INTERFACE PROGRAMOWANIA OPENMPI

MPI czyli Message Passing Interface stanowi specyfikację pewnego standardu wysokopoziomowego protokołu komunikacyjnego i powiązanego z nim interface'u programowania, służącego programowaniu współbieżnemu. Reprezentuje on znacznie wyższy poziom abstrakcji niż standardowe UNIX System V IPC czy POSIX Threads.

Początki MPI sięgają lat osiemdziesiątych i stanowią wynik prac zespołu William'a "Bill" Gropp'a z University of Illinois. Podobne próby i prace podejmowano także i w innych ośrodkach, wymienić warto w szczególności California Institute of Technology oraz Ohio Supercomputer Center.

W roku 1992, przy okazji konferencji Supercomputing, udało się jej uczestnikom w końcu wypracować porozumienia w tym zakresie – efektem tego był standard MPI-1, opublikowany w 5 maja 1994. Kolejne pojawiły MPI-1.1, MPI-1.2 aż w końcu MPI-2. Kompletne specyfikacje standardów pod adresem

`http://www.mcs.anl.gov/research/projects/mpi/standard.html`

Współcześnie istnieje wiele implementacji – z natury są to implementacje typu OPEN, dla języków programowani C/C++, FORTRAN oraz ADA – w szczególności:

- ❒ OpenMP (Open Multi-Processing), z zasady jest to implementacja nastawiona na wielowątkowość, równocześnie proces jest traktowany w kategoriach wątku, zatem nie jest to rozwiązanie właściwe dla środowisk typu Distributed Memory ale Shared;
- ❒ OpenMPI rozwijana głównie przez grupę z Indiana University, realizuje standard MPI-2
- ❒ MPICH2, zarówno o przeznaczeniu dla architektur Shared jak i Distributed Memory, rozwijana głównie przez Argonne National Labolatory (faktycznie jest to najbardziej pierwotna implementacja standardu MPI).

Ponieważ jest to oprogramowanie otwarte producenci oprogramowania próbują także na tej podstawie stworzyć własną – komercyjną – implementację, jak Sun HPC Cluster Tools.

MPI aktualnie stanowi de facto standard dla niekomercyjnych i komercyjnych aplikacji HPC (high-performance computing).

Interface programowania OpenMPI

Niniejsza część materiałów dotyczy zastosowań

Open MPI: Open Source High Performance Computing
`http://www.open-mpi.org/`

Realizuje ona standard MPI-2. Może być wykorzystywana celem konstruowania oprogramowania współbieżnego dla architektur SM jak i DM.

Projekt rozwijany jest – aktualnie – przez 9 instytucji:
- Lawrence Livermore National Laboratory (USA)
- Distributed Systems Group, University of British Columbia (KANADA)
- Friedrich-Schiller-Universitat Jena, (NIEMCY)
- Grid Technology Research Center, AIST (JAPONIA)
- Platform Computing (USA)
- Computer Architecture Group, Technische Universtaet Chemnitz (NIEMCY)
- a ponadto także BULL (USA), Chelsio Communications (USA), Evergrid (USA)

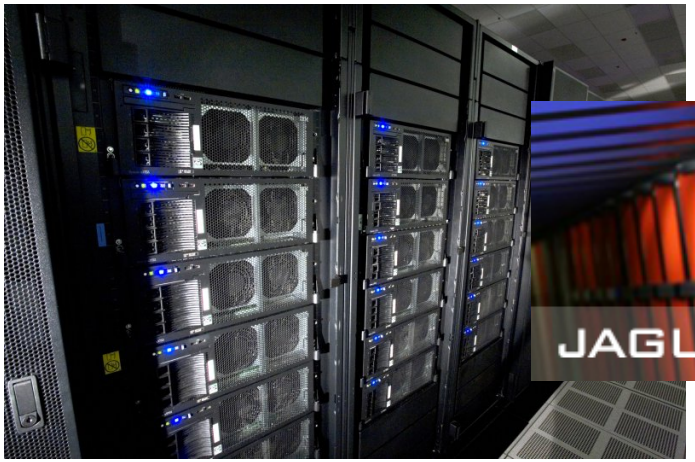
Dodatkowo zaangażowanych jest tu jeszcze 18 firm i instytucji, z czego do bardziej znanych należą:
- Los Alamos National Laboratory (USA)
- Cisco Systems, Inc. (USA)
- Intitut National de Recherche en Informatique et an Automatique (INRIA, FRANCJA)
- IBM (USA)
- Oak Ridge National Laboratory (USA)
- Sun Microsystems (USA)

Interface programowania OpenMPI

Według aktualnego rankingu top500
systemów o największej mocy obliczeniowej
http://www.top500.org/
mamy na kolejnych trzech pierwszych miejscach:

❶ Jaguar Cray XT5-HE Cluster, 224.162 6-rdzeniowych procesorach AMD x86_64 Opteron 2600 MHz , pracujący pod kontrolą systemu LINUX;
❷ Roadrunner BladeCenter QS22/LS21 Cluster, 122.400 procesorów PowerXCell 8i 3200 MHz, pod kontrolą systemu IBM System Cluster (UNIX'opodobny,POSIX);
❸ Kraken XT5 - Cray XT5-HE Cluster, 98.928 6-rdzeniowych procesorów AMD x86_64 Opteron Six Core 2600 MHz, pracujący pod kontrolą systemu LINUX;

wszystkie one wykorzystują
Open MPI: Open Source High Performance Computing



Interface programowania OpenMPI

OpenMPI jako oprogramowanie otwarte (BSD license) można pobrać w postaci kodu źródłowego z pod adresu

```
http://www.open-mpi.org/software/
```

aktualną realizacją jest

```
http://www.open-mpi.org/software/ompi/v1.4/downloads/openmpi-1.4.1.tar.gz
```

o rozmiarze 9.49MB, pochodząca z 15 stycznia 2010.

Oczywiście prawidłowe (pełne) przygotowanie do pracy OpenMPI wymaga właściwej konfiguracji wszystkich węzłów klastra oraz oprogramowania które za pośrednictwem MPI będzie uruchamianie. Ponieważ jednak tutaj omawiamy użycie OpenMPI więc zagadnienia te zostaną pominięte.

Po rozpakowaniu

```
$ tar xzf openmpi-1.4.1.tar.gz
```

w bieżącym katalogu powstaje podkatalog

```
./ openmpi-1.4.1
```

zawierający kompletne źródła, pliki pomocy i przykładu kodu a także uzupełniania.
Zmieniamy bieżący katalog

```
cd  openmpi-1.4.1
```

i uruchamiamy skrypt konfiguracyjny

```
$ ./configure -prefix=/usr/local/
```

Jeżeli konfiguracja zakończyła się sukcesem, to

```
$ make
$ make install
```

to ostatnie już koniecznie z uprawnieniami root'a.

Interface programowania OpenMPI

OpenMPI tworzy dowiązania do wrapper'a służącego kompilacji (i konsolidacji) oraz uruchomianiu plików wykonywalnych za pośrednictwem komunikatora.

Aby sprowadzić poprawność instalacji warto spróbować skompilować któryś z plików źródłowych. Najprostszym i najmniejszym z nich jest (dla języka C)

```
./openmpi-1.4/examples/hello_c.c
```

jest także wersja C++ (`hello_cxx.cc`) a także dwie FORTRAN'owe (`hello_f77.f` i `hello_f90.f90`). Zanim jednak będziemy mogli skorzystać z wrappera'a należy koniecznie zaktualizować cache konsolidatora, uruchamiając z prawami root'a

```
$ ldconfig -v | grep libopen
```

`grep` powinien wyświetlić na konsoli w uzyskanym listingu powinna się pojawić linia

```
libopen-pal.so.0 -> libopen-pal.so.0.0.0
```

jest to bliblioteka konsolidowana dynamicznie z wrapper'em niezbędna do jego pracy. Na skutek błędu w skrypcie instalacyjnym cache nie jest odświeżane i biblioteka ta, mimo że kopiowana na dysk jest niedostępna.

Kompilację (i konsolidację) najwygodniej wykonać za pomocą `mpicc` (opcje zwyczajowe, jak i w `gcc` i pozostałych)

```
$ mpicc hello_c.c -o hello
```

zaś uruchomienie

```
$ ./hello
Hello, world, I am 0 of 1
```

... program po prostu wyświetla informację na iloma procesami (wątkami) zarządza komunikator (tu: `1`) i jaki jest numer bieżącogo (tu: `0`).

Interface programowania OpenMPI

Ten sposób uruchomiania i wykorzystania komunikatora, jest w gruncie rzeczy bardzo nietypowy. Oczywiście można w ten sposób uruchamiać programy wykorzystujące API OpenMPI niemniej jest to sposób bardzo nietypowy i nie służący przetwarzaniu współbieżnemu.

Poprawnie powinniśmy użyć w tym celu wrapper'a **mpirun** podając ile procesów (**-np** lub **-n**) ma być uruchomionych

```
$ mpirun -np 4 ./hello
Hello, world, I am 0 of 4
Hello, world, I am 1 of 4
Hello, world, I am 2 of 4
Hello, world, I am 3 of 4
```

W trakcie uruchomiania programu, w linii komend **mpirun** można podać *explicite* plik konfiguracyjny określający ile zadań i na którym węźle klastra ma być uruchomionych. Służy temu opcje (równoważnie) **--machinefile** lub **-hostfile**, przykładowo

```
$ mpirun --hostfile hosts ./hello
Hello, world, I am 0 of 4
Hello, world, I am 1 of 4
Hello, world, I am 2 of 4
Hello, world, I am 3 of 4
```

a plik **hosts** poszukiwany jest domyślnie w bieżącym katalogu. W tym przypadku jego zawartość była następująca

```
10.60.20.76 slots=4
```

Identyczny efekt przyniesie użycie opcji --host

```
$ mpirun --host 10.60.20.76,10.60.20.76,10.60.20.76,10.60.20.76 ./hello
```

Plik zadawany opcją **--machinefile** lub **--hostfile** zawiera listę węzłów określonych albo poprzez nazwę

```
node17.ath.bielsko.pl
```

albo poprzez IP

```
10.20.24.51
```

bezpośrednio po nazwie może być podana ilość procesorów, a właściwie procesów jaka może być uruchamiana na danych węźle

```
10.60.20.76 slots=2 max_slots=4
10.20.24.51 slots=2 max_slots=2
```

przy czym szeregowanie zadań na węzłach może się odbywać wg slotu (**--byslot**, domyślnie) albo wg węzła (**--bynode**). Przyjrzyjmy się różnicą, zakładając że plik **hosts** zawiera linie jak wyżej. Dla **--bynode**

```
$ mpirun --hostfile hosts -np 6 --byslot | sort
Hello World I am rank 0 of 6 running on 10.60.20.76
Hello World I am rank 1 of 6 running on 10.60.20.76
Hello World I am rank 2 of 6 running on 10.20.24.51
Hello World I am rank 3 of 6 running on 10.20.24.51
Hello World I am rank 4 of 6 running on 10.60.20.76
Hello World I am rank 5 of 6 running on 10.60.20.76
```

zaś wybierając **--bynode**

```
$ mpirun --hostfile hosts -np 6 --bynode | sort
Hello World I am rank 0 of 6 running on 10.60.20.76
Hello World I am rank 1 of 6 running on 10.20.24.51
Hello World I am rank 2 of 6 running on 10.60.20.76
Hello World I am rank 3 of 6 running on 10.20.24.51
Hello World I am rank 4 of 6 running on 10.60.20.76
Hello World I am rank 5 of 6 running on 10.60.20.76
```

Interface programowania OpenMPI

Chcąc wprowadzić w kodzie programu przetwarzanie współbieżne z wykorzystaniem MPI, należy w części początkowej zawrzeć instrukcje inicjujące a końcowej zamykające komunikator.
Nie jest to trudne bowiem służą temu wywołania zaledwie dwóch funkcji:

```
int MPI_Init( int *argc, char ***argv );
int MPI_Finalize( void );
```

zwracające 0 w przypadku sukcesu lub kod błędu w przypadku niepowodzenia. Deklaracja nagłówkowa tej i wszystkich pozostałych funkcji MPI znajduje się w

```
#include <mpi.h>
```

W najprostszym przypadku kod programu korzystającego z MPI, będzie miał postać jak w listingu.

```
#include <stdio.h>
#include <mpi.h>
int main( int argc,char **argv )
{
    MPI_Init( &argc,&argv );
    printf( "...programowanie z MPI jest proste\n" );
    MPI_Finalize();
    return 0;
}
```

Jeżeli zapiszemy pod nazwą proste.c, to kompilacja

```
$ mpicc proste.c -o proste
```

a wykonanie na dwóch procesach

```
$ mpirun -np 2 ./proste
...programowanie z MPI jest proste
...programowanie z MPI jest proste
```

Interface programowania OpenMPI

Już na tym przykładzie, że na poziomie procesu konieczna jest możliwość jego identyfikacji. Służą temu dwie dalsze funkcje

```
int MPI_Comm_size( MPI_Comm comm, int *size );
int MPI_Comm_rank( MPI_Comm comm, int *rank );
```

przy czym parametr formalny `comm` określa komunikator, którego odwołania dotyczy – bieżący

```
MPI_COMM_WORLD
```

Pierwsza określa ilość procesów współbieżnych obsługiwanych przez dany komunikator, zaś druga numer identyfikujący bieżący proces w danej grupie (de facto jest to indeks, czyli od `0` do `size-1`).

W przykładzie pobierzemy informację o ilości procesów, a następnie każdy poda informację o sobie.

```
#include <stdio.h>
#include <mpi.h>
int main( int argc, char *argv[] )
{
    int world,current;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &current );
    MPI_Comm_size( MPI_COMM_WORLD, &world );
    printf("proces %d z %d\n",current+1,world );
    MPI_Finalize();
    return 0;
}
```

Jeżeli w takim razie zapiszemy kod źródłowy pod nazwą **kolejne.c**, to jego kompilacja

```
$ mpicc -Wall kolejne.c -o kolejne
```

a wykonanie na 3 procesach

```
$ mpirun -np 3 ./kolejne
proces 1 z 3
proces 2 z 3
proces 3 z 3
```

Warto pamiętać, że - tradycyjnie – jeżeli proces (wątek) będzie wykonywał jakiekolwiek zadania i może wystąpić niezrównoważenie obciążeń, to trzeba zapewnić przynajmniej synchronizację w pewnych punktach

```
int MPI_Barrier(MPI_Comm comm)
```

Naturalnym i często występującym elementem w kodzie współbieżnym jest jego różnicowanie między procesami. Równocześnie dość praktyczną możliwością jest pobranie nazwy węzła na którym znajduje się dany proces, czemu służy funkcja

```
int MPI_Get_processor_name( char *name,int *resultlen );
```

Pierwszy parametr określa nazwę symboliczna węzła w postaci łańcucha znakowego a drugi jego długość. Deklarując zmienną name wygodnie jest tu skorzystać ze stałej **MPI_MAX_PROCESSOR_NAME** zdefiniowanej w **mpi.h** aktualnie jako

```
#define MPI_MAX_PROCESSOR_NAME 256
```

Kolejny przykład prezentuje sposób różnicowania kodu między węzeł nadrzędny identyfikowany stałą symboliczną **MASTER** i pozostałe.

INTERFACE PROGRAMOWANIA OPENMPI

```
#include<stdio.h>
#include<mpi.h>
#define MASTER 0   //...wyróżniony węzeł MASTER o indeksie 0

int main( int argc, char *argv[] )
{
    int  world,current,len;
    char name[MPI_MAX_PROCESSOR_NAME+1];

    MPI_Init( &argc,&argv );                    //...inicowanie komunikatora
    MPI_Comm_size( MPI_COMM_WORLD,&world );     //...ilość procesów skojarzonych
    MPI_Comm_rank( MPI_COMM_WORLD,&current );   //...indeks bieżącego procesu
    MPI_Get_processor_name( name,&n );
    if( current==MASTER )
    { printf( "MASTER" ); }                     //...zadania dla MASTERa
    else
    { printf( "SLAVE"); }                       //...zadania dla SLAVE'ów
    //...o tego miejsca kod wspólny, dla wszystkich procesów
    printf( "\t%d.%d [%lu->%lu] | węzeł: %s\n",current+1,world,
           (unsigned long)getppid(),(unsigned long)getpid(),name );

    MPI_Finalize();                             //...zamykamy komunikator
    return 0;
}
```

Interface programowania OpenMPI

Sprawdźmy najpierw

```
$ ps -o cmd,ppid,pid
CMD                          PPID   PID
/bin/bash                    4339  4340
ps -o cmd,ppid,pid           4340  4907
```

Jeżeli skompilujemy ten kod pod nazwą **slaves**, to wykonanie dla 6 procesów

```
$ mpirun -np 6 ./slaves
MASTER  1.6 [4943->4944] | węzeł: localhost
SLAVE   2.6 [4943->4945] | węzeł: localhost
SLAVE   3.6 [4943->4946] | węzeł: localhost
SLAVE   4.6 [4943->4947] | węzeł: localhost
SLAVE   5.6 [4943->4948] | węzeł: localhost
SLAVE   6.6 [4943->4949] | węzeł: localhost
```

Zwróćmy uwagę na wyświetlane numery identyfikacyjne PPID i PID.

Interface programowania OpenMPI

Celem zapewnienia przenaszalności i niezależności od platformy systemowej MPI wprowadza nazwy równoważne typów. Jest ich bardzo wiele, a w szczególności

```
MPI_DATATYPE_NULL

MPI_BYTE
MPI_CHAR
MPI_UNSIGNED_CHAR
MPI_SIGNED_CHAR

MPI_INT
MPI_LONG
MPI_UNSIGNED_LONG
MPI_UNSIGNED
MPI_LONG_INT
MPI_LONG_LONG_INT
MPI_LONG_LONG
MPI_UNSIGNED_LONG_LONG

MPI_FLOAT
MPI_DOUBLE
```

Nie oznacza to wcale, że użycie nazw typów języków jest niedozwolone.

INTERFACE PROGRAMOWANIA OPENMPI

Współbieżność z reguły wymaga międzyprocesowej wymiany danych. Może ona mieć formę:

- blokującą, kiedy operacja wymiany danych nie zwraca sterowania dopóki dane i powiązane z nimi nie zostaną odebrane w miejscu docelowym tak że pośredniczący bufor będzie mógł być użyty ponownie przez proces wysyłający;
- nieblokującą, kiedy wymiana danych realizowana jest równolegle z innymi operacjami procesu, nie mającymi jednak wpływ na zawartość bufora.

W przypadku wymiany blokującej dane wysyłane są za pośrednictwem funkcji

```
int MPI_Send(  void *buffer, int count, MPI_Datatype type,
               int destination, int tag,MPI_Comm communicator );
```

gdzie:

buffer -wskazaniem do tablicy, która będzie wysłana,
count -ilość elementów tablicy **buffer**,
type -nazwa równoważna typu elementów tablicy **buffer**,
destination -numer procesu dla którego wiadomość jest przeznaczona
tag -liczba całkowita identyfikująca rodzaj danych
(umożliwiająca rozpoznanie odbiorcy ich przeznaczenia),
communicator -identyfikator określający komunikator, zwykle **MPI_COMM_WORLD**.

która sprawdza czy przez obszar pamięci użyty do komunikacji może być wykonana kolejna operacja **MPI_Send()**.

Wariantem tej funkcji nakładającej silniejsze więzy na bufor jest

```
int MPI_Ssend( void *buffer, int count, MPI_Datatype type, int destination,
int tag,MPI_Comm communicator );
```

realizująca komunikację blokującą, synchroniczną, sprawdzająca dodatkowo czy proces dla którego dane są przeznaczone rozpoczął ich odbiór. Wówczas to proces wysyłający uzyskuje potwierdzenie, że nastąpił odbiór, co nie będzie miało miejsca w przypadku `MPI_Send()` (która dodatkowo może być w różny sposób implementowana).

Dane odbieraną są za pomocą

```
int MPI_Recv(  void *buffer, int count, MPI_Datatype type, int source, int tag,
MPI_Comm communicator, MPI_Status *status );
```

oprócz parametrów formalnych identycznych jak w przypadku `MPI_Send()`, mamy tu jeszcze wskazanie do struktury status , której składowe identyfikują pochodzenie i rodzaj komunikatu

`status.MPI_SOURCE` -numer procesu który wysłał komunikat,

`status.MPI_TAG` -identyfikator rodzaju danych.

W przypadku jeżeli proces oczekuje dowolnego komunikatu, czy komunikatu pochodzącego z dowolnego źródło, to celem ich identyfikacji można użyć stałych symbolicznych

`MPI_ANY_SOURCE`

`MPI_ANY_TAG`

Jeżeli komunikacja międzyprocesowa wymaga równoległego przesłanie i odbioru danych między dwiema identycznymi procesami można użyć w tym celu funkcji

```
int MPI_Sendrecv(  void *buffer,int scount,MPI_Datatype stype,int destination,
int stag,void rbuffer,int rcount,MPI_Datatype rtype, int source,
int rtag,MPI_Comm communicator, MPI_Status *status );
```

w miejsce wywołań `MPI_Send()` i `MPI_Recv()`.

Interface programowania OpenMPI

Jako przykład użycia komunikacji blokującej przygotujemy program w którym węzły slave prześlą pojedynczą liczbą do węzła (procesu) master.

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

#define MASTER 0
#define TAG    'R'+'i'+'R'

int main(int argc, char *argv[])
{
    int current,world,counter;
    double x,Sx;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    //...pobieramy informacje o ilości procesów i indeksie do bieżącego
    MPI_Comm_size( MPI_COMM_WORLD,&world );
    MPI_Comm_rank( MPI_COMM_WORLD,&current );
    //...jeszcze tylko wartość startowa dla generatora zmiennych pseudolosowych
    srand( (unsigned)time( NULL ) );
    //...i możemy zaczynać
```

INTERFACE PROGRAMOWANIA OPENMPI

```
    if( current!=MASTER ) //...czyli dla SLAVE'ów
    {
        x = (double)rand()/RAND_MAX;
        MPI_Send( (void*)(&x),1,MPI_DOUBLE,MASTER,TAG,MPI_COMM_WORLD );
    }
    else //... tym razem dla MASTER'a
    {
        for( counter=MASTER+1,Sx=0.0;counter<world;counter++ )
        {
            MPI_Recv( (void*)(&x),1,MPI_DOUBLE,counter,TAG,MPI_COMM_WORLD,&status );
            Sx += x;
        }
        printf(     "\nProces %d odebrał od %d do %d,
                    sumę Sx=%f\n\n",MASTER,MASTER+1,world-1,Sx );
    }

    //...od tego miejsca już kod wspólny
    MPI_Finalize();
    return 0;
}
```

Efekt wykonania - powiedzmy - dla 100 procesów

```
$ mpirun -np 100 ./block
Proces 0 odebrał od 1 do 99, sumę Sx=49.318779
```

Interface programowania OpenMPI

Nie zawsze koordynacja czasowa wymiany danych ma znaczenie krytyczne dla procesu, tak że niekoniecznie muszą one wiązać się z koniecznością wprowadzenia blokowania. W przypadku operacji nieblokujących sterowanie zwracane jest natychmiastowo do procesu, tak że może on podjąć dalsze działania.

Dane przysyłane są wówczas wywołaniem

```
int MPI_Isend( void *buffer,int count, MPI_Datatype type, int destination, int tag,
MPI_Comm communicator, MPI_Request *request );
```

Znacznie parametrów formalnych jest tu identyczne z `MPI_Send()`, przy czym występuje tutaj uzupełniająco jeszcze

`request` -stanowiący wskazanie do struktury umożliwiającej późniejsze sprawdzenie stanu operacji.

Analogicznie jak w przypadku operacji blokujących możliwe jest tu zastosowanie także funkcji realizującej przesył synchroniczny danych

```
int MPI Issend( void *buffer,int count, MPI_Datatype type, int destination,
int tag,MPI_Comm communicator, MPI_Request *request );
```

Dane odbierane są za pośrednictwem

```
int MPI_Irecv( void *buffer, int count, MPI_Datatype type, int source, int tag,
MPI_Comm communicator,MPI_Request *request );
```

Interface programowania OpenMPI

Mimo zastosowania komunikacji nieblokującej, w pewnym momencie, zaistnieje zawsze potrzeba iż dane zostały w końcu odebrane czy wysłane. Umożliwia to request, przekazany do sprawdzenia funkcji

```
int MPI_Test( MPI_Request request, MPI_Status *status );
```

które w tym momencie wprowadza blokadę, albo

```
int MPI_Wait( MPI_Request request,int *flag, MPI_Status *status );
```

dokonującej wyłącznie sprawdzenia, czy dana operacja została zakończona. Nie wprowadza ona jednak blokady lecz testuje stan operacji, czego wynik zwraca w zmiennej flag, przyjmującej wartości **TRUE** albo **FALSE**.

Jeżeli proces wykonał ciąg operacji przesyłu danych, to sprawdzenie ich stanu można wykonać dla wszystkich poprzedzających, wywołaniami funkcji

```
int MPI_Testall( int count,MPI_Request *request, MPI_Status *status );
```

lub

```
int MPI_Waitall( int count, MPI_Request *request,int *flag, MPI_Status *status );
```

które zwraca status operacji podanych w tablicy **request**.

Interface programowania OpenMPI

Kolejny przykład użycia **MPI_Send()** i **MPI_Receive()**, tym razem celem wyznaczenia wariancji. Zwróćmy tutaj uwagę na sposób różnicowania kodu pomiędzy procesy.

```
#include <stdio.h>
#include <mpi.h>
#define MASTER 0
#define TAG    'K'+'M'
#define MEGA   100000LU

int main( int argc, char *argv[] )
{
    unsigned long N,counter;
    double x,Sxx,Var;
    int  world,current,len;
    char name[MPI_MAX_PROCESSOR_NAME+1];
    MPI_Status status;

    MPI_Init(&argc, &argv); //...standardowe inicjacja MPI
    MPI_Comm_size( MPI_COMM_WORLD, &world );

    if( argc>1 ){ sscanf( argv[1],"%lu" ,&N  ); }else{ N = 0; }
    N = ( N<1 )?( 1*MEGA  ):( N*MEGA ); //...ustalmy ilość składników

    MPI_Comm_rank( MPI_COMM_WORLD,&current );
```

INTERFACE PROGRAMOWANIA OPENMPI

```
    MPI_Get_processor_name( name,&len );
    printf( "(proces%2d)@%s",current,name );

    if( current==MASTER )
    {
        printf( "...ilość składników  %lu*%lu\n",N/MEGA,MEGA );
        for( current=MASTER+1,Sxx=0.0;current<world;current++ )
        {
            MPI_Recv( (void*)&x,1,MPI_DOUBLE,current,TAG,MPI_COMM_WORLD,&status );
            Sxx += x;
        }
        Var = Sxx/(N-1);
        printf( "...MASTER zyskał wariancję Var=%g\n",Var );
        printf( "   dla %lu*%lu składników sumy\n",N/MEGA,MEGA );
    }
    else
    {
        for( counter=current,Sxx=0.0;counter<=N;counter+=(world-1) )
        { x = (double)rand()/RAND_MAX; Sxx += (x-0.5)*(x-0.5); }
    printf( "...SLAVE wysyła sumę %g\n",Sxx );
    MPI_Send( (void*)&Sxx,1,MPI_DOUBLE,MASTER,TAG,MPI_COMM_WORLD );
    }
    MPI_Finalize();
    return 0;
}
```

INTERFACE PROGRAMOWANIA OPENMPI

Przykładowy efekt wykonania dla 5 procesów

```
mpirun -np 5 ./sr 10
(proces 0)@localhost...ilość składników  10*100000
(proces 1)@localhost...SLAVE wysyła sumę 20787.4
(proces 2)@localhost...SLAVE wysyła sumę 20787.4
(proces 3)@localhost...SLAVE wysyła sumę 20787.4
(proces 4)@localhost...SLAVE wysyła sumę 20787.4
...MASTER zyskał wariancję Var=0.0831498
   dla 10*100000 składników sumy
```

Ponieważ sytuacja taka kiedy grupa procesów wykonuje określone obliczenia a wynik opracowuje wybrany, jest niezmiernie częsta w związku z tym przewidziano w tym celu specjale funkcje.

```
int MPI_Reduce(void *send, void *receive, int count,MPI_Datatype type,
                    MPI_Op op,int root, MPI_Comm comm );
```

Parametrami wejściowymi są:

- **send** bufor do wysłania
- **count** ilość elementów w buforze
- **type** typ danych w buforze
- **op** rodzaj operacji redukcji
- **root** adresat operacji (odbiorca)
- **comm** komunikator

a wyjściowym

- **receive** czyli bufor przeznaczenia (odbiorcy)

Interface programowania OpenMPI

Istnieje także wersja tej funkcji, w efekcie wywołania której wszystkie procesy komunikatora otrzymują wynik operacji

```
int MPI_Allreduce( void *send, void *receivef, int count, MPI_Datatype datatype,
                   MPI_Op op,MPI_Comm comm);
```

Operacja redukcji może polegać na wyznaczeniu, w szczególności:

`MPI_MAX`	maksimum
`MPI_MIN`	minimum
`MPI_SUM`	sumy
`MPI_PROD`	iloczynu
`MPI_LAND`	logicznego AND
`MPI_BAND`	bitowego AND
`MPI_LOR`	logicznego OR
`MPI_BOR`	bitowego OR
`MPI_LXOR`	logicznego XOR
`MPI_BXOR`	bitowego XOR

Interface programowania OpenMPI

W takim razie nieco inny sposób rozwiązania wcześniejszego zadania.

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

#define MASTER 0
#define MEGA   100000LLU

int main( int argc, char *argv[] )
{
    unsigned long long N,i;
    double x,Sxx,Var;
    int  world,current;

    MPI_Init(&argc, &argv);
    MPI_Comm_size( MPI_COMM_WORLD, &world );

    if( argc>1 ){ sscanf( argv[1],"%llu" ,&N  ); }else{ N = 0; }
    N = ( N<1 )?( 1*MEGA  ):( N*MEGA );

    MPI_Comm_rank( MPI_COMM_WORLD,&current );

    for( i=current,Sxx=0.0;i<N;i+=world ) //...teraz wykonują wszyscy
    {  x    = (double)rand()/RAND_MAX; Sxx += (x-0.5)*(x-0.5); }
```

```
    MPI_Reduce( &Sxx, &Var, 1,MPI_DOUBLE,MPI_SUM,MASTER,MPI_COMM_WORLD );

    if( current==MASTER )
    {
        printf( "Ilość elementów   N*M=%llu*%llu\n",N/MEGA,MEGA );
        printf( "Otrzymana wartość Var= %lf\n",Var/(N-1) );
    }

    MPI_Finalize();

    return 0;
}
```

Zauważmy, w wywołaniu **MPI_Reduce()** mamy

Sxx, **Var**	bufor wysyłki i odbioru
1	ilość danych do wysłania/odbioru
MPI_DOUBLE	typ danych w buforach wysyłania/odbioru
MPI_SUM	rodzaj wykonywanej operacji redukcji
MASTER	adresat przesyłki
MPI_COMM_WORLD	identyfikator komunikatora

Przykładowy wynik

```
$ mpirun -np 4 ./reduce 10
Ilość elementów   N*M=10*100000
Otrzymana wartość Var= 0.083150
```

Interface programowania OpenMPI

Operacja dualną względem redukcji, a więc przesyła zawartość bufora do wszystkich pozostałych procesów, jest

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

której efektem jest rozesłanie danych

- **buffer** miejsce przechowywania danych do wysyłki (wskazanie)
- **count** ilość danych do wysłania
- **type** typ danych
- **root** indeks procesu wysyłającego (tutaj nadrzędny)

Przykładowo gdyby:

- wysłać z procesu identyfikowanego indeksem **MASTER**
- do wszystkich pozostałych skojarzonych z komunikatorem **MPI_COMM_WORLD**,
- 100 liczb
- całkowitych (**MPI_DOUBLE**)
- identyfikowanych przez **double *X**, zgodnie z deklaracją

to składnia wywołania funkcji byłaby

```
MPI_Bcast( (void*)X,100,MPI_DOUBLE,MASTER,MPI_COMM_WORLD );
```