# Programowanie współbieżne... (4)

**Andrzej Baran 2010/11**

LINK: http://kft.umcs.lublin.pl/baran/prir/index.html

# Przykład

Zaczniemy od znanego już przykładu:

Iloczyn skalarny – różne modele

# Obliczenia sekwencyjne

```
int main(argc,argv)
int argc;
char *argv[];
{
 double sum;
 double a [256], b [256];
 int n;
 n = 256;
 for (i = 0; i < n; i++) {
    a [i] = i * 0.5;
    b [i] = i * 2.0;
 }
 sum = 0;
 for (i = 1; i <= n; i++ ) {
    sum = sum + a[i]*b[i];
 }
 printf ("sum = %f", sum);
}
```

# Obliczenia MPI

```
int main(argc,argv)
int argc;
char *argv[];
{
 double sum, sum_local;
 double a [256], b [256];
 int n, numprocs, myid, my_first, my_last;

 n = 256;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
 my_first = myid *  n/numprocs;
 my_last = (myid + 1) *  n/numprocs;

 for (i = 0; i < n; i++) {
    a [i] = i * 0.5;
    b [i] = i * 2.0;
 }
 sum_local = 0;
 for (i = my_first; i < my_last; i++) {
      sum_local = sum_local + a[i]*b[i];
 }
 MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM,
                                          MPI_COMM_WORLD);
 if (myid==0)  intf ("sum = %f", sum);
}
```

# Obliczenia OpenMP

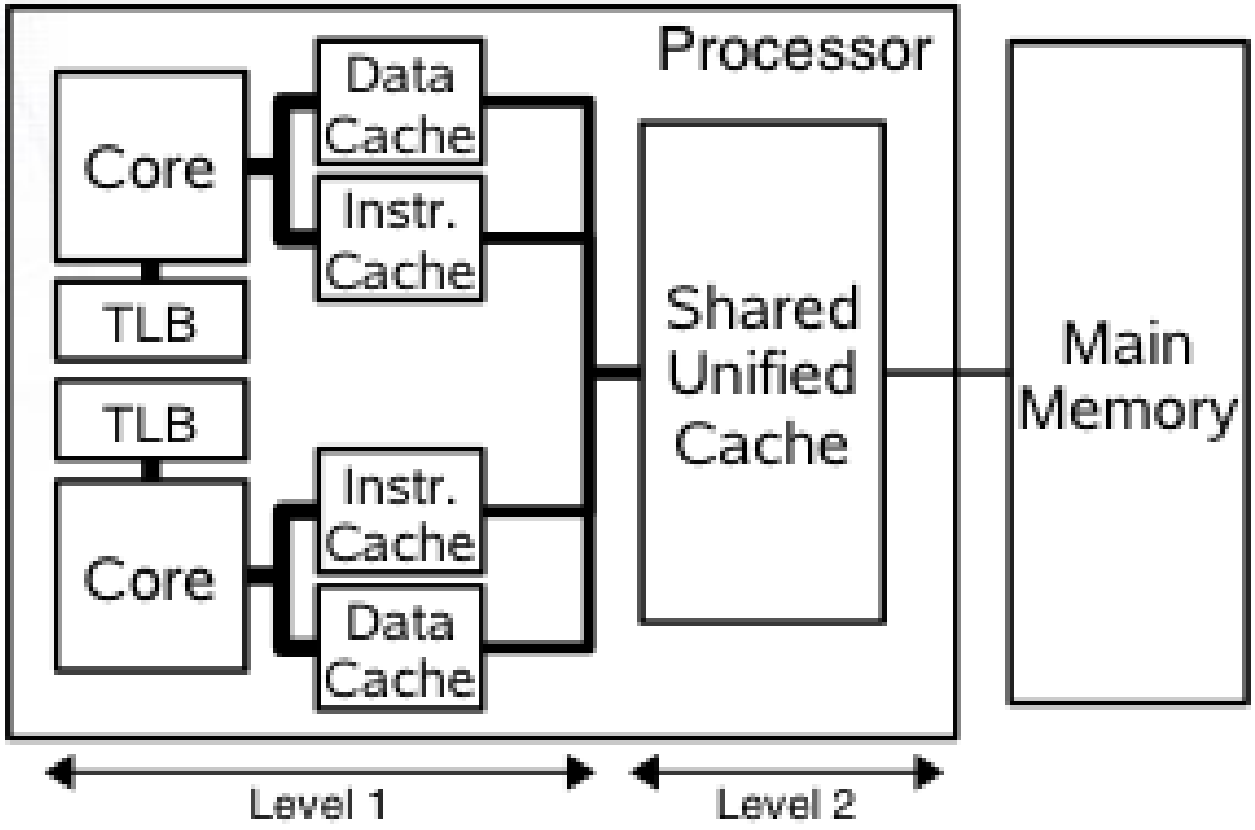
```
int main(argc,argv)
int argc; char *argv[];
{
 double sum;
 double a [256], b [256];
 int status;
 int n=256;

 for (i = 0; i < n; i++) {
    a [i] = i * 0.5;
    b [i] = i * 2.0;
 }

 sum = 0;
 #pragma omp for reduction(+:sum)
 for (i = 1; i <= n; i++ ) {
    sum = sum + a[i]*b[i];
 }
 printf ("sum =  %f \n", sum);
}
```

# Zadanie (Lab)

- Napisać program obliczania iloczynu skalarnego w języku Fortran95 w obu modelach obliczeń równoległych: MPI i OpenMP. Porównać czasy wykonania dla przypadku 1, 2 lub więcej procesów i czasy wykonania z MPI i OpenMP. Wybrać odpowiednio duże wektory a(:) i b(:). Który model daje większe przyspieszenia? Wykreślić zależność przyspieszenia od liczby procesów oraz od wymiaru n wektorów a oraz b.

# Schemat Core2Duo (?)

- Procesor dwurdzeniowy (większość komputerów)



# Historia MPI

- MPI forum: rządy, nauka, przemysł
  - Listopad 1992 utworzenie komitetu do spraw MPI
  - Maj 1994 opublikowano MPI 1.0
  - Czerwiec 1995 MPI 1.1
  - Kwiecień 1995 powstaje komitet MPI 2.0
  - Lipiec 1997 poblikacja MPI 2.0
  - Lipiec 1997 publikacja MPI 1.2

# Przykład OW

OW = obliczenia współbieżne

- Wczytać z pliku tablicę a(1..6), 6 elementową, wykonać obliczenia (dowolne) traktując elementy a() jako dane do obliczeń. Wyniki zapisać do tej samej tablicy i wypisać ją na wyjście (do pliku). Obliczenia wykonać
  - za pomocą 1 procesora
  - za pomocą 3 procesorów

# MPI

- Materiały o MPI http://www.netlib.org/utk/papers/mpi-book/mpi-book.html
- Dokumenty MPICH http://www-unix.mcs.anl.gov/mpi/mpich/
- C, C++ oraz FORTRAN z MPI-1.2 http://www.lam-mpi.org/tutorials/bindings/

- MPI Home

  http://www.mpi-forum.org/

  http://www-unix.mcs.anl.gov/mpi/

# Przegląd

- Message Passing Programming – Przegląd
- Czym jest MPI?
- Programowanie równoległe z MPI
- Podstawy Send i Receive
  - Buforowanie i dostarczanie komunikatów
- Non-blocking communication
- Collective Communication
- Uwagi o innych możliwościach

# Message Passing Programming

- Model
  - Grupa procesow z których  każdy posiada dostęp do danych lokalnych i może sie porozumiewać z innymi procesami wysyłając komunikaty
- Korzyści
  - Wygodny I zupełny model opisu algorytmów równoległych
  - Potencjalnie szybka realizacja algorytmów i obliczeń
  - Często używany

# Czym jest MPI?

- MPI: standard przesyłania komunikatów między procesami
  - Przed standardem MPI istniały “standardy” i biblioteki firmowe (Cray shmem, IBM MPL) i inne (PVM, p4)
- Specyfikacja biblioteki “message-passing” dla Fortranu, C oraz C++

# Stan realizacji - wdrożenia

- MPI 1.2, MPI 2
  - MPICH 2
  - ANL/MSU
    (Argonne National Lab., Michigan State University)
  - LAM/MPI, Indiana University, University of Notre Dame, Ohio State University
  - IBM, Cray, HP, SGI, NEC, Fujitsu

# Programowanie równoległe z MPI

- Komunikacja
  - podstawy send/receive (blocking)
  - Kolektywna
  - Non-blocking
  - Jednostronna (MPI 2)
- Synchronizacja
  - Bezpośrednia w komunikacji  point-to-point
  - Synchronizacja globalna poprzez komunikaty kolektywne
- Równoległe I/O (MPI 2)

# Współbieżność

- Single Program Multiple Data (SPMD)
  - Każdy proces wykonuje ten sam program z innymi danymi
  - Każda kopia pracuje w swoim tempie i nie wymaga synchronizacji
  - Program może realizować różne przebiegi
    - Kontrola odbywa się poprzez parametr *rank* oraz *liczbę zadań*

# Współbieżność

- Multiple Program Multiple Data
  - Każdy proces MPI może być oddzielnym programem
- Połączenie MPI I OpenMP lub pthreads
  - Każdy proces MPI można rozbić na wątki używając np. dyrektyw OpenMP

# PROGRAMY

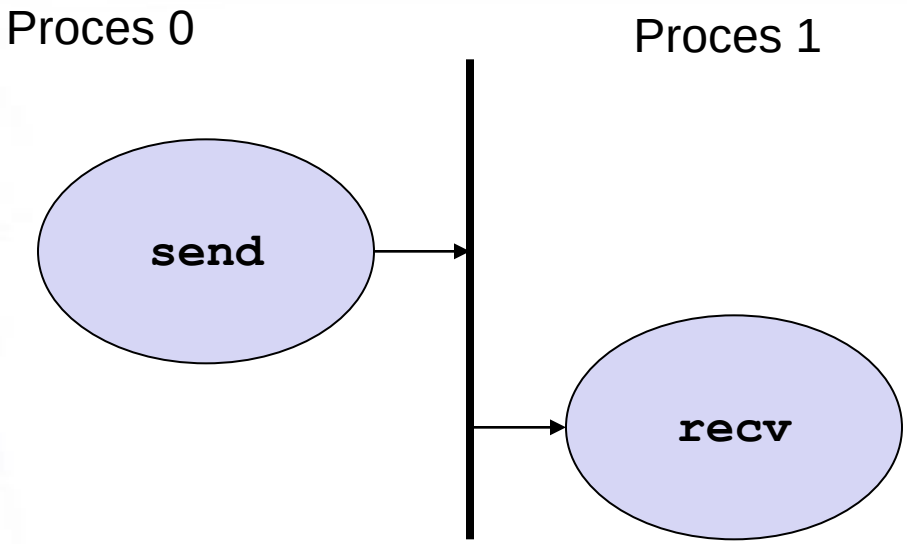
# Hello (C)

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

int main( argc, argv )
int argc;
char *argv[];
{
    int rank, size;
MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

# Hello (Fortran)

```
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )

print *, 'I am ', rank, ' of ', size

 call MPI_FINALIZE( ierr )
end
```

# Podstawa MPI: Send/Receive




- Dla poprawnego przesyłania komunikatów potrzebne są informacje:
  - Jak zidentyfikować proces?
  - Jak zidentyfikować komunikat (message)?
  - Jak zidentyfikować dane?

# Identyfikacja processów

- MPI Communicator (komunikator)
  - Definiuje *grupę* (zbiór uporządkowanych procesów) oraz *context* (sieć wirtualna)
- Rank (ranga)
  - Numer procesu w grupie
  - `MPI_ANY_SOURCE` przyjmuje komunikaty od dowolnych procesów
- Komunikator
  - `MPI_COMM_WORLD` oznacza całą grupę procesów

# Identyfikacja komunikatów

- MPI Communicator definiuje wirtualną sieć, para `send/recv` używa tego samego komunikatora
- Procedury `send/recv` posiadają znacznik (*tag;* zmienna całkowita), którego używa się do identyfikacji komunikatu (wiadomości)
  - `MPI_ANY_TAG` otrzymuje wiadomości z dowolnym znacznikiem (tag)

# Identyfikacja danych

- Dane opisane są trójką (addres, typ, ilość)
  - Dla `send`, definiuje to wiadomość
  - Dla `recv`, definiuje to rozmiar odebranego bufora
- Ilość otrzymanych danych, źródło danych oraz tag są znane poprzez *status* struktury danych
  - Użyteczny jeśli używa się `MPI_ANY_SOURCE`, `MPI_ANY_TAG`

# List ...

KOPERTA

Odbiorca:
Imię i Nazwisko
Ulica, nr domu
Kod, Miasto
Kraj

Nadawca:
Imię i Nazwisko
Ulica, nr domu
Miasto
Kraj

Szanowny Panie,

Bla, bla, bla...

= treść listu

# Komunikat MPI

- Koperta
  - source (źródło)
  - destination (cel)
  - communicator – grupa procesów wśród których jest nadawca i odbiorca
  - tag (znacznik) – uzywany do klasyfikacji komunikatów
- Treść
  - buffer (bufor) – tablica danych
  - count (ilość) – liczba danych
  - dtype (typ) – typ danych

# Blokujące operacje SEND, RECV

- Podstawowa operacja **wysyłania** komunikatu MPI
  - MPI_SEND
- Argumenty
  - buffer, count, datatype = TREŚĆ
  - destination, tag, communicator = KOPERTA
- FORTRAN
  - MPI_SEND(buf, count, dtype, dest, tag, comm, ierr)
- count, dtype, dest, tag, comm – INTEGER
- ierr – INTEGER, kod błędu

# Blokujące operacje SEND, RECV

- Podstawowa operacja **odbioru** komunikatu MPI
  - MPI_RECV
- Argumenty
  - buffer, count, datatype = TREŚĆ
  - source, tag, communicator, status = KOPERTA (odb. niejawny)
- FORTRAN
  - MPI_RECV(buf, count, dtype, source, tag, comm, status, ierr)
- count, dtype, source, tag, comm = INTEGER
- status = integer; tablica o wielkości MPI_STATUS_SIZE

# Uwagi, błędy...

- Niezgodność typów danych
- Niezgodność długości bufora danych
- Buf = tablica, której typ musi być zgodny z typem określonym przez dtype
- inne

# Typy danych MPI

- Typy danych MPI
  - Predefiniowane typy  MPI
  - Typy tablicowe ciągłe
  - Tablice jednakowej długości bloków
  - Tablice bloków różnej długości
  - Struktury dowolne
- Typy definiowane przez użycie odpowiednich procedur MPI, np. `MPI_TYPE_VECTOR`

# Predefiniowane typy MPI

C:

```
MPI_INT
MPI_FLOAT
MPI_DOUBLE
MPI_CHAR
MPI_UNSIGNED
MPI_LONG
```

Fortran:

```
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_CHARACTER
MPI_LOGICAL
MPI_COMPLEX
MPI_REAL8, MPI_REAL16
```

C, Fortran:

```
MPI_BYTE
```

# Przykład komunikacji point-to-point

## Process 0

```
#define TAG 999
float a[10];
int dest=1;
MPI_Send(a, 10,
MPI_FLOAT, dest, TAG,
MPI_COMM_WORLD);
```

## Process 1

```
#define TAG 999
MPI_Status status;
int count;
float b[20];
int sender=0;
MPI_Recv(b, 20,
MPI_FLOAT, sender, TAG,
MPI_COMM_WORLD,
&status);
MPI_Get_count(&status,
MPI_FLOAT, &count);
```

# MPI_Send

```
MPI_Send(address, count, type, dest, tag, comm)
```

- address: wskaźnik do danych (pointer)
- count: liczba wysyłanych elementów
- type: typ danych
- dest: proces odbierający
- tag: identyfikator (tag)
- comm: komunikator

Kiedy `MPI_Send` powraca oznacza to, że komunikat został wysłany (nie musiał być jednak odebrany)

# MPI_Recv

```
MPI_Recv(address, count, type, dest, tag, comm, status)
```

- address: pointer do danych
- count: liczba elementów do przesłania
- type: typ danych
- dest: proces odbiorca
- tag: identyfikator (tag)
- comm: komunikator
- status: nadawca, tag, I rozmiar komunikatu

Kiedy `MPI_Recv` wraca oznacza to, że komunikat został odebrany I można używać otrzymane dane.

# MPI Status Data Structure

- C

```
MPI_Status status;
int recvd_tag, recvd_from, recvd_count;
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, MPI_INT, &recvd_count);
```

- Fortran

```
integer status(MPI_STATUS_SIZE)
```

# Programowanie MPI: 6 procedur

- Pewne programy można napisać używając tylko 6 podstawowych procedur biblioteki MPI:
  - `MPI_Init`
  - `MPI_Finalize`
  - `MPI_Comm_size`
  - `MPI_Comm_rank`
  - `MPI_Send`
  - `MPI_Recv`

# Algorytmy

- Przy „poważnych” obliczeniach warto najpierw zajrzeć do bibliotek gotowych programów
- BLACS, SCALAPACK, ...
- Lub książek...
- Następny krok → programowanie

# Problemy...?