_PRECISION, status, ierr)

call MPI_FILE_CLOSE(fh, ierr)
```

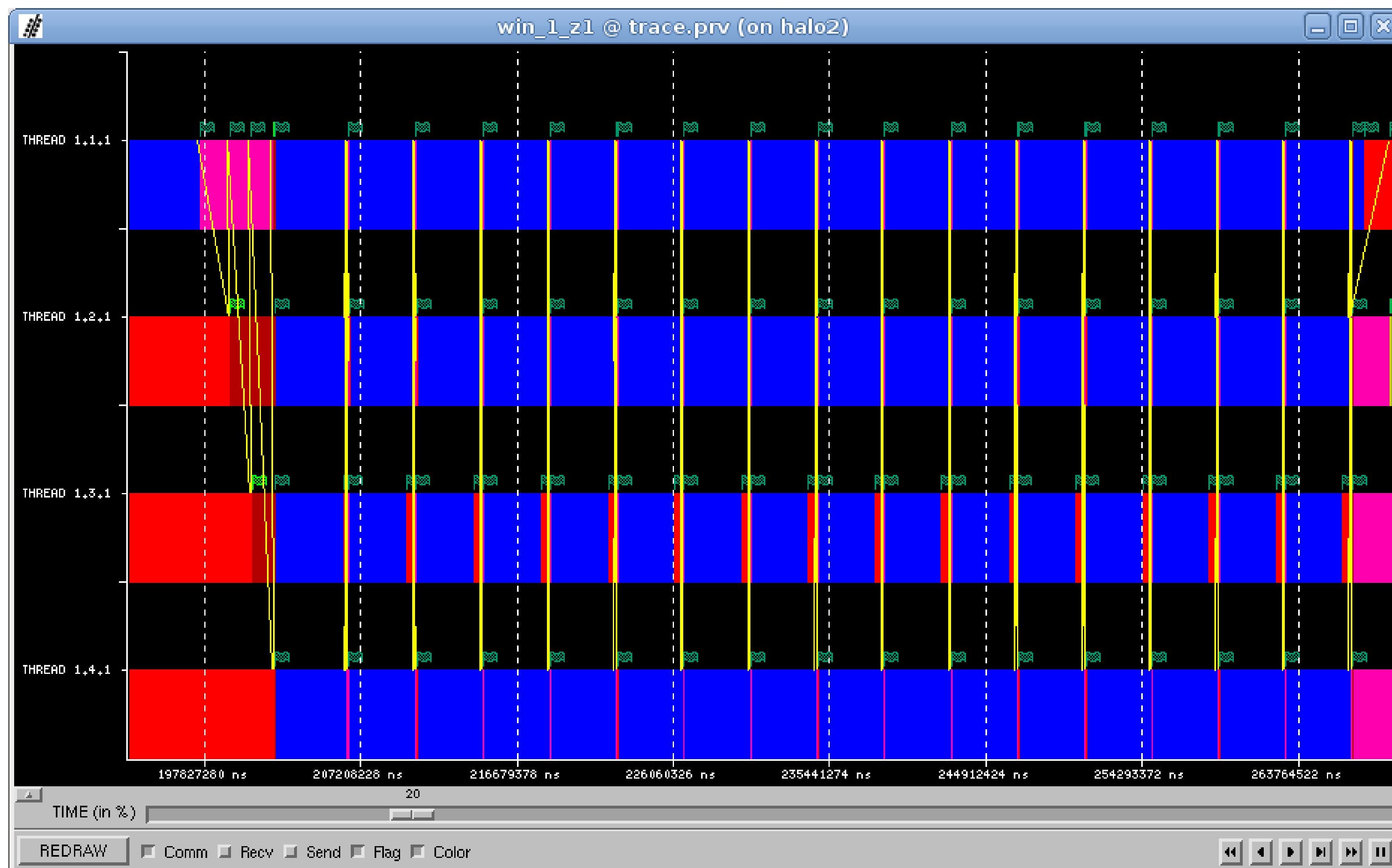
Nieblokujące I/O

```
MPI_Request request;
MPI_Status status;

MPI_File_iwrite_at(fh, offset, buf, count, datatype,
                   &request);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_Wait(&request, &status);
```

Co to jest Paraver?

- Oprogramowanie służące do analizy tzw. trace-ów równoległych
- Trace to plik zawierający informację dotyczącą przekroju wykonania aplikacji pod względem wybranych jej funkcjonalności
- Trace równoległy to zwykle plik, w którym znajdują się informację o wywołaniach np. biblioteki MPI lub dyrektyw OpenMP (które funkcje/dyrektywy i w jakiej postaci zostały wykonane w danej chwili czasowej)
- Paraver wspiera: MPI, OpenMP, MPI+OpenMP, Java, badanie liczników sprzętowych
- Producent: Barcelona Supercomputing Center
- Dostępność: halo2, blader

Paraver - przykład



Hands-on: Paraver

Wykonaj poniższe czynności w celu utworzenia i pliku Paraver do analizy.

W trakcie analizy programu odpowiedz na pytanie:

kto z kim komunikuje się w tym programie?

```
ssh -Y halo2
mkdir paraver_test
cd paraver_test
cp /opt/paraver/examples/* .
module load paraver
module load openmpi_gnu64
mpicc -o smoothing smoothing.c
qsub mpitrace_job.pbs
mpitrace_merge.tcsh
paraver trace.prv
```

Dziękujemy za uwagę ☺