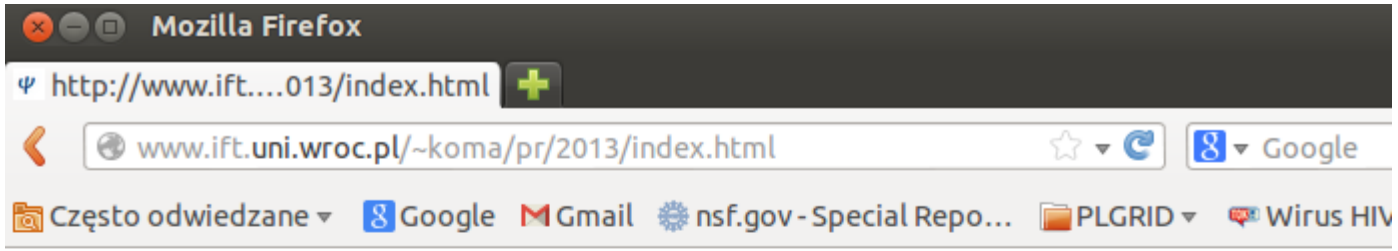
Uniwersytet Wrocławski

Programowanie Równoległe
Wykład 3

MPI - Message Passing Interface

Maciej Matyka
Instytut Fizyki Teoretycznej



Programowanie Rownolegle

Zbigniew Koza, Maciej Matyka

Lista wykladow:

- Wyklad 1, Wstep (ZK), 7/10/2013
- Wyklad 2, Projektowanie algorytmow rownoleglych (ZK), 14/10/2013 -
- Wyklad 3, MPI (MM), 21/10/2013 [pdf] [lista1]

  - Lista osob: [link]
  - http://www.mpi-forum.org/
  - http://people.sc.fsu.edu/~jburkardt/pdf/mpi_course.pdf
  - http://www.zib.de/zibdoc/mpikurs/

Uniwersytet Wrocławski

Jak optymalizować?

Donald Knuth:

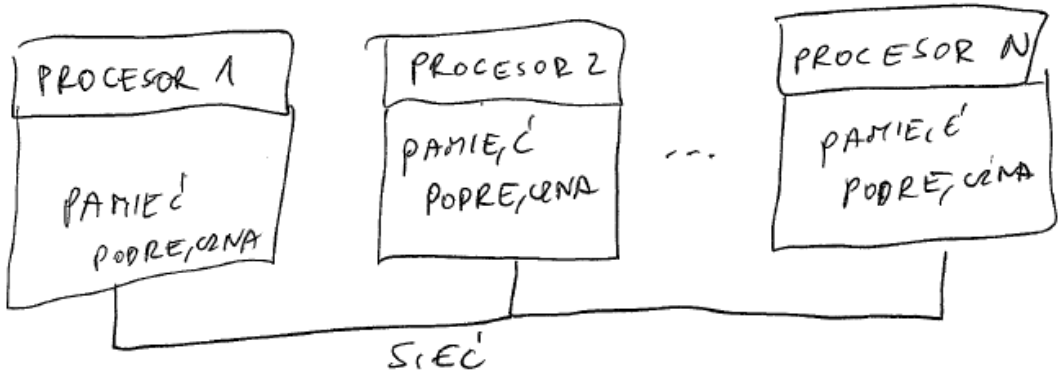
> „We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”

1. Napisanie działającego kodu
2. Weryfikacja rozwiązań
3. Ustalenie miejsc kluczowych do optymalizacji
4. Optymalizacja miejsc kluczowych

Modele pamięci

- Architektura z pamięcią współdzieloną

- Architektura z pamięcią rozproszoną



Uniwersytet Wrocławski

Modele programowania =

Jak programować równolegle?

- A) Pula wątków (POSIX Threads)
  - architektury z pamięcią współdzieloną
  - Ręczne tworzenie i obsługa komunikatów między wątkami programu.
- B) dyrektywy kompilatora (OpenMP)
  - architektury z pamięcią współdzieloną
  - kompilator urównolegla kod wg dyrektyw preprocesora
  - Podobne rozwiązanie dla GPU - OpenACC
- C) jawne przesyłanie komunikatów (MPI)
  - architektury z pamięcią rozproszoną
  - ręczne programowanie wymiany komunikatów

MPI

Message Passing Interface

- Standard komunikacji
- Implementacje: np. MPICH, OpenMPI
- C / C++ / Fortran77 / Fortran90
- dla programisty: zbiór procedur i funkcji
- http://www.mpi-forum.org

Wersje

- V1.0, 1994
- V1.1, 1995 – drobne poprawki
- V1.2, 1997 – drobne poprawki
- V2.0, 1997 – duże zmiany, nowy standard

MPI-1

- V1.3, 2008 – poprawki do „starej” wersji
- V2.1, 2008
- V2.2, 2009
- V3.0, 2012

MPI-2

„(...) Standard MPI-1.2 okazał się być uniwersalnym i obsługiwanym przez zdecydowaną większość klastrów obliczeniowych. Jednakże MPI-2.1 okazał się być bardziej limitowanym rozwiązaniem. Główne powody ku temu to: (...)”

za http://pl.wikipedia.org/wiki/Message_Passing_Interface

Kiedy używać MPI?

Kiedy nie pisać kodu samodzielnie z MPI?

- Jeśli możemy uniknąć obliczeń równoległych
- Jeśli możliwe jest równoległe uruchomienie tego samego programu na zbiorze różnych danych
- Jeśli możemy użyć biblioteki w wersji równoległej

Kiedy używać MPI?

- Pracujemy na systemie z pamięcią rozproszoną
- Chcemy mieć przenośny kod równoległy

Hello World

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

int main (int argc, char *argv[])
{
        MPI_Init(&argc, &argv);
        printf("Hello world!\n");
        MPI_Finalize();
        return 0;
}
```

```
maq@geordi ~/pr $ ls
hello.c
maq@geordi ~/pr $ gcc hello.c -lmpi
maq@geordi ~/pr $ mpirun -np 4 ./a.out
Hello world!
Hello world!
Hello world!
Hello world!
maq@geordi ~/pr $
```

```
PROGRAM hello

INCLUDE 'mpif.h'
INTEGER err
CALL MPI_INIT(err)
PRINT *, "Hello world!"
CALL
MPI_FINALIZE(err)

END
```

```
maq@zero ~/pr $ ls
a.out  hello.f  hello.f~
maq@zero ~/pr $ mpif77 hello.f
maq@zero ~/pr $ mpirun -np 8 ./a.out
 Hello world!
 Hello world!
 Hello world!
 Hello world!
 Hello world!
 Hello world!
 Hello world!
 Hello world!
maq@zero ~/pr $
```

Nazewnictwo

- Pierwszy człon funkcji MPI zapisujemy z dużej litery:

```
MPI_Init(&argc, &argv);
MPI_Finalize();
```

- Każdy następny człon nazwy – z małej:

```
MPI_Comm_rank(...)
MPI_Rank_size(...)
```

Uniwersytet Wrocławski

ID procesu

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

```
int rank;          // unikalna liczba (numer procesora)
MPI_Comm comm;     // komunikator
```

Komunikator - grupa procesów + kontekst

Kontekst – dodatkowe informacje (np. topologia, identyfikatory)

Komunikator standardowy to: MPI_COMM_WORLD

i zawiera listę wszystkich procesów.

Hello World 2

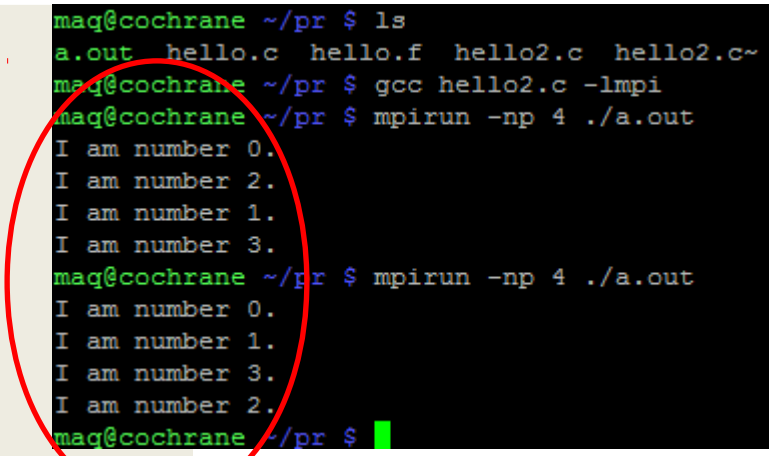
- Unikalne ID procesora:

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

int main (int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("I am number %d.\n", rank);

  MPI_Finalize();
  return 0;
}
```



Kolejność!

Komunikat

- Komunikat w MPI składa się 2 części:

- Treść
  - Bufor
  - Typ danych (typy wbudowane MPI - komentarz)
  - Ilość elementów

  Analogia do tablicy c: TYP tablica[N];

- Koperta
  - Nadawca
  - Odbiorca
  - Komunikator (np. MPI_COMM_WORLD)
  - Tag

  Ważne: odbiorca musi podać kopertę komunikatu do odebrania.

Uniwersytet Wrocławski

MPI_Send

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

treść

koperta

- Wysyła komunikat do procesu o id=dest
- Niejawnie przekazywany jest id nadawcy
- tag – może służyć do rozróżniania typów komunikatów (np. status, dane itp.)
- MPI_Send zwraca kod błędu (lub MPI_SUCCESS)
- MPI_Send blokuje wykonanie programu do czasu odebrania komunikatu i odesłania potwierdzenia

Uniwersytet Wrocławski

MPI_Recv

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

treść

koperta

- odbiera komunikat od procesu o id=src
- wybiera komunikat o określonym tagu
- MPI_Recv blokuje wykonanie programu do czasu odebrania komunikatu i odesłania potwierdzenia
- MPI_Status zawiera informacje o odebranych danych (ich ilość, źródło etc. Na przykład, gdy src=MPI_ANY_SOURCE)

- Uwaga: MPI_Send, MPI_Recv są funkcjami blokującymi, możliwe łatwe zatrzymanie programu w nieskończonej pętli.

Uniwersytet Wrocławski

Przykład komunikacji

```
int main( int argc, char **argv )
{
      char message[20];
      int myrank;
      MPI_Status status;

      MPI_Init( &argc, &argv );
      MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

       if (myrank == 0)        // kod dla procesu 0
      {
          strcpy(message,"Hello, here!");
         MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99,
      MPI_COMM_WORLD);
      }
      else             // kod dla procesu 1
      {
         MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
         printf("received :%s:\n", message);
      }

      MPI_Finalize();
      return 0;
}
```

```
maq@zero ~/pr $ mpirun -np 2 ./a.out
received :Hello, there:
maq@zero ~/pr $
```

Rodzaje komunikacji

- Blokująca – nadawca wysyła komunikat i czeka na potwierdzenie odbioru
- Nieblokująca – nadawca wysyła komunikat i nie oczekuje na potwierdzenie (może je odebrać w dowolnym momencie)

Uwaga:

Wprowadzone właśnie MPI_Send i MPI_Recv są blokujące.

Uniwersytet Wrocławski

Funkcje nieblokujące

```
int MPI_Isend(   void *buf,
                 int count,
                 MPI_Datatype datatype,
                 int dest,
                 int tag,
                 MPI_Comm comm,
                 MPI_Request *request);
```

- MPI_Request – identyfikator zadania, pozwala na sprawdzenie czy zadanie zostało wykonane

Funkcje nieblokujące

```
int MPI_Irecv(    void *buf,
                  int count,
                  MPI_Datatype datatype,
                  int src,
                  int tag,
                  MPI_Comm comm,
                  MPI_Request *request);
```

- MPI_Request – identyfikator zadania, pozwala na sprawdzenie czy zadanie zostało wykonane

Uniwersytet Wrocławski

Oczekiwanie na wykonanie

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

- Proces po wykonaniu MPI_Isend/MPI_Irecv może zaczekać na wykonanie zadania o określonym identyfikatorze MPI_Request
- status
  - dla MPI_Isend może zawierać kod błędu operacji
  - dla MPI_Irecv standardowy MPI_Status

Sprawdzenie wykonania

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

- Proces może też sprawdzić wykonanie zadania o określonym identyfikatorze MPI_Request
- flag (true/false): zadanie wykonane/niewykonane

Uniwersytet Wrocławski

Tryby wysyłania komunikatów

- synchroniczny
  - Wysłanie i odebranie są zsynchronizowane. W przypadku blokującym nadawca po zgłoszeniu wiadomości czeka, aż odbiorca zacznie odbieranie, wtedy kontynuuje pracę. Uwaga: to nie oznacza, że odbiorca skończył odbierać wiadomość.
- gotowości
  - Wymagane jest, by proces odbierający zgłosił wcześniej wiadomość do odebrania. W przypadku gdy tego nie zrobił zachowanie jest nieokreślone. Ten tryb zwiększa wydajność komunikacji.
- buforowany
  - Komunikacja przez bufor, zgłoszenie jest równoważne przesłaniu danych do bufora tymczasowego z którego jest odbierany w dowolnym momencie.

Komunikacja buforowana

```
int MPI_Ibsend( void *buf,
                int count,
                MPI_Datatype datatype,
                int dest,
                int tag,
                MPI_Comm comm,
                MPI_Request *request);
```

Example 3.11 Calls to attach and detach buffers.

```
#define BUFFSIZE 10000
int size;
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */
```

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

A quick overview of MPI's send modes

MPI has a number of different "send modes." These represent different choices of buffering (where is the data kept until it is received) and synchronization (when does a send complete). In the following, I use "send buffer" for the user-provided buffer to send.

MPI_Send

MPI_Send will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).

MPI_Bsend

May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.

MPI_Ssend

will not return until matching receive posted

MPI_Rsend

May be used ONLY if matching receive already posted. User responsible for writing a correct program.

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

A quick overview of MPI's send modes

MPI_Isend

Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free). Note also that while the I refers to immediate, there is no performance requirement on MPI_Isend. An immediate send must return to the user without requiring a matching receive at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does not block waiting for a matching receive. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application.

MPI_Ibsend

buffered nonblocking

MPI_Issend

Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted.

MPI_Irsend

As with MPI_Rsend, but nonblocking.

Tryby wysyłania komunikatów

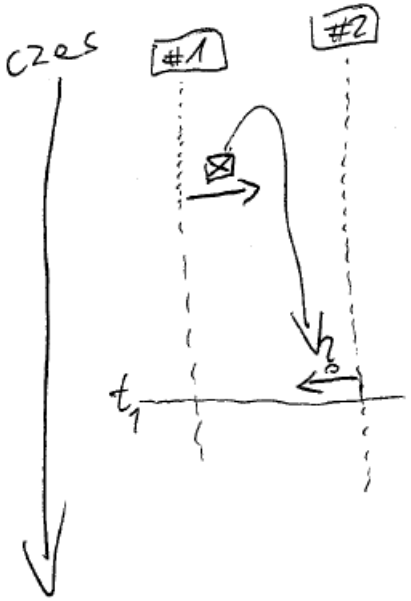
Tryb	Blokująca	Nieblokująca
Standardowy	MPI_Send	MPI_Isend
Synchroniczny	MPI_Ssend	MPI_Issend
Gotowości	MPI_Rsend	MPI_Irsend
Buforowany	MPI_Bsend	MPI_Ibsend

Uwaga: Nieblokujące MPI_Isend oraz MPI_Issend różnią się ze względu na sposób działania MPI_Wait/MPI_Test.

Szczegółowy opis:
http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

Komunikacja, uwagi

- Proces może odebrać komunikat długo po jego nadaniu przez nadawcę
- Proces może zainicjować odbieranie komunikatu na długo przed jego wysłaniem przez nadawcę (!)



Komunikacja

- aktywny udział nadawcy i adresata
- różne czasy wywołania
- najczęściej, najlepiej **niesynchroniczna**
- 1 proces może być adresatem wielu komunikatów
- proces wybiera dowolny komunikatat z kolejki (niekoniecznie w kolejności nadania)

Uniwersytet Wrocławski

Literatura

- http://people.sc.fsu.edu/~jburkardt/pdf/mpi_course.pdf (polecam!)
- MPI: A Message-Passing Interface Standard, Version 1.3
- http://aragorn.pb.bialystok.pl/~wkwedlo/PC3.pdf

Uniwersytet Wrocławski

MPI, kolejny wykład.

28 Października 2013, MPI – część 2

Plan:

- komunikacja kolektywna (MPI_Reduce, MPI_Scatter etc.)
- typy pochodne (MPI_Type_)
- przykłady praktyczne

Uniwersytet Wrocławski

Koniec.

Uniwersytet Wrocławski

SLAJDY ODRZUCONE

Uniwersytet Wrocławski

Modele komunikacji

- Niesynchroniczna – nadawca zgłasza komunikat. Dane kopiowane są do bufora pośredniego. Dane są wysyłane z bufora pośredniego po tym jak odbiorca zgłosi chęć odebrania.

- Synchroniczna – nadawca zgłasza komunikat. Dane komunikatu wysyłane są dopiero po tym jak odbiorca zgłosi chęć odebrania.

Typy danych w MPI

To ważne żeby używać wbudowanych typów do komunikacji (komunikacja między różnymi systemami).

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI_BYTE – 8 bitów, ogólny typ dla dowolnych danych
MPI_PACKED – typ danych jawnie scalonych (nie: spakowanych)

Ćwiczenia

Jedno z zadań na ćwiczenia polega na zrozumieniu i Poprawieniu błędu w poniższym programie:

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv)
{
  int myrank;
  MPI_Status status;
  double a[100], b[100];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if( myrank == 0 )
  {
    // odbierz i wyślij komunikat
    MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  }
  else if( myrank == 1 )
  {
    // odbierz i wyślij komunikat
    MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
    MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
  }
  MPI_Finalize();
}
```

Uniwersytet Wrocławski

Literatura

- http://people.sc.fsu.edu/~jburkardt/pdf/mpi_course.pdf (polecam!)
- MPI: A Message-Passing Interface Standard, Version 1.3
- http://aragorn.pb.bialystok.pl/~wkwedlo/PC3.pdf