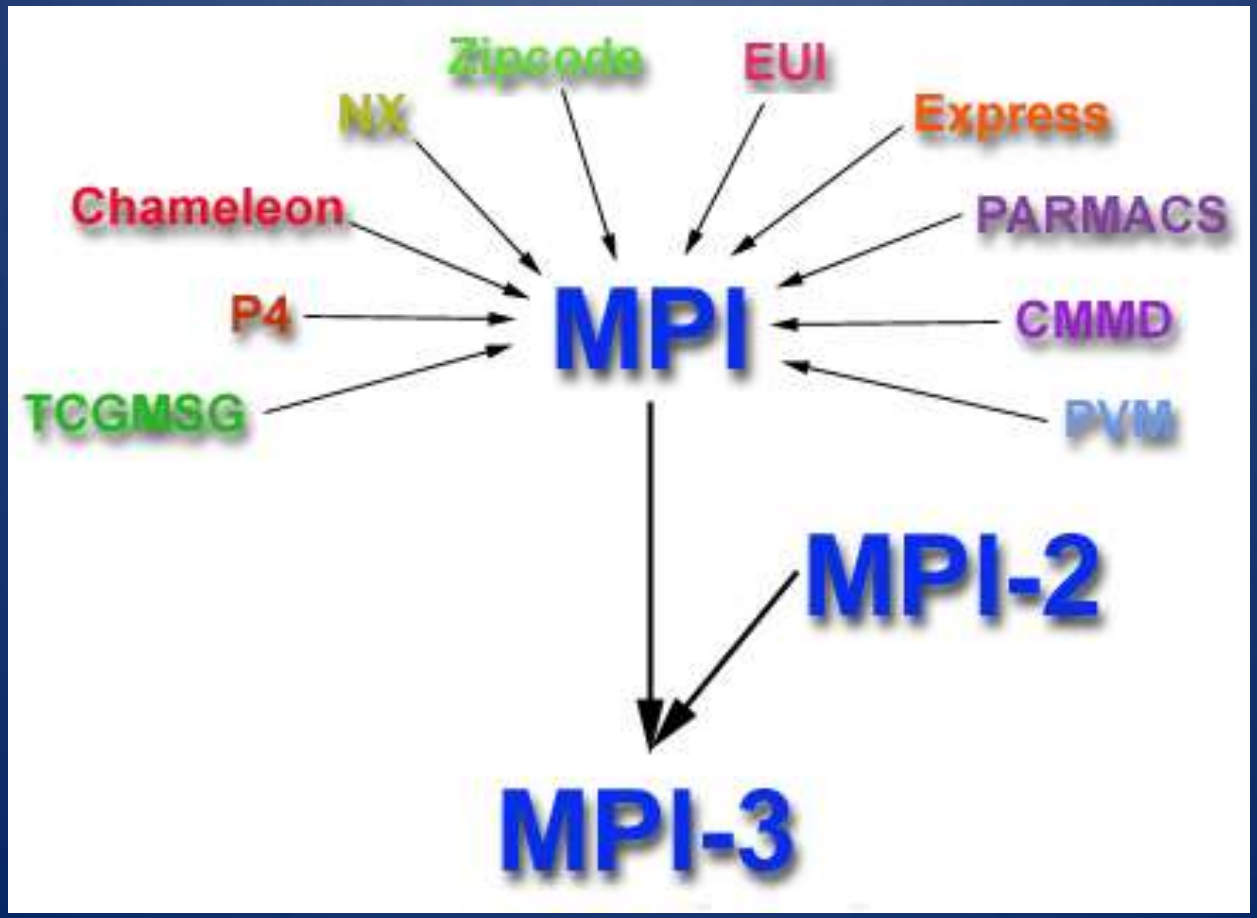
DEFINICJA STANDARDU MPI

LESŁAW SIENIAWSKI © 2017

SPECYFIKACJA STANDARDU MPI

- Opracowany przez MPI Forum
  (40+ organizacji, w tym dostawcy, naukowcy, twórcy bibliotek oprogramowania, użytkownicy)
- Nie jest oficjalnym standardem ISO ani IEEE
  praktyczny, przenośny, efektywny, elastyczny, uznany jako standard *de facto* do tworzenia aplikacji obliczeniowych z przekazywaniem komunikatów
- Specyfikacja interfejsu dla C/C++ i Fortranu

GENEZA MPI



[źródło: https://computing.llnl.gov/tutorials/mpi/]

SKRÓT HISTORII MPI

- Lata 1980-początek lat 1990 – różne wzajemnie niespójne koncepcje narzędzi
- Kwiecień 1992 – spotkanie grupy roboczej *Workshop for Message Passing in Distributed Memory Environment*
- Maj 1994 – oficjalne ogłoszenie MPI-1.0 (aktualizacje: MPI-1.1: 06.1995, MPI-1.2: 07.1997, MPI-1.3: 05.2008)
- MPI-2 (MPI-2.1: 09.2008, MPI-2.2: 09.200) – traktowany niekiedy jako dodatek do MPI-1
- MPI-3 (MPI-3.0: 09.2012, MPI-3.1:: 06.2015)
- MPI-4 (prace w toku)

(wg https://computing.llnl.gov/tutorials/mpi/)

Dokumentacja różnych wersji MPI: http://www.mpi-forum.org/docs/

WŁAŚCIWOŚCI MPI

- Standaryzacja – dostępność praktycznie na wszystkich platformach
- Przenośność – brak konieczności modyfikacji kodu źródłowego po przeniesieniu na inną platformę zgodną ze standardem MPI
- Implementacje dla konkretnych platform mogą wykorzystywać ich specyficzne cechy dla zwiększania wydajności
- Funkcjonalność
  - MPI-1 = 128 procedur
  - MPI-2 = +152 procedury
  - MPI-3 > 430 procedur (większość dostępnych w MPI-1 i MPI-2)
- Dostępność – różnorodność implementacji komercyjnych i *public domain*

MPI – MODEL PROGRAMOWANIA

- Wsparcie: praktycznie dla każdego modelu programowania równoległego z rozproszoną pamięcią tj. MIMD (SPMD)
- Platformy sprzętowe:
  - Z pamięcią rozproszoną (cel wyjściowy)
  - Z pamięć współdzieloną (w tym SMP, NUMA)
  - Hybrydowe (masowe maszyny równoległe, klastry SMP, klastry stacji roboczych, sieci heterogeniczne)
- Równoległość jawna, obsługiwana przez programistę
- Stała liczba podzadań (*task*) *wspólnie* realizujących równoległe zadanie (*job*). W MPI-1 brak możliwości dynamicznego tworzenia podzadań.

MPI – MODEL PROGRAMOWANIA (2)

- Wiele programów można zbudować korzystając tylko z 6 funkcji:
  1. MPI_Init,
  2. MPI_Comm_size,
  3. MPI_Comm_rank,
  4. MPI_Send / MPI_Bcast,
  5. MPI_Recv / MPI_Reduce,
  6. MPI_Finalize
- Dodatkowe funkcje → zwiększenie elastyczności

MPI – SKŁADNIKI API

- Biblioteki funkcji i plików nagłówkowych
  - Języka C
  - Języka C++
  - Języka Fortran 77
  - Języka Fortran 90
- Środowisko uruchomieniowe (*runtime*) analogicznie do apletów języka Java

MPI – PODSTAWOWE OBIEKTY

- Komunikator
  - Zbiór procesów które mogą się ze sobą komunikować (kontekst komunikacji)
  - Predefiniowany komunikator `MPI_COMM_WORLD` obejmujący wszystkie procesy MPI
  - Programista może definiować nowe komunikatory
- Grupa – kolekcja komunikujących się procesów
- Identyfikator procesu w komunikatorze
  - tzw. *rank* – nieujemna liczba całkowita (0, 1, …) nadana przez system w chwili inicjalizacji procesu,
  - używany do wskazywania nadawców i odbiorców komunikatów i do adresowania wykonawców czynności w zadaniu

KOMUNIKACJA PUNKT-PUNKT

- Zasady
  - Przekazywanie komunikatów pomiędzy dwoma (!) podzadaniami MPI (nadawca – odbiorca)
  - Różne typy procedur:
    - Synchroniczne wysyłanie
    - Blokujące wysyłanie / blokujące odbieranie
    - Nieblokujące wysyłanie / nieblokujące odbieranie
    - Buforowane wysyłanie
    - Kombinowane wysyłanie / odbieranie
  - Dozwolone łączenie różnych typów procedur wysyłania i odbierania
  - Dodatkowe procedury obsługi (np. czekanie na komunikat, badanie czy komunikat już nadszedł)

KOMUNIKACJA PUNKT-PUNKT (2)

- Buforowanie

  konieczność obsługi komunikacji przy braku synchronizacji podzadań, np.

  - Nadejścia komunikatu przed uzyskaniem gotowości odbiorcy do przyjęcia go
  - Równoczesnego nadejścia wielu komunikatów do odbiorcy zdolnego tylko do pojedynczego ich przyjmowania

  →Konieczność ustalenia, co zrobić z komunikatami, które nie mogą być odebrane w chwili nadejścia - zadanie rozwiązywane poprzez:

  →bufor systemowy (składnik implementacji MPI) lub

  →bufor aplikacyjny (składnik standardu MPI)

KOMUNIKACJA PUNKT-PUNKT . BUFOR SYSTEMOWY

[na podstawie: https://computing.llnl.gov/tutorials/mpi/]

KOMUNIKACJA PUNKT-PUNKT. BUFOR SYSTEMOWY (2)

- Cechy bufora systemowego
  - Nieprzezroczystość dla programisty
  - Niewystarczające udokumentowanie
  - Ograniczona pojemność
  - Niekiedy niedostępność tam, gdzie wymagany
  - Zdolność poprawiania wydajności programu przez umożliwienie komunikacji asynchronicznej

KOMUNIKACJA PUNKT-PUNKT. BUFOR APLIKACYJNY

- Bufor aplikacyjny = przestrzeń adresowa zarządzana przez programistę (zmienne programowe)
- Bufor wysyłania zapewniony przez MPI

TRYBY BLOKOWANIA/NIEBLOKOWANIA

Z blokowaniem

- Procedura wysyłająca (send) kończy pracę, kiedy jest gwarancja, że modyfikacja danych w buforze nadawczym nie zmieni danych przeznaczonych dla odbiorcy; w tym celu dane nie muszą być już odebrane przez adresata
  - Synchroniczne wysyłanie z blokowaniem – zastosowanie tzw. *handshakingu*
  - Asynchroniczne wysyłanie z blokowaniem – wykorzystanie bufora systemowego
- Procedura blokowanego odbioru (recv) kończy pracę, gdy dane nadeszły i są gotowe do wykorzystania przez program

TRYBY BLOKOWANIA/NIEBLOKOWANIA (2)

Bez blokowania

- Procedury nieblokującego wysyłania i nieblokującego odbioru kończą pracę bez czekania na efekty komunikacji,
- Moment wystąpienia zdarzenia dotyczącego realizowanej komunikacji nie jest przewidywalny
- Modyfikacja bufora aplikacyjnego przed upewnieniem się, że nieblokująca operacja została ukończona jest niebezpieczna; zalecane jest wykorzystanie odpowiedniej procedury czekania
- Komunikacja nieblokująca służy głównie do nakładania w czasie obliczeń i komunikacji oraz zwiększania wydajności

TRYBY BLOKOWANIA/NIEBLOKOWANIA (3)

- Większość procedur komunikacyjnych punkt-punkt działa w obydwu trybach

PORZĄDEK KOMUNIKATÓW

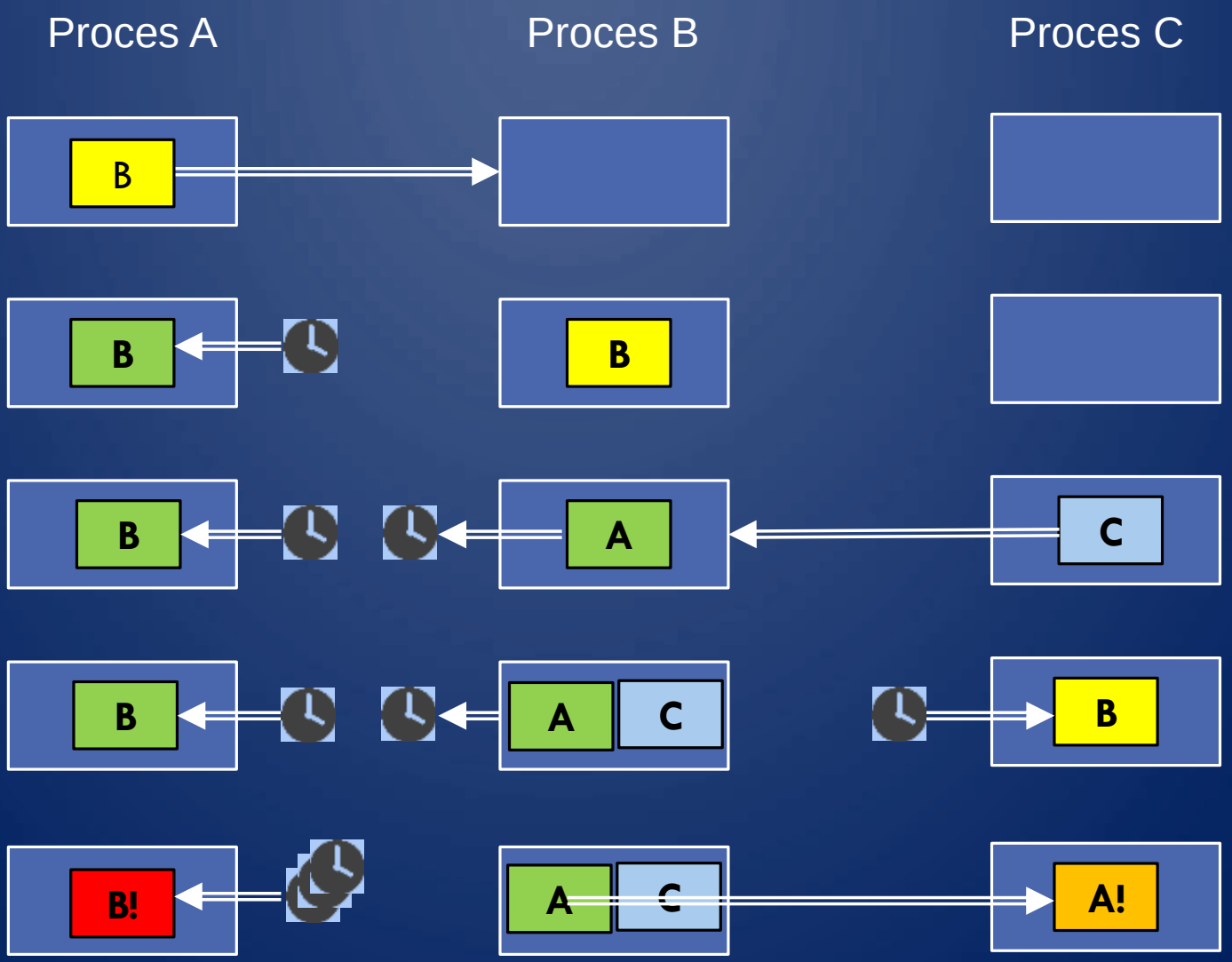
- Porządek
  - Gwarancja braku wzajemnego wyprzedzania się przez komunikaty
  - Reguły kolejności nie obowiązują przy wykorzystywaniu do komunikacji mechanizmu wielowątkowości

SPRAWIEDLIWOŚĆ

Sprawiedliwość

- Brak gwarancji sprawiedliwości,
- Programista musi zabezpieczyć program przed tzw. *zagłodzeniem operacji*, np.
  - podzadanie A wysyła komunikat do podzadania B,
  - podzadanie B ma przygotowaną (ale nie wysłaną) odpowiedź dla A,
  - w tym czasie podzadanie C wysyła komunikat do B i odbiera stąd odpowiedź przeznaczoną dla A
  - dlatego A nigdy nie dostaje odpowiedzi od B

SPRAWIEDLIWOŚĆ (2)



MPI – STRUKTURA PROGRAMU

```
#include <mpi.h>
inne dyrektywy, deklaracje i prototypy
```

Początek programu

```
Kod sekwencyjny
```

Inicjalizacja środowiska MPI = początek kodu równoległego

```
Obliczenia i komunikacja
```

Zakończenie pracy środ. MPI = koniec kodu równoległego

```
Kod sekwencyjny
```

Koniec programu

MPI – FORMAT WYWOŁANIA FUNKCJI

Postać ogólna:

```
rc = MPI_Xxxx(parametr,…);
```

`rc` – kod powrotu (ang. *return code*)

Po poprawnym wykonaniu procedury

`rc = MPI_SUCCESS` (predefiniowana stała)

UWAGA: Bywa, że w programach nie pobiera się i nie bada kodu powrotu.

MPI – PROCEDURY WYMIANY KOMUNIKATÓW

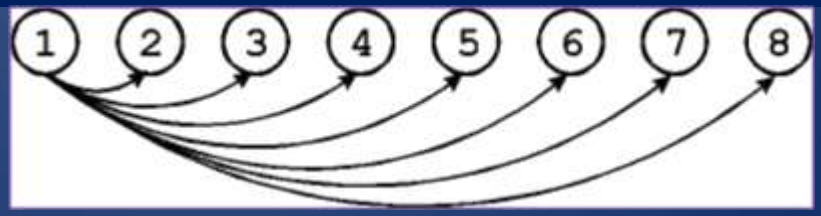
- Przeznaczenie
  - Wymiana danych między procesami
  - Wysyłanie komunikatów kontrolnych
  - Synchronizacja procesów
- Właściwości
  - Niezależność od platformy
    - kolejność przesyłania bajtów dla standardowych typów danych
    - Dodatkowe funkcje dla typów niestandardowych
  - Adresowanie komunikatów do konkretnych procesów i/lub określonych grup odbiorców
  - Możliwość etykietowania komunikatów (oznaczanie rodzajów) dla selektywnej obsługi kolejki

MPI – PROCEDURY WYMIANY KOMUNIKATÓW (2)

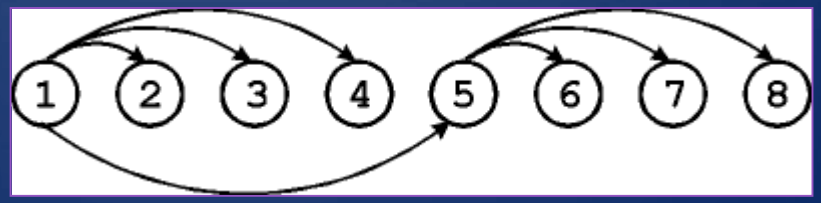
- Rozsyłanie komunikatów
  - Ukrywanie szczegółów realizacyjnych
  - Automatyzacja doboru schematu przepływu danych

Przykład: Badamy czas potrzebny procesowi #1 do wysłania tego samego komunikatu do procesów #2 – #8.

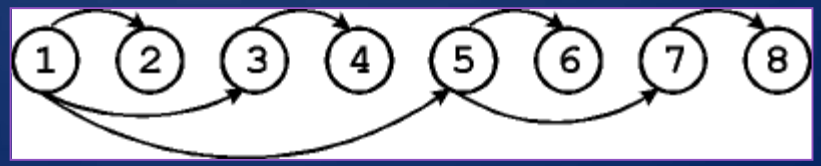
Wariant podstawowy
(7 komunikatów)



Wariant ulepszony
(4 komunikaty)



Wariant optymalny
(3 komunikaty)



[rysunki: http://www-users.mat.uni.torun.pl/~bala/sem_mgr_2000/mpi.html]

MPI – PROCEDURY ZARZĄDZANIA ŚRODOWISKIEM WYKONAWCZYM

Przeznaczenie

- Inicjalizacja i kończenie działania środowiska MPI
- Badanie właściwości środowiska
- Sprawdzanie tożsamości
- Kontrola poprawności przesyłania komunikatów
- Inne

MPI – PROCEDURY ZARZĄDZANIA ŚRODOWISKIEM WYKONAWCZYM (2)

```
int MPI_Init(int *argc,char *argv[])
```

jednorazowa inicjalizacja środowiska MPI, w tym utworzenie domyślnego komunikatora `MPI_COMM_WORLD;` konieczne jest wywołanie tej procedury przed pierwszym wywołaniem jakiejkolwiek innej procedury MPI

```
int MPI_Comm_size(MPI_Comm comm,int *size)
```

ustalenie liczby procesów w grupie związanej z komunikatorem

```
int MPI_Comm_rank(MPI_Comm comm,int *rank)
```

określenie identyfikatora wywołującego procesu w obrębie komunikatora; początkowo każdy proces ma swój ID w obrębie komunikatora globalnego `MPI_COMM_WORLD` (tzw. *identyfikator podzadania*). W razie związania się z innymi komunikatorami, proces posiada w każdym z nich odrębny unikalny ID

MPI – PROCEDURY ZARZĄDZANIA ŚRODOWISKIEM WYKONAWCZYM (3)

```
int MPI_Abort(MPI_Comm comm,int error_code)
```

zakończenie wszystkich procesów MPI związanych ze wskazanym komunikatorem (w większości implementacji zamyka wszystkie procesy, bez względu na podany komunikator)

```
int MPI_Get_Processor_name(char *name,
    int *name_length)
```

zwrócenie nazwy wykonawcy i jej długości; bufor nazwy musi pomieścić co najmniej `MPI_MAX_PROCESSOR_NAME` znaków; rezultat zależny od implementacji (np. nazwa hosta)

MPI – PROCEDURY ZARZĄDZANIA ŚRODOWISKIEM WYKONAWCZYM (4)

`int MPI_Initialized(int *flag)`

zwrócenie **`true`**, jeśli wcześniej wywołano **`MPI_Init`** lub **`false`**, w przeciwnym przypadku; ułatwia to konstruowanie złożonych struktur, bo każdy proces musi wywołać **`MPI_Init`** dokładnie jeden raz

`double MPI_Wtime (void)`

zwrócenie czasu astronomicznego w [s] (typ **`double`**)

`double MPI_Wtick (void)`

zwrócenie rozdzielczości zegara **`MPI_Wtime`** w [s]

`int MPI_Finalize (void)`

zakończenie działania środowiska wykonawczego MPI; ostatnie wywołanie procedury MPI w każdym programie korzystającym z MPI

MPI – PODSTAWOWE PROCEDURY KOMUNIKACJI

```
int MPI_Send(void *buf, int count,
MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

Blokujące wysłanie komunikatu typu **datatype** zawartego w zmiennej ***buf** i oznaczonego znacznikiem (etykietą) **tag** do procesu o numerze **dest.**

Typ komunikatu – predefiniowany (**MPI_INT, MPI_FLOAT, MPI_CHAR**, itp.) lub zdefiniowany przez użytkownika

Znacznik (etykieta) – liczba z zakresu **[0..MPI_TAG_UB]** – dodatkowe oznaczenie typu komunikatu wykorzystywane przez **MPI_Recv**

MPI – PODSTAWOWE PROCEDURY KOMUNIKACJI (2)

```
int MPI_Recv(void *buf, int count,
MPI_Datatype datatype, int srce, int tag,
MPI_Comm comm, MPI_Status *status)
```

blokujące odbieranie z kolejki komunikatora **comm** pierwszego komunikatu typu **datatype** od procesu **srce**, posiadającego znacznik **tag** oraz umieszczenie komunikatu w **buf** i kodu wyniku operacji w **status**

srce=MPI_ANY_SOURCE → odczytanie 1-go komunikatu od dowolnego nadawcy

tag=MPI_ANY_TAG → ignorowanie znacznika

MPI – PODSTAWOWE PROCEDURY KOMUNIKACJI (3)

Bufor `status` typu `MPI_Status` jest tablicą struktur złożonych z 3 pól typu int:

- `MPI_SOURCE`,
- `MPI_TAG`,
- `MPI_STATUS`

Bufor musi być uprzednio zadeklarowany. Zawiera informacje o źródle i typie każdego odebranego komunikatu

MPI – PODSTAWOWE PROCEDURY KOMUNIKACJI (4)

```
int MPI_Get_count(MPI_Status *status,
MPI_Datatype datatype,
int *count)
```

Ustalenie liczby odebranych komunikatów typu **datatype** na podstawie zawartości zmiennej **status** i umieszczenie wyniku w zmiennej **count**

KOMPILACJA PROGRAMÓW

Zamiast zwykłego wywołania

```
gcc plik.c -o plik opcja…
```

stosujemy

```
mpicc plik.c -o plik opcja…
```

Wymagania:

Ścieżka do katalogu **bin** z plikami wykonywalnymi MPI umieszczona w zmiennej środowiska (powłoki) **PATH**

Nazwa **mpicc** stanowi opakowanie (ang. *wrapper*) zasłaniające nazwy plików kompilatorów stosowanych w różnych środowiskach MPI.

URUCHAMIANIE PROGRAMÓW

Zamiast polecenia

```
./plik   parametr_wywołania...
```

jak dla zwykłego programu w jęz. C, stosujemy

```
mpirun -n liczba_wykonawców \
    --mca btl tcp,self plik \
    parametr_wywołania...
```

Nazwa **mpirun** stanowi opakowanie (ang. *wrapper*) zasłaniające nazwy plików programów wykonawczych stosowanych w różnych środowiskach MPI.

URUCHAMIANIE PROGRAMÓW (2)

- Wymagania:
  - Użytkownik posiada konto na wszystkich komputerach wykorzystywanych do wykonywania programu
  - We wszystkich komputerach, w katalogu prywatnym użytkownika znajduje się kopia pliku programu w postaci wykonywalnej (np. dzięki współdzieleniu jednego katalogu w sieci przez NFS, przez kopiowanie, lub tp.)

URUCHAMIANIE PROGRAMÓW (3)

- Korzyści wynikające ze sposobu uruchamiania
  - Uniezależnienie programu od sposobu jego wykonywania;
  - Bez rekompilacji można użyć tego samego programu na klastrze komputerów i na pojedynczej maszynie SMP (o tej samej architekturze)

Przykład - program MPI #1 [wg: https://computing.llnl.gov/tutorials/mpi/]

```
#include <mpi.h>
#include <stdio.h>
int main(int argc,char *argv[])
{ int numtasks, rank, rc;
  rc = MPI_Init(&argc,&argv);
  if (rc != MPI_SUCCESS) {
    printf ("Error starting MPI program.
  Terminating.\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
  }
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf ("Number of tasks= %d My rank= %d\n",
       numtasks,rank);
  /******* do some work *******/
  MPI_Finalize();
}
```

Badanie wyniku inicjalizacji

Przykład - program MPI #1 – kompilacja i uruchomienie

```
[root@p205 openMPI]# mpicc mpi1.c -o mpi1
[root@p205 openMPI]# mpirun -n 1 --mca btl tcp,self
  mpi1
Number of tasks= 1 My rank= 0
[root@p205 openMPI]# mpirun -n 2 --mca btl tcp,self
  mpi1
Number of tasks= 2 My rank= 0
Number of tasks= 2 My rank= 1
[root@p205 openMPI]#
```

Przykład - program MPI #2 – dane środowiska wykonawczego

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
char nazwa_proc[256];
int  nazwa_proc_len, id;
MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    printf("--------------------\nTu wykonawca nr
  %d\n", id);
    printf("Wartosc stalej MPI_MAX_PROCESSOR_NAME =
  %d\n", MPI_MAX_PROCESSOR_NAME);
    MPI_Get_processor_name(nazwa_proc,
  &nazwa_proc_len);
    printf("Nazwa procesora = %s, dlugosc nazwy =
  %d\n", nazwa_proc, nazwa_proc_len);
    printf("Czas aktualny = %f [s], rozdzielczosc
  zegara = %f [s]\n", MPI_Wtime(), \ MPI_Wtick());
MPI_Finalize();
    return 0;
}
```

Przykład - program MPI #2 – kompilacja i uruchomienie

```
[root@p205 openMPI]# mpicc mpi2.c -o mpi2
[root@p205 openMPI]# mpirun -n 2 --mca btl tcp,self pi2
--------------------
Tu wykonawca nr 0
Wartosc stalej MPI_MAX_PROCESSOR_NAME = 256
Nazwa procesora = p205, dlugosc nazwy = 4
--------------------
Tu wykonawca nr 1
Wartosc stalej MPI_MAX_PROCESSOR_NAME = 256
Czas aktualny = 1270674511.641517 [s], rozdzielczosc zegara =
  0.000001 [s]
Nazwa procesora = p205, dlugosc nazwy = 4
Czas aktualny = 1270674511.641643 [s], rozdzielczosc zegara =
  0.000001 [s]
```

Przykład - program MPI #3

[wg: http://www-users.mat.uni.torun.pl/~bala/sem_mgr_2000/mpi.html]

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv)
{ int my_rank;
  int p;
  int source;
  int dest;
  int tag=50;
  char message[100];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
```

(c.d. na następnym slajdzie)

(kontynuacja poprzedniego slajdu)

```
if (my_rank != 0) {
   sprintf(message, "Hello from process %d.",
 my_rank);
   dest = 0;
   MPI_Send(message, strlen(message)+1,
 MPI_CHAR, dest, tag, MPI_COMM_WORLD);
 } else
     for (source=1; source<p; source++)
     { MPI_Recv(message, 100, MPI_CHAR, source,
 tag, MPI_COMM_WORLD, &status);
       printf("%s\n", message);
     }
MPI_Finalize();
}
```

Przykład - program MPI #3 – kompilacja i uruchomienie

```
[root@p205 programy.c]# mpicc MPI-helloW.c -o MPI-helloW
[root@p205 programy.c]# mpirun -n 1 --mca btl tcp,self MPI-helloW
```

Pojedynczy wykonawca nie ma z kim wymieniać komunikatów

```
[root@p205 programy.c]# mpirun -n 2 --mca btl tcp,self MPI-helloW
Hello from process 1.
[root@p205 programy.c]# mpirun -n 4 --mca btl tcp,self MPI-helloW
Hello from process 1.
Hello from process 2.
Hello from process 3.
[root@p205 programy.c]#
```

Wszystkie zademonstrowane tu przykłady wykonano na maszynie o architekturze SMP (tj. wieloprocesorowej z pamięcią współdzieloną)

```
$ Press to shutdown
```