Programowanie współbieżne... (5)

Andrzej Baran 2010/11

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

6 FUNKCJI

- Proste programy MPI można pisać używając tylko 6 funkcji

6 podstawowych funkcji (1)

- MPI_INIT
  Zainicjuj obliczenia MPI

- MPI_FINALIZE
  Zakończ obliczenia MPI

6 podstawowych funkcji (2)

- MPI_COMM_SIZE
  Określ liczbę procesów w komunikatorze

- MPI_COMM_RANK
  Określ identyfikator procesu w danym komunikatorze

6 podstawowych funkcji (3)

- MPI_SEND
  Wyślij wiadomość do innego procesu

- MPI_RECV
  Odbierz wiadomość od innego procesu

Wymiana danych - problemy

Proces 0	Proces 1
`MPI_recv(…,1,…)`	`MPI_recv(…,0,…)`
`MPI_send(…,1,…)`	`MPI_send(…,0,…)`

Zakleszczenie (Deadlock).

`MPI_recv` czeka dopóty, dopóki nie wykona się send

lub odwrotnie: czeka **`MPI_send`**!

Wymiana danych - problemy

Proces 0	Proces 1
`MPI_send(…,1,…)`	`MPI_send(…,0,…)`
`MPI_recv(…,1,…)`	`MPI_recv(…,0,…)`

Może dojść do impasu (zależy od implementacji). Jeśli komunikaty będą buforowane program może działać. (W standardzie MPI nosi to nazwę *unsafe send/recv*)

Non-blocking Communication

- Komunikacja rozdzielona jest na dwie części:
  - **`MPI_Isend`** lub **`MPI_Irecv`** startuje komunikację zwracając `request`
  - **`MPI_Wait`** (również **`MPI_Waitall`**, **`MPI_Waitany`**) używa `request` jako argumentu i blokuje do momentu ukończenia komunikacji
  - **`MPI_Test`** używa `request` jako argumentu i sprawdza kompletność
- Zyski
  - Nie ma sytuacji zakleszczenia
  - Komunikacja i obliczenia przekrywają się (w tym samym czasie)
  - Komunikacja dwukierunkowa

Wymiana danych: Isend/Irecv

Process 0

```
MPI_Isend(…,1,…,
          request)

MPI_Recv(…,1,…)

MPI_Wait(request,
          status)
```

Process 1

```
MPI_Isend(…,0,…,
          request)

MPI_Recv(…,0,…)

MPI_Wait(request,
          status)
```

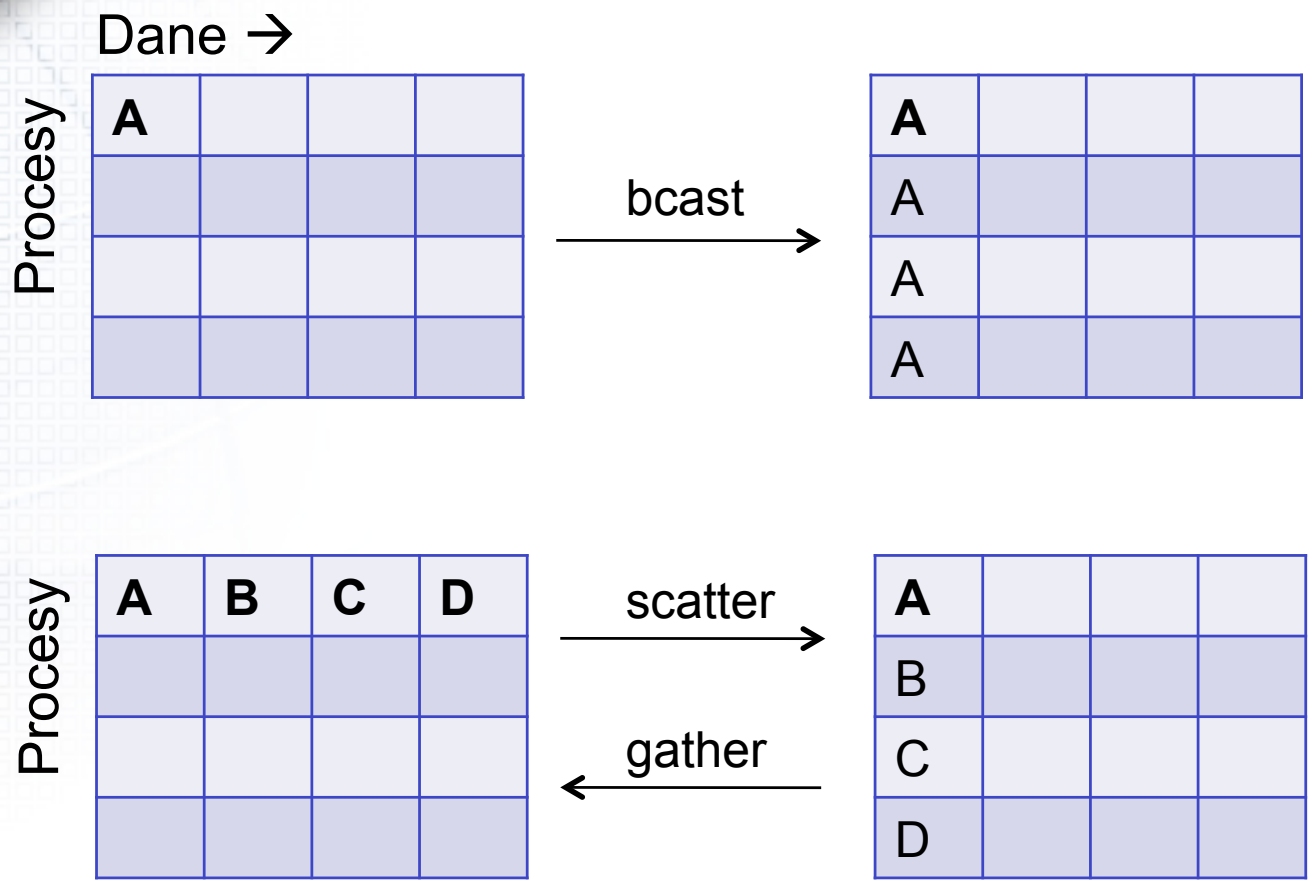
Komunikacja kolektywna

- Operacje kolektywne wywoływane są przez wszystkie procesy w komunikatorze
- **`MPI_BCAST`** rozdziela dane z jednego procesu (root) na wszystkie procesory w komunikatorze
- **`MPI_REDUCE`** zbiera dane od wszystkich procesów w komunikatorze I zwraca do jednego procesu
- W wielu algorytmach numerycznych, **`SEND/RECEIVE`** można zastąpić parą **`BCAST/REDUCE`**, upraszczając program i podnosząc efektywność

Procedury kolektywne

- **`MPI_Bcast(data, count, type, src, comm)`**
  - Rozesłanie danych z **`src`** do wszystkich procesów w komunikatorze.
- **`MPI_Gather(in, count, type, out, count, type, dest, comm)`**
  - Zbieranie danych ze wszystkich węzłów do węzła **`dest`**
- **`MPI_Scatter(in, count, type, out, count, type, src, comm)`**
  - Wysłanie (spec) danych z węzła **`src`** do wszystkich innych

Procedury kolektywne



Procedury kolektywne

- Funkcje dodatkowe
  - `MPI_Allgather`
  - `MPI_Gatherv`
  - `MPI_Scatterv`
  - `MPI_Allgatherv`
  - `MPI_Alltoall`

Procedury redukcji

- `MPI_Reduce(send, recv, count, type, op, root, comm)`
  - Globalna operacja “op” redukcji; wynik znajduje się w buforze *recv* procesu *root*; “op” może być zdefiniowane przez użytkownika (predefiiowane operacje MPI)
- `MPI_Allreduce(send, recv, count, type, op, comm)`
  - Jak wyżej, lecz wynik jest przekazywany do wszystkich procesów komunikatora.

Procedury redukcji

dane

procesy

A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

reduce →

A0#A1# A2#A3	B0+B1# B2#B3	C0#C1# C2#C3	D0#D1# D2#D3

\# jest jednym z operatorów redukcji

A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

allreduce →

A0#A1# A2#A3	B0#B1# B2#B3	C0#C1# C2#C3	D0#D1# D2#D3
A0#A1# A2#A3	B0#B1# B2#B3	C0#C1# C2#C3	D0#D1# D2#D3
A0#A1# A2#A3	B0#B1# B2#B3	C0#C1# C2#C3	D0#D1# D2#D3
A0#A1# A2#A3	B0#B1# B2#B3	C0#C1# C2#C3	D0#D1# D2#D3

Procedury redukcji

- Procedury dodatkowe
  - **`MPI_Reduce_scatter()`, `MPI_Scan()`**
- Operacje predefiniowane
  - **`sum`, `product`, `min`, `max`**, ...
- Operacje definiowane przez użytkownika: patrz procedura MPI
  - **`MPI_Op_create()`**

Komunikacja kolektywna

A			
B			
C			
D			

allgather →

A	**B**	**C**	**D**
A	B	C	D
A	B	C	D
A	B	C	D

A0	**A1**	**A2**	**A3**
B0	B1	B2	B3
C0	C1	C2	C3
D0	D1	D2	D3

alltoall →

A0	**B0**	**C0**	**D0**
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

Komunikacja kolektywna

- operator

A
B
C
D

scan →

A
A#B
A#B#C
A#B#C#D

A0	**A1**	**A2**	**A3**
B0	B1	B2	B3
C0	C1	C2	C3
D0	D1	D2	D3

reduce scatter →

A0#B0#C0#D0
A1#B1#C1#D1
A2#B2#C2#D2
A3#B3#C3#D3

Uwagi o C, Fortran, C++

- W C:
  - `#include mpi.h`
  - Funkcje MPI zwracają kody błędów lub `MPI_SUCCESS`
- W Fortranie
  - `include mpif.h`
  - `use mpi` (MPI 2)
  - Wszystkie procedury MPI wywołuje się jak `subroutine` (`call` ...); kodem powrotu jest ostatni argument procedury
- W C++
  - `Size = MPI::COMM_WORLD.Get_size();` (MPI 2)

Darmowe wdrożenia MPI

- MPICH z Argonne National Lab oraz Mississippi State University
  - http://www-unix.mcs.anl.gov/mpi/mpich/
- Pracuje na:
  - Stacjach roboczych
  - SMP, używających pamięci dzielonych
  - MPP
  - Windows

Darmowe wdrożenia MPI

- LAM z Ohio Supercomputer Center, University of Notre Dame, Indiana University
  - http://www.lam-mpi.org/
- Posiada wiele własności MPI 2
- Działa:
  - W sieci i na stacjach roboczych

Co dalej

- MPI Standard
  - http://www.mpi-forum.org/
- Książki
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface (second edition),* by Gropp, Lusk and Skjellum
  - *Using MPI-2: Advanced Features of the Message-Passing Interface,* by Gropp, Lusk and Thakur
  - *MPI – The Complete Reference. Volume 1*, by Snir, Otto, Huss-Lederman, Walker and Dongarra
  - *MPI – The Complete Reference. Volume 2*, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir

Przykłady

- Macierz x wektor
- Pi
- Cząstki oddziałujące siłami Coulomba

Macierz * wektor

Rozpatrzmy prosty przykład mnożenia macierzy A przez wektor x z użyciem MPI.

Przykład jest nieco sztuczny ale zawiera istotne elementy ważne w programowaniu równoległym.

Chcemy wyliczyć ***A***x* = ***b***.

Założymy, że cała macierz ***A*** i wektor ***x*** znane są dla procesu głównego (root lub master; proces o identyfikatorze 0).

Musimy przesłać odpowiednie dane do innych procesów, które wykonają obliczenia, prześlą je do procesu głównego, a ten odpowiednio je zbierze.

Macierz * wektor

Ponieważ jeden z procesów jest specjalnie wyróżniony i organizuje pracę pozostałym , więc model ten nosi nazwę *master-slave*
(lub zarządca-wykonawca).

Element b(i) jest iloczynem skalarnym i-tego wiersza macierzy ***A*** oraz wektora ***x***

$$b_i = \sum_{j=1}^{N} A_{ij} x_j$$

Jeśli mamy N procesów, to każdy może obliczyć jeden element wektora ***b.***

Jest tylko p procesów i tylko p-1 z tej liczby (bez procesu głównego) wykonuje obliczenia. Musimy wykonać zadanie na raty.

Macierz * wektor

Wysyłamy kopie wektora ***x*** do każdego procesu.

Następnie wysyłamy *i*-ty wiersz macierzy ***A*** do procesu *i*.

Jeśli proces i-ty zwróci b(i) wówczas wysyłamy mu następny wolny wiersz macierzy ***A*** .

Sposób w jaki to robimy pozwala na wykonanie obliczeń w dowolnej kolejności - jest więc bardzo elastyczny.

W wyniku, proces główny nie musi wiedzieć, który z procesów wysyła odpowiedź. Wystarczy, że wie jakie dane są przesyłane i kiedy obliczenia są skończone.

Macierz * wektor: pseudokod

```
(wg. Burkardt, wykład o MPI)
If jestem mistrzem:
   SEND N to all workers.
   SEND X to all workers.
   SEND out first batch of rows.
   While ( elementy wektora B nie zostały zwrócone)
      RECEIVE komunikat, element ? wekt. B, od
                        procesu ?
      If ( jakiś wiersz tablicy A nie został wysłany)
         SEND wiersz ? tablicy A do procesu ?
      else
      SEND komunikat "FINALIZE" do procesu ?
   end
end
FINALIZE
```

Macierz * wektor: pseudokod

```
else jwśli jestem robotnikiem:
    RECEIVE N
        RECEIVE X
        do
            RECEIVE komunikat
            if ( komunikat brzmi "FINALIZE" ) then
                FINALIZE
            else
                jest jeszcze wiersz A,
                a więc wylicz jego iloczyn skalarny z X
                SEND wynik do mistrza
            end
        end
end
```

Broadcast: rozgłaszenie

W wielu wypadkach procesy nie muszą porozumiewać sie każdy z każdym. W sytuacji, którą się zajmujemy jeden z procesów porozumiewa się z resztą, informując je w jakiś sposób.

Wybrany proces (zero) porozumiewa się również z użytkownikiem, zajmuje się wejściem/wyjściem, zbiera dane od innych procesów, dokonuje ich redukcji, itd.

W MPI istnieje funkcja (procedura) *rozgłaszania* (broadcast), która pozwala procesowi głównemu przesłać informacje pozostałym procesom.

W tym wypadku ta sama procedura służy do przesyłania i odbierania informacji.

...algorytm

Obliczanie ***A x = b.***

- zadaniem jest pomnożenie wiersza macierzy A przez x;
- możemy przypisać jedno zadanie każdemu z procesorów. Jeśli proces wykona zadanie przydzielamy mu zadanie następne
- każdy z procesów potrzebuje kopii x za każdym razem; do wykonania nowego zadania proces potrzebuje odpowiedniego wiersza A.
- proces 0 nie będzie wykonywał zadań; zamiast tego będzie je przydzielać i będzie zbierać wyniki obliczeń od innych procesów.

MPI_BCAST()

MPI_BCAST (data, count, type, from, communicator)

- **data**, addres danych;
- **count**, liczba elementów w danych;
- **type**, typ danych;
- **from**, ID procesu, który wysyła dane;
- **communicator,** komunikator;

Macierz * wektor (szkic pgm)

```
if ( myid == master )
   numsent = 0
   !
   ! BROADCAST X to  all the workers.
   !
   call MPI BCAST(x, cols, MPI_DOUBLE_PRECISION, master, &
                           MPI_COMM_WORLD, ierr )
   !
   ! SEND row i to worker process i;
   !      tag the message wi th the row number.
   !
   do i = 1, min(num_procs-1, rows)
      do j = 1, cols
         buffer(j) = a(i, j)
      end do
      call MPI SEND(buffer, cols, MPI_DOUBLE_PRECISION, i,  &
                                  MPI_COMM_WORLD, ierr )
      numsent = numsent + 1
   end do
```

Macierz * wektor (szkic pgm)

```
! Wait to receive a result back from any processor ;
! If more rows to do, send the next one back to that processor.
!
do i = 1 , rows
   call MPI_RECV ( ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,&
                   MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
   sender = status(MPI_SOURCE)
   anstype = status(MPI_TAG)
   b(anstype) = ans

   if (numsent .lt. rows) then
      numsent = numsent + 1

      do j = 1, cols
         buffer(j) = a( numsent , j )
      end do
      call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, &
                    sender, numsent, MPI_COMM_WORLD, ierr)
   else
      call MPI_SEND ( MPI_BOTTOM, 0, MPI_DOUBLE PRECISION,
                      sender, 0, MPI_COMM_WORLD, ierr)
   end if
end do ! i
```

Macierz * wektor (szkic pgm)

```
!
! Workers receive X, then compute dot products until
! done message received
!
else
   call MPI_BCAST(x, cols, MPI_DOUBLE_PRECISION, master, &
                           MPI_COMM_WORLD, ierr)
90 continue
   call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master, &
                 MPI_ANY_TAG, MPI COMM WORLD, status, ierr)
   if (status (MPI_TAG) .eq. 0) then
      go to 200
   end if

   row = status(MPI_TAG)
   ans = 0.0
   do i = 1 , cols
      ans = ans + buffer(i) *x(i)
   end do

   call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, row, &
                         MPI_COMM_WORLD, ierr )
   go to 90
200 continue
end if
```

Przykład. Obliczanie π

Wyliczyć wartość π według formuły (pokazać najpierw, że formuła jest słuszna):

$$\pi = \int_0^1 \frac{4}{1 + x^2} dx$$

Przykład. π w C (1)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done)  {
      if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
      }
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (n == 0) break;
```

Przykład. π w C (2)

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is .16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

Ćwiczenie

- Zmień program Hello tak, by każdy proces wysłał do procesu 0 nazwę maszyny, na której jest wykonywany i by ten ją wydrukował.
  - Znajdź w dokumentacji MPI i zapoznaj się z procedurą: `MPI_Get_processor_name()`
- Wykonaj to tak, by procesy drukowały informacje zgodnie z rangą (rank)

Oddziaływanie kulombowskie

- Wyliczyć energię układu cząstek oddziałujących potencjałem Coulomba

```
real coord(3,n), charge(n)

energy=0.0
do i = 1, n
  do j = 1, i-1
    rdist = 1.0/sqrt((coord(1,i)-coord(1,j))**2+
      (coord(2,i)-coord(2,j))**2+(coord(3,i)-coord(3,j))**2)
    energy = energy + charge(i)*charge(j)*rdist
  end do
end do
```

- Ddekompozycja funkcjonalna
  - Każdy proces powinien obliczać taką samą w przybliżeniu liczbę oddziaływań
  - Aby to osiągnąć dzielimy zewnętrzną pętlę (do)
  - W celu uproszczenia komunikacji powielamy dane

```
include 'mpif.h'
      parameter(n=50000)
      dimension coord(3,n), charge(n)

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD, mype, ierr)
      call mpi_comm_size(MPI_COMM_WORLD, npes, ierr)

      call initdata(n,coord,charge,mype)

      e = energy(mype,npes,n,coord,charge)

      etotal=0.0
      call mpi_reduce(e, etotal, 1, MPI_REAL, MPI_SUM, 0,
        MPI_COMM_WORLD, ierr)
      if (mype.eq.0) write(*,*) etotal

      call mpi_finalize(ierr)
```

```
subroutine initdata(n,coord,charge,mype)
   include 'mpif.h'
   dimension coord(3,n), charge(n)

   if (mype.eq.0) then
   !   GENERATE coords, charge
   end if

   ! broadcast data to slaves

   call mpi_bcast(coord, 3*n, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
   call mpi_bcast(charge, n, MPI_REAL, 0, MPI_COMM_WORLD, ierr)

   return
```

```
real function energy(mype,npes,n,coord,charge)
    dimension coord(3,n), charge(n)

    inter=n*(n-1)/npes
    nstart=nint(sqrt(real(mype*inter)))+1
    nfinish=nint(sqrt(real((mype+1)*inter)))
    if (mype.eq.npes-1) nfinish=n

    total = 0.0
    do i = nstart, nfinish
      do j = 1, i-1
        ....
        total = total + charge(i)*charge(j)*rdist
      end do
    end do
    energy = total
    return
```

MPI - przykład cd

- Dekompozycja dziedziny
  - Każdy proces otrzymuje zbiór cząstek
  - Proces wylicza oddziaływania cząstek własnych i oddziaływania z innymi cząstkami na podstawie danych otrzymanych od innych procesów; sam też wysyła informacje
  - Wszystko się powtarza do momentu aż wyliczy się wszystkie oddziaływania

```
subroutine initdata(n,coord,charge,mype,npes,npepmax,nmax,nmin)
      include 'mpif.h'
      dimension coord(3,n), charge(n)
      integer status(MPI_STATUS_SIZE)
      itag=0
      isender=0
      if (mype.eq.0) then
         do ipe=1,npes-1
             ! Utwórz coord, charge dla PE=ipe
             call mpi_send(coord, nj*3, MPI_REAL, ipe, itag,
             MPI_COMM_WORLD, ierror)
             call mpi_send(charge, nj, MPI_REAL, ipe, itag,
             MPI_COMM_WORLD, ierror)
         end do
         ! Utwórz coord, charge dla siebie
      else   ! receive particles
          call mpi_recv(coord, 3*n, MPI_REAL, isender, itag,
          MPI_COMM_WORLD, status, ierror)
          call mpi_recv(charge, n, MPI_REAL, isender, itag,
          MPI_COMM_WORLD, status, ierror)
      endif
return
```

```
niter=npes/2
   do iter=1, niter

      ! PE do wysłania i odbioru
      if (ipsend.eq.npes-1) then
         ipsend=0
      else
         ipsend=ipsend+1
      end if
      if (iprecv.eq.0) then
         iprecv=npes-1
      else
         iprecv=iprecv-1
      end if

      ! Wyślij i odbierz cząstki
      call mpi_sendrecv(coordi, 3*n, MPI_REAL, ipsend, &
            itag, coordj, 3*n, MPI_REAL, iprecv, itag, &
                        MPI_COMM_WORLD, status, ierror)
```

```
call mpi_sendrecv( &
          chargei, n, MPI_REAL, ipsend, itag, &
          chargej, n, MPI_REAL, iprecv, itag, &
          MPI_COMM_WORLD, status, ierror)

! Sumowanie energii
          e = e+energy2(n,coordi,chargei,n,coordj,chargej)

       end do
```

LITERATURA

- Gropp, **Using MPI**;
- Mascani, Srinavasan, **Algorithm 806*:* SPRNG*:* a scalable library for pseudorandom number generation**, ACM Transactions on Mathematical Software**;**
- Openshaw, **High Performance Computing**;
- Pacheco, **Parallel Programming with MPI** ;
- Petersen, **Introduction to Parallel Computing**;
- Quinn**, Parallel Programming in C with MPI and**
- **OpenMP**;
- Snir, MPI: **The Complete Reference**;
- Burkardt: **FDI Summer Track V: Parallel Programming, Using MPI**, 2008**.**

Problemy...?