Distributed Memory Programming with MPI

ISC5316-01: Applied Computational Science II

....

John Burkardt
Department of Scientific Computing
Florida State University

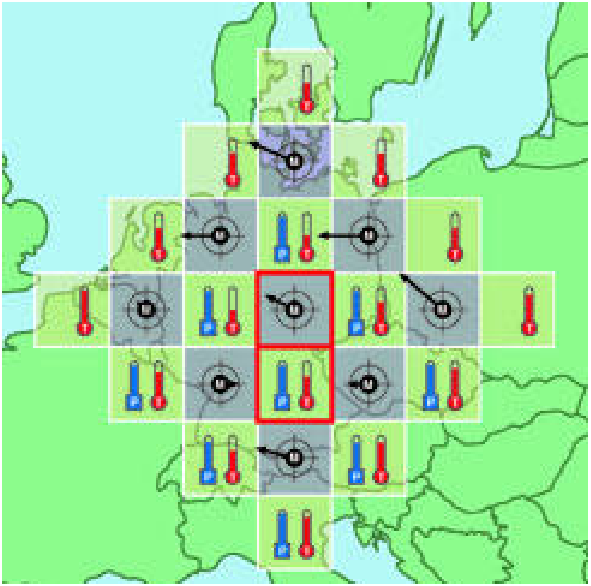
http://people.sc.fsu.edu/∼jburkardt/presentations/...
...mpi_2013_acs2.pdf

10/15 October 2013

Distributed Memory Programming with MPI

INTRO: We Work on Problems that are Too Big

Richardson tries to predict the weather.



INTRO: Big Problems Are Worth Doing

In 1917, Richardson's first efforts to compute a weather prediction were simplistic and mysteriously inaccurate.

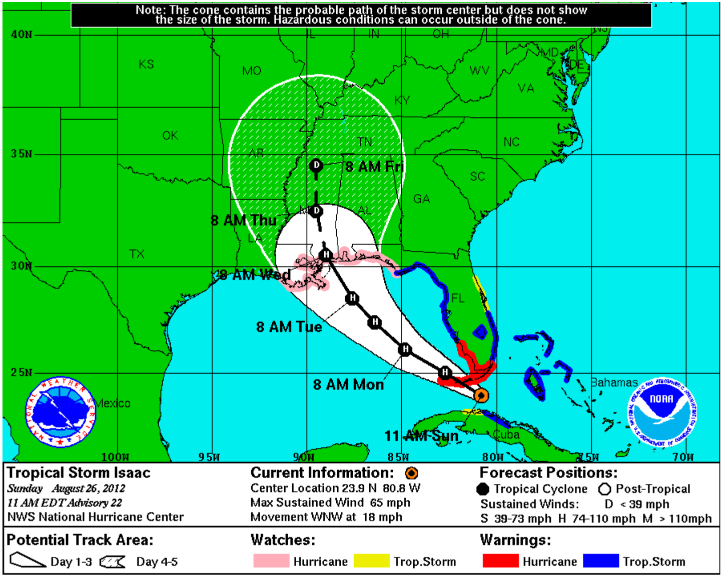
But he believed that with better algorithms and more data, it would be possible to predict the weather reliably.

Over time, he was proved right, and the prediction of weather became one of the classic computational problems.

Soon there was so much data that making a prediction 24 hours in advance could take...24 hours of computer time.

Even now, accurate prediction of weather is only possible in the short range. In August 2012, the predicted path of Hurricane Isaac was first going to be near Tampa, but then the predictions gradually shifted westward to New Orleans, giving that city only a few days warning.

INTRO: Big Problems Can Be Solved



INTRO: For Years, Speeding Up the Clock was Enough

When computer clock speeds stopped improving, computer architects explored other ways that hardware could be used to produce results faster, using parallel programming.

OpenMP takes advantage of the fact that a single computer chip now contains a big blob of memory, and many cores. The cores could be running independent programs, each with separate memory. But OpenMP runs a single program, with one core in charge, and the others helping out on loops, working on a shared memory.

MPI takes advantage of the dramatic improvement in communication speed between separate computers. It assumes that programs are running on each computer, and figures out how to quickly send small set of information between the programs that allow a single calculation to be distributed over separate computers.

INTRO: Networking Has Speeded Up

Inter-computer communication has gotten faster and cheaper.

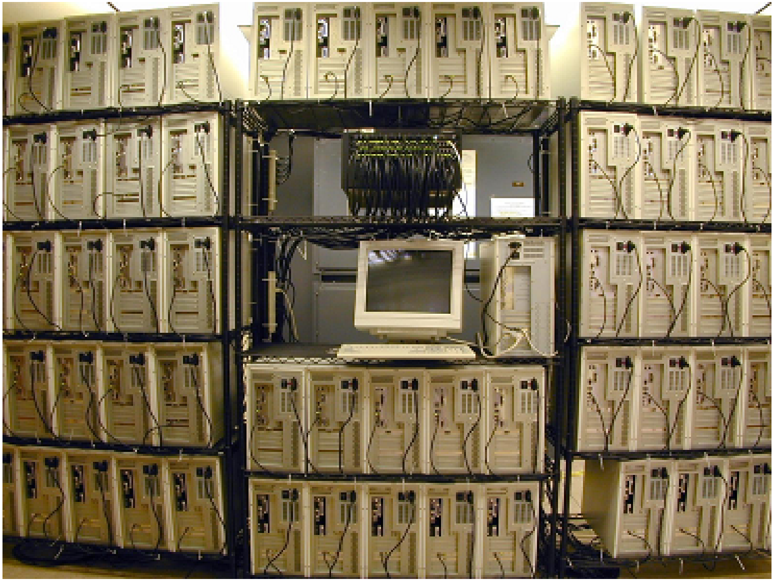
It seemed possible to imagine that an “orchestra” of low-cost machines could work together and outperform supercomputers, in speed and cost.

That is, each computer could work on part of the problem, and occasionally send data to others. At the end, one computer could gather up the results.

If this was true, then the quest for speed would simply require connecting more machines.

But where was the conductor for this orchestra?
And who would write the score?

INTRO: Early Clusters Were Ugly, but Worked



INTRO: Cluster Computing Can Use MPI

MPI (the Message Passing Interface) manages a parallel computation on a distributed memory system.

The user arranges an algorithm so that pieces of work can be carried out as simultaneous but separate processes, and expresses this in a C or FORTRAN program that includes calls to MPI functions.

At runtime, MPI:

- distributes a copy of the program to each processor;
- assigns each process a distinct ID;
- synchronizes the start of the programs;
- transfers *messages* between the processes;
- manages an orderly shutdown of the programs at the end.

INTRO: Compilation of an MPI Program

Suppose a user has written a C program *myprog.c* that includes the MPI calls necessary for it to run in parallel on a cluster.

The program invokes a new include file, accesses some newly defined symbolic values, and calls a few functions... but it is still just a recognizable and compilable C program.

The program must be compiled and loaded into an executable program. This is usually done on a special *compile node* of the cluster, which is available for just this kind of interactive use.

```
  mpicc -o myprog myprog.c
```

A command like **mpicc** is a customized call to the regular compiler (**gcc** or **icc**, for instance) which adds information about MPI include files and libraries.

INTRO: Interactive Execution

On some systems, the user's executable program can be run interactively, with the **mpirun** command:

```
  mpirun -np 4 ./myprog
```

Here, we request that 4 processors be used in the execution.

What happens next depends on how MPI has been configured.

INTRO: Interactive Execution on One Machine

If the machine where you are working has multiple cores, then it's actually possible for MPI to run a separate program on each core, all locally on that one machine.

In this case, it is trivial for these programs to be synchronized (started at the same time) and to communicate.

However, of course, the number of MPI processes that can be run in this way is strictly limited by the number of cores you have. And on your laptop or desktop, that won't be many.

On the other hand, it's a very nice way to test drive MPI before going to an unfriendly, far away cluster.

This is how we will do almost all our MPI experiments in the lab session!

INTRO: Interactive Execution on Multiple Machines

The other possibility for interactive use is that your machine can reach other machines, that MPI is set up on those machines as well, that these machines all have the same architecture (so a program compiled on one machine will run on any of them) and that there is fast communication among the machines.

If MPI is configured in the right way, then a request like

```
mpirun -np 4 ./myprog
```

will cause your compiled program to be copied to the other machines on the list. Then MPI will "magically" start all the programs at the same time, allow them to communicate, and manage a clean shutdown.

(I am leaving out one detail, which is that usually, your **mpirun** *command will include a list of the machines that you want to try to use for your run.)*

INTRO: Indirect Execution on Multiple Machines

Finally, when people are serious about running a lot of processes in parallel with MPI, then they end up having to work on a cluster.

The user compiles the program on the head node or compile node of the cluster, but then has to write a special script file, which requests that the parallel job be run on “so many” cores.

This request is then submitted to a scheduling program, which (after a minute, an hour, or a day) finds enough machines available to run your program, returning the results to you on completion.

Our local Research Computing Center (RCC) cluster works this way, and you can easily get an account there.

INTRO: Compare OpenMP and MPI

If you are familiar with OpenMP, then it may help to point out some important differences between these two parallel programming systems:

In OpenMP, there is a shared memory, which is usually a single processor. One program is running, but it can call on helper processes or “threads” to carry out parallel loop operations.

In MPI, separate programs are running. They may all be running on a single processor, but they are more likely on separate computers. In any case, they do not share memory. If they need to communicate, they must send a message.

We will see that MPI requires more programming effort than OpenMP, but is much more powerful. We’ll start our MPI exploration with a simple problem.

Distributed Memory Programming with MPI

EXAMPLE:

Suppose we want to approximate an integral of the form $\int_{[0,1]^m} f(x)dx$ by a Monte Carlo procedure.

Perhaps we want to do this in such a way that we have confidence that our error is less than some tolerance ϵ.

Calling the random number generator m times gives us a random sample argument $x^i = (x_1^i, x_2^i, ..., x_m^i)$. We evaluate $f(x^i)$ and add that to a running total. When we've generated n such samples, our integral estimate is simply $\sum_{i=1}^n f(x^i)/n$.

A good error estimate comes by summing the squares and subtracting the square of the sum, and this allows us to decide when to stop.

EXAMPLE:

While this simple process converges inevitably, it can converge very slowly. The rate of convergence depends only on n, but the “distance” we have to converge, the constant in front of the rate of convergence, can be very large, especially as the spatial dimension m increases.

Since there are interesting and important problems with dimensions in the hundreds, it’s natural to look for help from parallel processing.

Estimating an integral is a perfect problem for parallel processing. We will look at a simple version of the problem, and show how we can solve it in parallel using MPI, the Message Passing Interface.

EXAMPLE:

For a specific example, let's try to approximate:

$$I(f) = \int_{[0,1]^2} |4x-2| * |4y-2| \, dx \, dy$$

The exact value of this integral is 1.

If we wish to consider an m dimensional problem, we have

$$I(f) = \int_{[0,1]^m} \prod_{i=1}^{m} |4x_i - 2| dx_1 dx_2 ... dx_m$$

for which the exact value is also 1.

EXAMPLE: C Version

```c
  f1 = 0.0;
  f2 = 0.0;
  for ( i = 0; i < n; i++)
  {
    for ( j = 0; j < m; j++ )
    {
      x[j] = (double) rand () / (double) RAND_MAX;
    }
    value  = f ( m, x );
    f1 = f1 + value;
    f2 = f2 + value * value;
  }
  f1 = f1 / (double) n;
  f2 = f2 / (double) ( n - 1 );
  stdev = sqrt ( f2 - f1 * f1 );
  sterr = stdev / sqrt ( (double) n );
  error = fabs ( f1 - 1.0 );
```

http://people.sc.fsu.edu/∼jburkardt/latex/acs2_mpi_2013/myprog_seq.c

EXAMPLE: F90 Version

```
  f1 = 0.0
  f2 = 0.0
  do i = 1, n
    call random_number ( harvest = x(1:m) )
    value = f ( m, x )
    f1 = f1 + value
    f2 = f2 + value * value
  end do  f1 = f1 / (double) n;
  f1 = f1 / dble ( n )
  f2 = f2 / dble ( n - 1 )
  stdev = sqrt ( f2 - f1 * f1 )
  sterr = stdev / sqrt ( dble ( n ) )
  error = abs ( f1 - 1.0D+00 )
```

http://people.sc.fsu.edu/∼jburkardt/latex/acs2_mpi_2013/myprog_seq.f90

EXAMPLE: Output for M = 2

```
      N        F1       stdev   sterr   error

      1  1.2329416      inf     inf  0.232942
     10  0.7625974  0.9751  0.3084  0.237403
    100  1.0609715  0.8748  0.0875  0.060971
   1000  1.0037517  0.8818  0.0279  0.003751
  10000  0.9969711  0.8703  0.0087  0.003028
 100000  0.9974288  0.8787  0.0028  0.002571
1000000  1.0005395  0.8824  0.0009  0.000539
```

We can only print the **error** because we already know the answer. If we didn't know the answer, what other information suggests how well we are doing?

EXAMPLE: Output for M = 10

```
      N         F1       stdev    sterr    error

      1   0.0643729      inf      inf   0.935627
     10   1.1999289   2.4393   0.7714   0.199929
    100   1.2155225   6.2188   0.6219   0.215523
   1000   1.2706223   6.2971   0.1991   0.270622
  10000   0.9958461   4.4049   0.0440   0.004153
 100000   1.0016405   4.3104   0.0136   0.001640
1000000   1.0056709   4.1007   0.0041   0.005670
```

The standard deviation and error are significantly larger at 1,000,000 samples than for the M=2 case.

EXAMPLE: Output for M = 20

```
      N         F1        stdev    sterr    error

      1  0.0055534       inf      inf    0.99444
     10  0.3171449  0.9767    0.3089   0.68285
    100  0.2272844  0.9545    0.0954   0.77271
   1000  1.7362339  17.6923   0.5595   0.73623
  10000  0.7468981  7.7458    0.0775   0.25310
 100000  1.0327975  17.8886   0.0566   0.03279
1000000  0.9951882  16.5772   0.0166   0.00481
```

The standard deviation and error continue to rise for the M = 20 case

EXAMPLE: Output for M $= 20$

The Monte Carlo method will converge "eventually".

If we have a fixed error tolerance in mind, and (of course) we don't know our actual correct answer, then we look at the standard error as an estimate of our accuracy.

Although 1,000,000 points was good enough in a 2 dimensional space, the standard error is warning us that we need more data in higher dimensions.

If we want to work on high dimensional problems, we will be desperate to find ways to speed up these calculations!

Distributed Memory Programming with MPI

PARALLEL: Is Sequential Execution Necessary?

Now let's ask ourselves a peculiar question:

We used a serial or sequential computer. The final integral estimate required computing 1,000,000 random x and y coordinates, 1,000,000 function evaluations, and 1,000,000 additions, plus a little more (dividing by N at the end, for instance).

The computation was done in steps, and in every step, we computed exactly one number, random x, random y, f(x,y), adding to the sum...

A sequential computer only does one thing at a time. But did our computation actually <u>require</u> this?

PARALLEL: Is Sequential Execution Necessary?

Look at the computation of F1, our approximate integral:

F1 = (f(x1,y1) + f(x2,y2) + ... + f(xn,yn)) / n

We have to divide by n at the end.

The sum was computed from left to right, but we didn't have to do it that way. The sum can be computed in any order.

To evaluate f(x1,y1), we had to generate a random point (x1,y1).

Does the next evaluation, of f(x2,y2) have to come later? Not really! It could be done at the same time.

PARALLEL: Is Sequential Execution Necessary?

So a picture of the logical **priority** of our operations is:

```
  x1 y1    x2 y2  .......   xn yn     <--Generate
 f(x1,y1) f(x2,y2) ....... f(xn,yn)    <--F()
 f(x1,y1)+f(x2,y2)+.......+f(xn,yn)    <--Add
(f(x1,y1)+f(x2,y2)+.......+f(xn,yn))/n <--Average
```

So we have about $2 * n + m * n + n + 1$ operations, so for our example, an answer would take about 5,000,000 "steps".

But if we had n cooperating processors, generation takes 1 step, function evaluation $m = 2$ steps, addition $log(n) \approx 20$ steps, and averaging 1, for a total of 25 steps.

And if we only have k processors, we still run k times faster, because almost all the work can be done in parallel.

PARALLEL: What Tools Do We Need?

To take advantage of parallelism we need:

- multiple cheap processors
- communication between processors
- synchronization between processors
- a programming language that allows us to express which processor does which task;

And in fact, we have multicore computers and computer clusters, high speed communication switches, and MPI.

PARALLEL: Is Sequential Execution Necessary?

The FSU Research Computing Center (RCC) facility has a High Performance Computing (HPC) cluster with 6,464 cores, using the high speed Infiniband communication protocol.

User programs written in MPI can be placed on the “head node” of the cluster.

The user asks for the program to be run on a certain number of cores; a scheduling program locates the necessary cores, copies the user program to all the cores. The programs start up, communicate, and the final results are collected back to the head node.

PARALLEL: What Must the User Do?

Someone wishing to run a problem in parallel can take an existing program (perhaps written in C or C++), and add calls to MPI functions that divide the problem up among multiple processes, while collecting the results at the end.

Because the Monte Carlo integration example has a very simple structure, making an MPI version is relatively simple.

Distributed Memory Programming with MPI

MPI: Logical outline of computation

We want to have one program be in charge, the *master*. We assume there are K worker programs available, and that the master can communicate with the workers, sending or receiving numeric data.

Our program outline is:

- The master chooses the value N, the number of samples.
- The master asks the K workers to compute N/K samples.
- Each worker sums N/K values of $f(x)$.
- Each worker sends the result to the master.
- The master averages the sums, reports the result.

MPI: The Master Program

```
SEND value of n/k to all workers
RECEIVE f1_part and f2_part from each worker
f1 = 0.0;
f2 = 0.0;
for ( j = 0; j < k; j++ )
{
  f1 = f1 + f1_part[j];
  f2 = f2 + f2_part[j];
}
f1 = f1 / (double) n;
f2 = f2 / (double) ( n - 1 );
stdev = sqrt ( f2 - f1 * f1 );
sterr = stdev / sqrt ( (double) n );
```

MPI: The Worker Program

```
RECEIVE value of n/k from master
f1_part = 0.0;
f2_part = 0.0;
for ( i = 0; i < n/k; i++)
{
  for ( j = 0; j < m; j++ )
  {
    x[j] = (double) rand () / (double) RAND_MAX;
  }
  value  = f ( m, x );
  f1_part = f1_part + value;
  f2_part = f2_part + value * value;
}
SEND f1_part and f2_part to master.
```

MPI: Parallelism

Communication overhead: If we have K workers, the master needs to send one number N/K to all workers. The master needs to receive $2 * K$ real numbers from the workers.

Computational Speedup: The sampling computation, which originally took N steps, should now run as fast as a single computation of N/P steps.

$$\begin{aligned} \text{Old time} &\approx N * \text{one sample} \\ \text{New time} &\approx (N/K) * \text{one sample} + (3 * K) * \text{communications.} \end{aligned}$$

MPI: How Many Programs Do You Write?

So, to use MPI with one master and 4 workers, we write 5 programs, put each on a separate computer, start them at the same time, and hope they can talk ...right?

It's not as bad. Your computer cluster software copies your information to each processor, sets up communication, and starts them. We'll get to that soon.

It's much more surprising that you don't a separate program for each worker. That's good, because otherwise how do people run on hundreds of processors? In fact, it's a little bit amazing, because you only have to write one program!

MPI: One Program Runs Everything

The secret that allows one program to be the master and all the workers is simple. If we start five copies of the program, each copy is given a unique identifier of 0, 1, 2, 3 and 4.

The program can the decide that whoever has ID 0 is the master, and should carry out the master's tasks. The programs with ID's 1 through 4 can be given the worker tasks.

That way, we can write a single program. It will be a little bit complicated, looking something like the following example.

MPI: One Program Runs Everything

```
everyone does this line;
if ( id == 0 )
{
  only the master does lines in here.
}
else
{
  each worker does these lines.
}
something that everyone does.
if ( id == 0 )
{
  only the master does this.
}
if ( id == 2 )
{
  only worker 2 does this.
}
```

MPI: Communication Waits

The other thing to realize about MPI is that the programs start at the same time, but run independently...until a program reaches a communication point.

If a program reaches a RECEIVE statement, it expects a message, (a piece of data), from another program. It cannot move to the next computation until that data arrives.

Similarly, if a program reaches a SEND statement, it sends a message to one or more other programs, and cannot continue until it confirms the message was received (or at least was placed in a buffer).

Programs with a lot of communication, or badly arranged communication, suffer a heavy penalty because of idle time!

Distributed Memory Programming with MPI

CODE: Accessing the MPI Include File

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

# include "mpi.h"     <-- Necessary MPI definitions

double f ( int m, double x[] );

int main ( int argc, char *argv[] )
{
  ...
  return 0;
}
double f ( int m, double x[] )
{
  ...
  return value;
}
```

CODE: Initialize and Terminate MPI

```
int main ( int argc, char *argv[] )
{
  MPI_Init ( &argc, &argv );
  ...
   <-- Now the program can access MPI,
       and communicate with other processes.
  ...
  MPI_Finalize ( );
  return 0;
}
```

CODE: Get Process ID and Number of Processes

```
int main ( int argc, char *argv[] )
{
  int id, p;
  ...
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &id ); <-- id
  MPI_Comm_size ( MPI_COMM_WORLD, &p );  <-- count
  ...
    <-- Where the MPI action will occur.
  ...
  MPI_Finalize ( );
  return 0;
}
```

CODE: Master sends N/(P-1) to each Worker

```
//  The master sets NP = N/(P-1).
//  Each worker will do NP steps.
//  N is adjusted so it equals NP * ( P - 1 ).
//
  if ( id == 0 )
  {
    n = 1000000;
    np = n / ( p - 1 );
    n = ( p - 1 ) * np;
  }
//
//  The Broadcast command sends the value NP
//  from the master to all workers.
//
  MPI_Bcast ( &np, 1, MPI_INT, 0, MPI_COMM_WORLD );
  ...(more)...
  return 0;
}
```

CODE: Rules for MPI_Bcast (C version)

error = MPI_Bcast (data, count, type, from, communicator);

- Sender input/Receiver output, **data**, the address of data;
- Input, int **count**, number of data items;
- Input, **type**, the data type, such as **MPI_INT**;
- Input, int **from**, the process ID which sends the data;
- Input, **communicator**, usually **MPI_COMM_WORLD**;
- Output, int **error**, is 1 if an error occurred.

The values in the **data** array on process **from** are copied into the **data** arrays on all other processes, overwriting the current values (if any).

CODE: Rules for MPI_Bcast (F90 version)

call MPI_Bcast (data, count, type, from, communicator, error);

- Sender input/Receiver output, **data**, the data value or vector;
- Input, integer **count**, number of data items;
- Input, **type**, the data type, such as **MPI_INT**;
- Input, integer **from**, the process ID which sends the data;
- Input, **communicator**, usually **MPI_COMM_WORLD**;
- Output, integer **error**, is 1 if an error occurred.

The values in the **data** array on process **from** are copied into the **data** arrays on all other processes, overwriting the current values (if any).

CODE: Examples of MPI_Bcast

```
MPI_Bcast ( &np, 1, MPI_INT, 0, MPI_COMM_WORLD );
sends the integer stored in the scalar np
from process 0 to all processes.

MPI_Bcast ( a, 2, MPI_FLOAT, 7, MPI_COMM_WORLD );
sends the first 2 floats stored in the array a
(a[0] and a[1]) from process 7 to all processes.

MPI_Bcast ( x, 100, MPI_DOUBLE, 1, MPI_COMM_WORLD );
sends the first 100 doubles stored in the array x
(x[0] through x[99]) from process 1 to all processes.
```

CODE: The Workers Work

```
f1_part = 0.0;     <-- Even the master does this!
f2_part = 0.0;
if ( 0 < id )
{
  seed = 12345 + id;  <-- What's going on here?
  srand ( seed );
  for ( i = 0; i < np; i++)
  {
    for ( j = 0; j < m; j++ )
    {
      x[j] = ( double ) rand ( ) / ( double ) RAND_MAX;
    }
    value  = f ( m, x );
    f1_part = f1_part + value;
    f2_part = f2_part + value * value;
  }
}
```

CODE: Almost There!

Once all the workers have completed their loops, the answer has been computed, but it's all over the place. It needs to be communicated to the master.

Each worker has variables called **f1_part** and **f2_part**, and the master has these same variables, set to 0. We know **MPI_Bcast()** sends data, so is that what we do?

The first worker to broadcast **f1_part** would:

- successfully transmit that value to the master, replacing the value 0 by the value it computed;
- unfortunately, also transmit that same value to all the other workers, overwriting their values. That's a catastrophe!

CODE: The MPI_Reduce Command

In parallel programming, it is very common to have pieces of a computation spread out across the processes in such a way that the final required step is to add up all the pieces. A gathering process like this is sometimes called a *reduction operation*.

The function **MPI_Reduce()** can be used for our problem. It assumes that every process has a piece of information, and that this information should be assembled (added? multiplied?) into one value on one process.

CODE: Rules for MPI_Reduce (C version)

ierr = MPI_Reduce (data, result, count, type, op, to, comm)

- Input, **data**, the address of the local data;
- Output only on receiver, **result**, the address of the result;
- Input, int **count**, the number of data items;
- Input, **type**, the data type, such as **MPI_DOUBLE**;
- Input, **op**, **MPI_SUM, MPI_PROD, MPI_MAX...**;
- Input, int **to**, the process ID which collects data;
- Input, **comm**, usually **MPI_COMM_WORLD**;
- Output, int **ierr**, is nonzero if an error occurred.

CODE: Rules for MPI_Reduce (F90 version)

```
call MPI_Reduce ( data, result, count, type, op, to, comm, ierr )
```

- Input, **data**, the address of the local data;
- Output only on receiver, **result**, the address of the result;
- Input, integer **count**, the number of data items;
- Input, **type**, the data type, such as **MPI_DOUBLE**;
- Input, **op**, **MPI_SUM, MPI_PROD, MPI_MAX...**;
- Input, integer **to**, the process ID which collects data;
- Input, **comm**, usually **MPI_COMM_WORLD**;
- Output, integer **ierr**, is nonzero if an error occurred.

CODE: using MPI_Reduce()

```
//  The master zeros out F1 and F2.
//
  if ( id == 0 )
  {
    f1 = 0.0;
    f2 = 0.0;
  }
//
//  The partial results in F1_PART and F2_PART
//  are gathered into F1 and F2 on the master
//
  MPI_Reduce ( &f1_part, &f1, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD );

  MPI_Reduce ( &f2_part, &f2, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD );
```

CODE: The Master Finishes up

The master process has the values of **f1** and **f2** summed up from all the workers. Now there's just a little more to do!

```
  if ( id == 0 )
  {
    f1 = f1 / ( double ) n;
    f2 = f2 / ( double ) ( n - 1 );
    stdev = sqrt ( f2 - f1 * f1 );
    sterr = stdev / sqrt ( ( double ) n );
    error = fabs ( f1 - 1.0 );
    printf ( "%7d  %.15g  %6.4f  %6.4f  %8g\n",
      n, f1, stdev, sterr, error );
  }

  MPI_Finalize ( );
```

http://people.sc.fsu.edu/∼jburkardt/latex/acs2_mpi_2013/myprog_mpi.c

TIME FOR A BREAK!

CODE: FORTRAN90 Version

Let's go through the equivalent FORTRAN90 version of our code.

```
 program main

 use mpi

 (declarations here)

 call MPI_Init ( ierr )
 call MPI_Comm_size ( MPI_COMM_WORLD, p, ierr )
 call MPI_Comm_rank ( MPI_COMM_WORLD, id , ierr )

m = 2
if ( id == 0 ) then
  n = 1000000
  np = n / ( p - 1 )
  n = ( p - 1 ) * np
end if

 call MPI_Bcast ( np, 1, MPI_INTEGER, 0, MPI_COMM_WORLD,
     ierr )
```

http://people.sc.fsu.edu/∼jburkardt/latex/acs2_mpi_2013/myprog_mpi.f90

CODE: FORTRAN90 Version

The workers do their part:

```
  f1_part = 0.0
  f2_part = 0.0

  if ( 0 < id ) then
!
!  Call my function to scramble the random numbers.
!
    seed = 12345 + id
    call srand_f90 ( seed )

    do i = 1, np
      call random_number ( harvest = x(1:m) )
      value = f ( m, x )
      f1_part = f1_part + value
      f2_part = f2_part + value * value
    end do
  end if
```

CODE: FORTRAN90 Version

The master gathers and publishes

```
  if ( id == 0 ) then
    f1 = 0.0
    f2 = 0.0
  end if

  call MPI_Reduce ( f1_part , f1 , 1, MPI_DOUBLE_PRECISION ,
      MPI_SUM , 0 , MPI_COMM_WORLD , ierr )
  call MPI_Reduce ( f2_part , f2 , 1, MPI_DOUBLE_PRECISION ,
      MPI_SUM , 0 , MPI_COMM_WORLD , ierr )

  if ( id == 0 ) then
    f1 = f1 / dble ( n )
    f2 = f2 / dble ( n - 1 )
    stdev = sqrt ( f2 - f1 * f1 )
    sterr = stdev / sqrt ( dble ( n ) )
    error = abs ( f1 - 1.0D+00 )
    write ( *, '(i7,g14.6,2x,f10.4,2x,f10.4,2x,g8.2)' ) n,
        f1 , stdev , sterr , error
  end if

  call MPI_Finalize ( ierr )
```

CODE: A “Minor” Improvement

When we run our program, what really happens?

1. The master sets up stuff, the workers are idle;
2. The master is idle, the workers compute;
3. The master collects stuff, the workers are idle.

The first and last steps don’t take very long.

But why do we leave the master idle for the (perhaps lengthy) time that the workers are busy? Can’t it help?

Indeed, all we have to do is remove the restriction:

```
if ( 0 < id )
```

on the computation, and divide the work in pieces of size:

```
np = n / p
```

CODE: Comments

To do simple things in MPI can seem pretty complicated. To send an integer from one process to another, I have to call a function with a long name, specify the address of the data, the number of data items, the type of the data, identify the process and the communicator group.

But if you learn MPI, you realize that this complicated call means something like "send the integer NP to process 7". The complicated part is just because one function has to deal with different data types and sizes and other possibilities.

By the way, MPI_COMM_WORLD simply refers to all the processes. It's there because sometimes we want to define a subset of the processes that can talk just to each other. We're allowed to make up a new name for that subset, and to restrict the "broadcast" and "reduce" and "send/receive" communications to members only.

Distributed Memory Programming with MPI

RUN: Installation of MPI

To use MPI, you need a version of MPI installed somewhere (your desktop, or a cluster), which:

- places the include files “mpi.h”, “mpif.h” and the F90 MPI module where the compilers can find it;
- places the MPI library where the loader can find it;
- defines MPI compilers that automatically find the include file and library;
- installs mpirun, or some other system that synchronizes the startup of the user programs, and allows them to communicate.

Two free implementations of MPI are OpenMPI and MPICH:

- http://www.open-mpi.org/
- http://www.mcs.anl.gov/research/projects/mpich2/

RUN: MPI on the Lab Machines

MPI is installed on the lab machines.

Our lab machines have 8 cores, and each can run an MPI process.

To access the MPI include files, compilers, libraries, and run time commands, you must type

```
module load openmpi-x86_64
```

Rather than typing this command every time you log in, you can insert it into your **.bashrc** file in your main directory. Then it will be set up automatically for you every time you log in.

RUN: Working on Your Laptop

You can install MPI on your laptop or personal machine. If you're using a Windows machine, you'll need to make a Unix partition or install VirtualBox or Cygwin or somehow get Unix set up on a piece of your system.

If your machine only has one core, that doesn't matter. MPI will still run; it just won't show any speedup.

Moreover, you can run MPI on a 1-core machine and ask it to emulate several processes. This will do a "logical" parallel execution, creating several executables with separate memory that run in a sort of timesharing way.

RUN: Working on Your Laptop

If your desktop actually has multiple cores, MPI can use them.

Even if you plan to do your main work on a cluster, working on your desktop allows you to test your program for syntax (does it compile?) and for correctness (does it work correctly on a small version of the problem?).

If your MPI program is called **mprog.c**, then depending on the version of MPI you installed, you might be able to compile and run with the commands:

```
mpicc -o myprog myprog.c
mpirun -np 4 ./myprog     <-- -np 4 means run
                              with 4 MPI processes
```

RUN: Compiler Names

Of course, you have to use the right compiler for your language. Although the names can vary, here are typical names for the MPI compilers:

- **mpicc** for C programs;
- **mpic++**, for C++;
- **mpif77**, for Fortran77;
- **mpif90**, for Fortran90.

RUN: MPICC is “Really” GCC, or Something

The **mpicc** command is typically a souped up version of **gcc**, or the Intel C compiler or something similar.

The modified command knows where to find the extra features (the include file, the MPI run time library) that are needed.

Since **mpicc** is “really” a familiar compiler, you can pass the usual switches, include optimization levels, extra libraries, etc.

For instance, on the FSU HPC system, you type:

```
mpicc -c myprog_mpi.c        <-- to compile
mpicc myprog_mpi.c           <-- to compile and load,
                             creating "a.out"
mpicc myprog_mpi.c -lm       <-- to compile and load,
                             with math library
mpicc -o myprog myprog_mpi.c <-- to compile, load,
                             creating "myprog"
```

RUN: The FSU HPC

A big advantage of writing a program in MPI is that you can transfer your program to a computer cluster to get a huge amount of memory and a huge number of processors.

To take advantage of more processors, your **mpirun** command simply asks for more processes.

The FSU RCC HPC facility is one such cluster; any FSU researcher can get access, although students require sponsorship by a faculty member.

While some parts of the cluster are reserved for users who have contributed to support the system, there is always time and space available for general users.

RUN: Working with a Cluster

There are several headaches on a computer cluster:

- It's not your machine, so there are rules;
- You have to login from your computer to the cluster, using a terminal program like **ssh**;
- You'll need to move files between your local machine and the cluster; you do this with a program called **sftp**;
- The thousands of processors are not directly and immediately accessible to you. You have to put your desired job into a queue, specifying the number of processors you want, the time limit and so on. You have to wait your turn before it will even start to run.

RUN: Compiling on the FSU HPC

To compile on the HPC machine, transfer all necessary files to **sc.hpc.fsu.edu** using **sftp**, and then log in using **ssh** or some other terminal program.

On the HPC machine, there are several MPI environments. We'll set up the Gnu OpenMPI environment. For every interactive or batch session using OpenMPI, we will need to issue the following command first:

```
module load gnu-openmpi
```

You can insert this command into your **.bashrc** file on the cluster login node, so that it is done automatically for you.

Compile your program:

```
    mpicc -o myprog myprog_mpi.c
```

RUN: Executing in Batch on the FSU HPC

Jobs on the HPC system go through a batch system controlled by a scheduler called MOAB;

In exchange for being willing to wait, you get exclusive access to a given number of processors so that your program does not have to compete with other users for memory or CPU time.

To run your program, you prepare a batch script file. Some of the commands in the script "talk" to MOAB, which decides where to run and how to run the job. Other commands are essentially the same as you would type if you were running the job interactively.

One command will be the same **module...** command we needed earlier to set up OpenMPI.

RUN: A Batch Script for the HPC

```
#!/bin/bash
```

(Commands to MOAB:)

```
#MOAB -N myprog           <-- Name is "myprog"
#MOAB -q backfill         <-- Queue is "backfill"
#MOAB -l nodes=1:ppn=8    <-- Limit to 8 processors
#MOAB -l walltime=00:00:30 <-- Limit to 30 seconds
#MOAB -j oe               <-- Join output and error
```

(Define OpenMPI:)

```
module load gnu-openmpi
```

(Set up and run the job using ordinary interactive commands:)

```
cd $PBS_O_WORKDIR       <-- move to directory
mpirun -np 8 ./myprog   <-- run with 8 processes
```

---

http://people.sc.fsu.edu/∼jburkardt/latex/acs2_mpi_2013/myprog.sh

RUN: Submitting the Job

The command **-l nodes=1:ppn=8** says we want to get 8 processors. We are using the "backfill" queue, which allows general users to grab portions of the computer currently not being used by the "paying customers".

The **msub** command will submit your batch script to MOAB. If your script was called **myprog.sh**, the command would be:

```
msub myprog.sh
```

The system will accept your job, and immediately print a job id, just as **65057**. This number is used to track your job, and when the job is completed, the output file will include this number in its name.

RUN: The Job Waits For its Chance

The command **showq** lists all the jobs in the queue, with jobid, "owner", status, processors, time limit, and date of submission. The job we just submitted had jobid 65057.

```
44006     tomek   Idle  64 14:00:00:00  Mon Aug 25 12:11:12
64326  harianto   Idle  16 99:23:59:59  Fri Aug 29 11:51:05
64871   bazavov   Idle   1 99:23:59:59  Fri Aug 29 21:04:35
65059   ptaylor   Idle   1  4:00:00:00  Sat Aug 30 15:11:11
65057  jburkardt Idle   4     00:02:00  Sat Aug 30 14:41:39
```

To only show the lines of text with your name in it, type

```
  showq | grep jburkardt
```

...assuming your name is *jburkardt*, of course!

RUN: All Done!

At some point, the "idle" job will switch to "Run" mode. Some time after that, it will be completed. At that point, MOAB will create an output file, which in this case will be called **myprog.o65057**, containing the output that would have shown up on the screen. We can now examine the output and decide if we are satisfied, or need to modify our program and try again!

I often submit a job several times, trying to work out bugs. I hate having to remember the job number each time. Instead, I usually have the program write the "interesting" output to a file whose name I can remember:

```
mpirun -np 8 ./myprog  > myprog_output.txt
```

RUN:

Our simple integration problem seems to have become very complicated.

However, many of the things I showed you only have to be done once, and always in the same way:

- initialization
- getting ID and number of processes
- getting the elapsed time
- shutting down

The interesting part is determining how to use MPI to solve your problem, so we can put the uninteresting stuff in the main program, where it never changes.

RUN: Write the Main Program Once (C version)

```
int main ( int argc, char *argv[] )
{
  int id, p;
  double wtime;
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &id ); <-- id
  MPI_Comm_size ( MPI_COMM_WORLD, &p );  <-- count
  wtime = MPI_Wtime ( );
  ...                            Now write a function
  do_work ( id, p );    <-- that does work of
  ...                            process id out of p.
  wtime = MPI_Wtime ( ) - wtime;
  MPI_Finalize ( );
  return 0;
}
```

RUN: Write the Main Program Once (F90 version)

```
program main
  use mpi
  integer id, ierr, p
  double precision wtime
  call MPI_Init ( ierr )
  call MPI_Comm_rank ( MPI_COMM_WORLD, id, ierr ) <-- id
  call MPI_Comm_size ( MPI_COMM_WORLD, p, ierr )  <-- count
  wtime = MPI_Wtime ( )
  ...                            Now write a function
  call do_work ( id, p ) <--     that does work of
  ...                            process id out of p.
  wtime = MPI_Wtime ( ) - wtime
  call MPI_Finalize ( ierr )
  stop
end
```

RUN:

Similarly, the process of compiling your program with MPI is typically the same each time;

Submitting your program to the scheduler for execution also is done with a file, that you can reuse; occasionally you may need to modify it to ask for more time or processors.

But the point is, that many of the details I've discussed you only have to worry about once.

What's important then is to concentrate on how to set up your problem in parallel, using the ideas of message passing.

RUN:

For our integration problem, I used the simple communication routines, **MPI_Bcast()** and **MPI_Reduce()**.

But you can send any piece of data from any process to any other process, using **MPI_Send()** and **MPI_Receive()**.

These commands are trickier to understand and use, so you should refer to a good reference, and find an example that you can understand, before trying to use them.

However, basically, they are meant to do exactly what you think: send some numbers from one program to another.

If you understand the Send and Receive commands, you should be able to create pretty much any parallel program you need in MPI.

Distributed Memory Programming with MPI

SEND/RECV: Message Passing

Our integration example used **MPI_Bcast()** to broadcast the number of evaluation points for each process, and **MPI_Reduce()** to gather the partial sums together. These are two examples of message passing functions. Both functions assume that the communication is between one special process and all the others.

The **MPI_Send** and **MPI_Recv** functions allow any pair of processes to communicate directly.

Let's redo the integration program using the send/receive model.

SEND/RECV: The Communicator Group

An MPI message is simply a set of **n** items of data, all of the same type.

The values in this data are to be copied from one particular process to another. Both processes have an array of the same name. The data from the sender is copied to the receiver, overwriting whatever was there.

The data constitutes a message, and this message can be given a *tag*, which is simply a numeric identifier. In complicated programs, a receiver might have several incoming messages, and could choose which one to open first based on the tag value.

MPI allows you to organize processes into communicator groups, and then restrict some communication to that group. We'll never use that capability, but we still have to declare our sender and receiver are in **MP_COMM_WORLD**.

In cases where multiple messages are expected, a structure (C) or array (FORTRAN) called **status** is used to figure out who sent the message. This is something else we will not use!

SEND/RECV: The Message Is Transferred

The default send and receive functions are very polite. The send sends the data, and then program execution pauses...

The receiver may have been waiting a long time for this message. Or, it may be doing other things, and only after a while reaches the receive command. Once that happens, the message is received, and the receiver can go to the next command.

Once the message is received, MPI tells the sender the communication was successful, and the sender goes on.

You can see that, aside from the time it takes to transmit data, there is also a definite synchronization issue that can arise. What's worse, it's possible to accidentally write your program in such a way that everyone is sending, and no one is receiving, causing deadlock. We won't go into this now.

SEND/RECV: MPI_Send (C version)

error = MPI_Send (&data, count, type, to, tag, communicator);

- Input, (any type) **&data**, the address of the data;
- Input, int **count**, the number of data items;
- Input, int **type**, the data type (**MPI_INT**, **MPI_FLOAT**...);
- Input, int **to**, the process ID to which data is sent;
- Input, int **tag**, a message identifier, (0, 1, 1492 etc);
- Input, int **communicator**, usually **MPI_COMM_WORLD**;
- Output, int **error**, is 1 if an error occurred;

SEND/RECV: MPI_Send (FORTRAN version)

call MPI_Send (data, count, type, to, tag, communicator, error)

- Input, (any type) **data**, the data, scalar or array;
- Input, integer **count**, the number of data items;
- Input, integer **type**, the data type (**MPI_INTEGER**, **MPI_REAL**...);
- Input, integer **to**, the process ID to which data is sent;
- Input, integer **tag**, a message identifier, that is, some numeric identifier, such as 0, 1, 1492, etc;
- Input, integer **communicator**, set this to **MPI_COMM_WORLD**;
- Output, integer **error**, is 1 if an error occurred;

SEND/RECV: MPI_Recv (C version)

error= MPI_Recv (&data, count, type, from, tag, communicator, status);

- Input, (any type) **&data**, the address of the data;
- Input, int **count**, number of data items;
- Input, int **type**, the data type (must match what is sent);
- Input, int **from**, the process ID you expect the message from, or if don't care, **MPI_ANY_SOURCE**;
- Input, int **tag**, the message identifier you expect, or, if don't care, **MPI_ANY_TAG**;
- Input, int **communicator**, usually **MPI_COMM_WORLD**;
- Input, MPI_Status **status**, (auxiliary diagnostic information).
- Output, int **error**, is 1 if an error occurred;

SEND/RECV: MPI_Recv (FORTRAN version)

call MPI_Recv(data, count, type, from, tag, communicator, status, error)

- Output, (any type) **data**, the data (scalar or array);
- Input, integer **count**, number of data items expected;
- Input, integer **type**, the data type (must match what is sent);
- Input, integer **from**, the process ID from which data is received (must match the sender, or if don't care, **MPI_ANY_SOURCE)**;
- Input, integer **tag**, the message identifier (must match what is sent, or, if don't care, **MPI_ANY_TAG**);
- Input, integer **communicator**, (must match what is sent);
- Input, integer **status**(MPI_STATUS_SIZE), (auxiliary diagnostic information in an array).
- Output, integer **error**, is 1 if an error occurred;

SEND/RECV: How SEND and RECV Must Match

```
MPI_Send ( data, count, type, to,   tag, comm )
                     |      |    V     |     |
MPI_Recv ( data, count, type, from, tag, comm, status )
```

The **MPI_SEND()** and **MPI_RECV()** must match:

1. **count**, the number of data items, must match;
2. **type**, the type of the data, must match;
3. **to**, must be the ID of the receiver.
4. **from**, must be the ID of the sender, or **MPI_ANY_SOURCE**.
5. **tag**, a message "tag" must match, or **MPI_ANY_TAG**.
6. **comm**, the name of the communicator, must match (for us, always **MPI_COMM_WORLD**).

Distributed Memory Programming with MPI

CODE2: What Do We Change?

Using **MPI_Bcast()** to broadcast the value of **np**:

- all the processes executed the same **MPI_Broadcast()** command;
- process 0 was identified as the “sender”;

Using **MPI_Send()** and **MPI_Recv()** to broadcast the value of **np**:

- process 0 calls **MPI_Send()** **p-1** times, sending to each process;
- Each process calls **MPI_Recv()** once, to receive its message.

CODE2: The Original Version

```
//
//  The Broadcast command sends the value NP
//  from the master to all workers.
//
  MPI_Bcast ( &np, 1, MPI_INT, 0, MPI_COMM_WORLD );
```

CODE2: The Revised Version

```
//
//  The master process sends the value NP to all workers.
//
tag1 = 100;
if ( id == 0 )
{
  for ( to = 1; to < p; to++ )
  {
    MPI_Send ( &np, 1, MPI_INT, to, tag1, MPI_COMM_WORLD );
  }
}
else
{
 from = 0;
  MPI_Recv ( &np, 1, MPI_INT, from, tag1, MPI_COMM_WORLD,
    status );
}
```

CODE2: What Do We Change?

Using **MPI_Reduce()** to sum the items **f1_part** into **f1**:

- all processes executed **MPI_Reduce()**, with process 0 identified as receiver;
- the reduction was identified as an **MPI_SUM**;
- the values of **f1_part** were automatically summed into **f1**;

Using **MPI_Send()** and **MPI_Recv()**:

- Process 0 must call **MPI_Recv() p-1** times, "collecting" **f1_part** from each worker process;
- Process 0 must explicitly add **f1_part** to **f1**;
- Each worker process calls **MPI_Send()** to send its contribution.

CODE2: Original Version

```
//
//  The partial results in F1_PART and F2_PART
//  are gathered into F1 and F2 on the master
//
  MPI_Reduce ( &f1_part, &f1, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD );

  MPI_Reduce ( &f2_part, &f2, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD );
```

CODE2: Revised Version

```
tag2 = 200;
tag3 = 300;
if ( id != 0 )
{
  to = 0;
  MPI_Send ( &f1_part , 1, MPI_DOUBLE, to , tag2 ,
       MPI_COMM_WORLD ) ;
  MPI_Send ( &f2_part , 1, MPI_DOUBLE, to , tag3 ,
       MPI_COMM_WORLD ) ;
}
else
{
  f1 = f1_part ;
  f2 = f2_part ;
  for ( from = 1; from < p; from++ )
  {
    MPI_Recv ( &f1_part , 1, MPI_DOUBLE, from , tag2 ,
         MPI_COMM_WORLD, status ) ;
    f1 = f1 + f1_part ;
    MPI_Recv ( &f2_part , 1, MPI_DOUBLE, from , tag3 ,
         MPI_COMM_WORLD, status ) ;
    f2 = f2 + f2_part ;
  }
```

CODE2: Comments

It should be clear that we just made our program longer and uglier.

You can see that **MPI_Bcast()** and **MPI_Reduce()** are very convenient abbreviations for somewhat complicated operations.

We went "backwards" because we want to understand how the **MPI_Send()** and **MPI_Recv()** commands work, so we took a simple example and looked at it more closely.

The "tag" variables allow us to have several kinds of messages, but to send or receive only a particular one. Here, we used tags of 100, 200 and 300 to indicate the messages containing **np**, **f1_part** and **f2_part**.

Distributed Memory Programming with MPI

SATISFY: The Logical Satisfaction Problem

Recall the "SATISFY" problem, which we discussed earlier as an example for OpenMP programming.

Logicians and electric circuit designers both worry about the logical satisfaction problem. When logicians describe this problem, they assume that they have n logical variables b_1 through b_n, each of which can be **false=0** or **true=1**.

Moreover, they have a formula involving these variables, as well as logical operators such as **and**, **or**, **not**.

```
    f = b_1 AND ( b_2 OR ( b_1 AND NOT b_5 ) OR ...
```

Their simple question is, what values of the variables b will make the formula have a value that is **true**, that is, what values satisfy this formula?

SATISFY: The Brute Force Approach

While there are some cases of the satisfaction problem that allow for an efficient approach, there is an obvious brute force approach that has three advantages:

- it always finds all the solutions;
- it is easy to program;
- it is easy to do in parallel.

The brute force approach is:

1. generate every possibility
2. check if it's a solution

Under MPI, we can solve this problem in parallel if we can determine how to have each process check a limited range, and report all solutions encountered.

SATISFY: Logical Vector = Binary Number

We are searching all possible logical vectors of dimension n. A logical vector, containing TRUE or FALSE, can be though of as a binary vector, which can be thought of as an integer:

```
FFFF = (0,0,0,0) = 0
FFFT = (0,0,0,1) = 1
FFTF = (0,0,1,0) = 2
FFTT = (0,0,1,1) = 3
...
TTTT = (1,1,1,1) = 15
```

so we can parallelize this problem by computing the total number of possibilities, which will be 2^n, and dividing up the range among the processors.

SATISFY: Logical Vector = Binary Number

For example, we might have 1,024 possibilities, and 10 processes.

A formula for the ranges is:

```
lo = floor ( (   p       * 1024 ) / 10 )
hi = floor ( ( ( p + 1 ) * 1024 ) / 10 )
```

giving us the table:

```
Process P    Start     Stop before
-----------  --------  -----------
    0          0       102
    1         102      204
    2         204      307
    3         307      409
   ...        ...      ...
    9         921     1024
```

SATISFY: A Simple FORTRAN Code

```
!
!  Compute the number of binary vectors to check.
!
  ihi = 2 ** n

  write ( *, '(a)' ) ''
  write ( *, '(a,i6)' ) '  The number of logical variables is N = ", n
  write ( *, '(a,i6)' ) '  The number of input vectors to check is ", ihi
  write ( *, '(a)' ) ''
  write ( *, '(a)' ) '   #       Index    ---------Input Values------------------------'
  write ( *, '(a)' ) ''
!
!  Check every possible input vector.
!
  solution_num = 0;

  do i = 0, ihi - 1

    call i4_to_bvec ( i, n, bvec )

    value = circuit_value ( n, bvec )

    if ( value == 1 ) then
      solution_num = solution_num + 1
      write ( *, '(2x,i2,2x,i10)' ) solution_num, i
      write ( *, '(23i1)' ) bvec(1:n)
    end if
  end do
```

http://people.sc.fsu.edu/∼jburkardt/f_src/satisfy/satisfy.html

SATISFY: Issues

What issues do we face in creating a parallel version?

- We must define a range for each processor;
- The only input is **n**;
- Each processor works completely independently;
- Each processor prints out any solution it finds;
- The only common output variable is **solution_num**.

SATISFY: Write the Main Program Once (F90 version)

```
program main
  use mpi
  integer id, ierr, p
  double precision wtime
  call MPI_Init ( ierr )
  call MPI_Comm_rank ( MPI_COMM_WORLD, id, ierr ) <-- id
  call MPI_Comm_size ( MPI_COMM_WORLD, p, ierr )  <-- count
  wtime = MPI_Wtime ( )
  ...
  call satisfy_part ( id, p )
  ...
  wtime = MPI_Wtime ( ) - wtime
  call MPI_Finalize ( ierr )
  stop
end
```

http://people.sc.fsu.edu/∼jburkardt/f_src/satisfy_mpi/satisfy_mpi.html

SATISFY: The Subprogram

```
subroutine satisfy_part ( id, p )
  integer, parameter :: n = 23
  integer bvec(n), circuit_value, i, ihi, ilo
  integer solution_num, solution_num_local, value
  integer id, ierr, p

  ilo =   id         * ( 2 ** n ) / p
  ihi = ( id + 1 ) * ( 2 ** n ) / p
  solution_num_local = 0

  do i = ilo, ihi - 1
    call i4_to_bvec ( i, n, bvec )
    value = circuit_value ( n, bvec )
    if ( value == 1 ) then
     solution_num_local = solution_num_local + 1
     write ( *, * ) solution_num_local, id, i, bvec(1:n)
    end if
  end do
  call MPI_Reduce ( solution_num_local, solution_num, 1,
      MPI_INTEGER, &
    MPI_SUM, 0, MPI_COMM_WORLD, ierr )
  return
end
```

SATISFY: The output

If we run our code, assuming the functions i4_to_bvec() and circuit_value() are suppplied, we get output like this:

```
#   P      I       BVEC
1   1   3656933   01101111100110011100101
2   1   3656941   01101111100110011101101
3   1   3656957   01101111100110011111101
4   1   3661029   01101111101110011100101
5   1   3661037   01101111101110011101101
6   1   3661053   01101111101110011111101
7   1   3665125   01101111110110011100101
1   2   5754104   10101111100110011111000
2   2   5754109   10101111100110011111101
3   2   5758200    0101111101110011111000
4   2   5758205   10101111101110011111101
1   3   7851229   11101111100110011011101
2   3   7851261   11101111100110011111101
3   3   7855325   11101111101110011011101
4   3   7855357   11101111101110011111101
```

Distributed Memory Programming with MPI

HEAT: Independent Cooperating Processes

Now that we have an idea of how the **MPI_Send()** and **MPI_Recv()** commands work, we will look at an example where we really need them!

The physical problem we will consider involves the behavior of temperature along a thin wire over time.

The heat equation tells us that we can solve such a problem given an initial condition (all temperatures at the starting time) and the boundary conditions (temperatures, for all times, at the ends of the wires.)

There are standard techniques for discretizing this problem.

The parallel feature will come when we apply domain decomposition - that is, we will divide the wire up into segments that are assigned to a process. As you can imagine, each process will need to communicate with its left and right neighbors.

HEAT: Statement of the Problem

Determine the values of $H(x,t)$ over a range of time $t_0 <= t <= t_1$ and space $x_0 <= x <= x_1$, given an initial condition:

$$H(x, t_0) = ic(x)$$

boundary conditions:

$$\begin{aligned} H(x_0, t) =& \text{bcleft}(t) \\ H(x_1, t) =& \text{bcright}(t) \end{aligned}$$

and a partial differential equation:

$$\frac{\partial H}{\partial t} - k\frac{\partial^2 H}{\partial x^2} = f(x,t)$$

where k is the thermal diffusivity and f represents a heat source term.

HEAT: Discretized Heat Equation

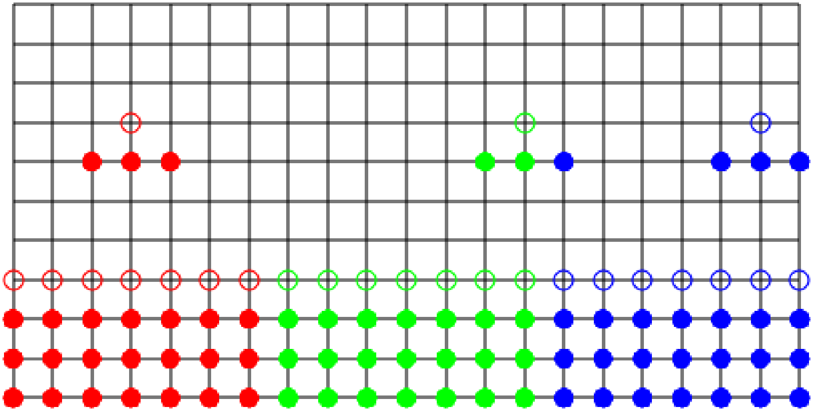
The discrete version of the differential equation is:

$$\frac{h(i,j+1)-h(i,j)}{dt} - k\frac{h(i-1,j)-2h(i,j)+h(i+1,j)}{dx^2} = f(i,j)$$

We have the values of $h(i,j)$ for $0 <= i <= N$ and a particular "time" j. We seek value of h at the "next time", $j+1$.

Boundary conditions give us $h(0,j+1)$ and $h(N,j+1)$, and we use the discrete equation to get the values of h for the remaining spatial indices $0 < i < N$.

HEAT: Red, Green and Blue Processes Cooperating



HEAT: What Messages Are Needed?

At a high level of abstraction, it's easy to see how this computation could be done by three processes, which we can call **red**, **green** and **blue**, or perhaps "0", "1", and "2".

Each process has a part of the *h* array.

The **red** process, for instance, updates $h(0)$ using boundary conditions, and h(1) through h(6) using the differential equation.

Because **red** and **green** are neighbors, they will also need to exchange messages containing the values of $h(6)$ and $h(7)$ at the nodes that are touching.

C, C++, FORTRAN77 and FORTRAN90 versions of an MPI program for this 1D heat program are available.

See, for example http://people.sc.fsu.edu/∼jburkardt/c_src/heat_mpi/heat_mpi.html

HEAT: Can This Work For A Hairier Problem?

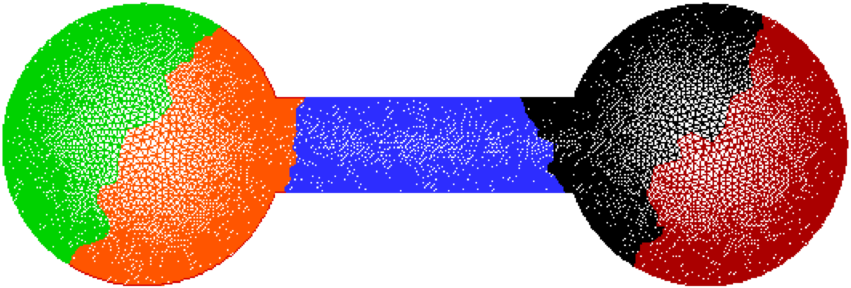
In more realistic examples, it's actually difficult just to figure out what parts of the problem are neighbors, and to figure out what data they must share in order to do the computation.

In a finite element calculation, in general geometry, the boundaries between the computational regions can be complicated.

But we can still break big problems into smaller ones if we can:

- create smaller, reasonably shaped geometries;
- identify the boundary elements;
- locate the neighbors across the boundaries;
- communicate data across those boundaries.

HEAT: A Complicated Domain Can Be Decomposed



A region of 6,770 elements, subdivided into 5 regions of similar size and small boundary by the ParMETIS partitioning program.

ParMETIS is available from http://glaros.dtc.umn.edu/gkhome/

HEAT: Decomposition is a Common MPI Technique

So why does domain decomposition work for us?

Domain decomposition is one simple way to break a big problem into smaller ones.

Because the decomposition involves geometry, it's often easy to see a good way to break the problem up, and to understand the structure of the boundaries.

Each computer sees the small problem and can solve it quickly.

The artificial boundaries we created by subdivision must be “healed” by trading data with the appropriate neighbors.

To keep our communication costs down, we want the boundaries between regions to be as compact as possible.

HEAT: One Process's View of the Heat Equation

If we have N nodes and P processes, then process K is responsible for computing the heat values at $K = N/P$ nodes. We suppose the process is given the starting values, and must compute estimates of their changing values over M time steps.

If you have the current values of your K numbers, you have enough information to update $K - 2$ of them.

But to update your first value, you need to:

- use a boundary rule if your ID is 0
- or call process ID-1 to copy his K-th value.

Similarly, updating your K-th value requires you to:

- use a boundary rule if your ID is P-1
- or call process $ID+1$ to copy his first value.

Obviously, your neighbors will also be calling you!

HEAT: A Process Communicates with Its Neighbors

We assume here that each process is responsible for **K** nodes, and that each process stores the heat values in an array called **H**. Since each process has separate memory, each process uses the same indexing scheme, **H[1]** through **H[K]**, even though these values are associated with different subintervals of [0,1].

The X interval associated with process ID is $[\frac{ID*N}{P*N-1}, \frac{(ID+1)*N-1}{P*N-1}]$;

Include two locations, **H[0]** and **H[K+1]**, for values copied from neighbors. These are sometimes called "ghost values".

It's easy to update **H[2]** through **H[K-1]**.

To update **H[1]**, we'll need **H[0]**, copied from our lefthand neighbor (where this same number is stored as **H[K]**!).

To update **H[K]**, we'll need **H[K+1]** copied from our righthand neighbor.

HEAT: It Looks Like This Might Work

This program would be considered a good use of MPI, since the problem is easy to break up into cooperating processes.

The amount of communication between processes is small, and the pattern of communication is very regular.

The data for this problem is truly distributed. No single process has access to the whole solution.

The individual program that runs on one process looks a lot like the sequential program that would solve the whole problem.

It's not too hard to see how this idea could be extended to a similar time-dependent heat problem in a 2D rectangle.

Distributed Memory Programming with MPI

PROGRAM: Initialize and Finalize

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

int main ( int argc, char *argv[] )
{
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &id );
  MPI_Comm_size ( MPI_COMM_WORLD, &p );

   The "good stuff" goes here in the middle!

  MPI_Finalize ( );
  return 0;
}
```

PROGRAM: What Needs to Be Communicated?

As we begin our calculation, processes 1 through **P**-1 must send what they call **h[1]** to their "left neighbor".

Processes 0 through **P**-2 must receive these values, storing them in the ghost value slot **h[k+1]**.

Similarly, processes 0 through **P**-2 send **h[k]** to their "right neighbor", which stores that value into the ghost slot **h[0]**.

Sending this data is done with matching calls to **MPI_Send** and **MPI_Recv**. The details of the call are more complicated than I am showing here!

PROGRAM: Pseudo Code for Communication

```
if ( 0 < id )
  MPI_Send ( h[1] => id-1 )

if ( id < p-1 )
  MPI_Recv ( h[k+1] <= id+1 )

if ( id < p-1 )
  MPI_Send ( h[k] => id+1 )

if ( 0 < id )
  MPI_Recv ( h[0] <= id-1 )
```

PROGRAM: Actually, This is a Bad Idea

Our communication scheme is defective however. It comes very close to **deadlock**, when processes sit in a useless wait state that will never end.

The problem here is that by default, an MPI process that sends a message won't continue until that message has been received.

So, if all the processes work exactly in synch, they will all try to send something to their left at the same time, and they can't move to the next step until the send is completed (that is, until the message is received.)

But no one can actually receive, because that's the **next** instruction... except that process 0 didn't have to send, so it can receive! That frees up process 1, which can now receive from process 2, and so on...

Our program will work, but MPI provides a combined **MPI_Send_Recv()** that could handle this situation in a better way.

PROGRAM: Once Data is Transmitted, Compute!

Once each process has received the necessary boundary information in **h[0]** and **h[k+1]**, it can use the four node stencil to compute the updated value of **h** at nodes 1 through **k**.

Actually, **hnew[1]** in the first process, and **hnew[k]** in the last one, need to be computed by boundary conditions.

But it's easier to treat them all the same way, and then correct the two special cases afterwards.

PROGRAM: The Time Update Step

```
  for ( i = 1; i <= k; i++ )
  {
    hnew[i] = h[i] + dt * (
    + k * ( h[i-1] - 2 * h[i] + h[i+1] ) /dx/dx
    + f ( x[i], t ) );
  }

  if ( id == 0    )
  {
    hnew[1] = bc ( 0.0, t );
  }
  if ( id == p-1 )
  {
    hnew[k] = bc ( 1.0, t );
  }
/*
  Replace old H by new.
*/
  for ( i = 1; i <= k; i++ )
  {
    h[i] = hnew[i];
  }
```

PROGRAM: The Brutal Details

Here is almost all the source code for a working version of the heat equation solver.

I've chopped it up a bit and compressed it, but I wanted you to see how things really look.

This example is available in C, C++, F77 and F90 versions. We will be able try it out with MPI later.

---

http://people.sc.fsu.edu/∼jburkardt/c_src/heat_mpi/heat_mpi.html
http://people.sc.fsu.edu/∼jburkardt/cpp_src/heat_mpi/heat_mpi.html
http://people.sc.fsu.edu/∼jburkardt/f77_src/heat_mpi/heat_mpi.html
http://people.sc.fsu.edu/∼jburkardt/f_src/heat_mpi/heat_mpi.html

PROGRAM: Main program

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include "mpi.h"                         <-- MPI Include file

int main ( int argc, char *argv[] )
{
  int id, p;
  double wtime;

  MPI_Init ( &argc, &argv );              <-- Start MPI
  MPI_Comm_rank ( MPI_COMM_WORLD, &id ); <-- Assign ID
  MPI_Comm_size ( MPI_COMM_WORLD, &p );  <-- Report number of processes.
  wtime = MPI_Wtime();                    <-- Start timer.

  update ( id, p );                       <-- Execute subprogram.

  wtime = MPI_Wtime() - wtime;            <-- Stop timer.
  MPI_Finalize ( );                       <-- Terminate.

  return 0;
}
```

PROGRAM: Auxilliary Functions

```
double boundary_condition ( double x, double time )

/*  BOUNDARY_CONDITION returns H(0,T) or H(1,T), any time. */
{
  if ( x < 0.5 )
  {
    return ( 100.0 + 10.0 * sin ( time ) );
  }
  else
  {
    return ( 75.0 );
  }
}

double initial_condition ( double x, double time )

/*  INITIAL_CONDITION returns H(X,T) for initial time. */
{
  return 95.0;
}

double rhs ( double x, double time )

/*  RHS returns right hand side function f(x,t). */
{
  return 0.0;
}
```

PROGRAM: UPDATE Initialization

```
void update ( int id, int p )
{
  Omitting declarations...
  k = n / p;

/* Set the X coordinates of the K nodes. */

  x = ( double * ) malloc ( ( k + 2 ) * sizeof ( double ) );

  for ( i = 0; i <= k + 1; i++ )
  {
    x[i] = ( ( double ) (         id * k + i - 1 ) * x_max
           + ( double ) ( p * k - id * k - i     ) * x_min )
           / ( double ) ( p * k              - 1 );
  }
/*  Set the values of H at the initial time. */

  time = 0.0;
  h     = ( double * ) malloc ( ( k + 2 ) * sizeof ( double ) );
  h_new = ( double * ) malloc ( ( k + 2 ) * sizeof ( double ) );
  h[0] = 0.0;
  for ( i = 1; i <= k; i++ )
  {
    h[i] = initial_condition ( x[i], time );
  }
  h[k+1] = 0.0;

  dt = time_max / ( double ) ( j_max - j_min );
  dx = ( x_max - x_min ) / ( double ) ( p * n - 1 );
```

PROGRAM: Set H[0] and H[K+1]

```
for ( j = 1; j <= j_max; j++ )
{
  time_new = j * dt;
/* Send H[1] to ID-1. */

  if ( 0 < id ) {
    tag = 1;
    MPI_Send ( &h[1], 1, MPI_DOUBLE, id-1, tag, MPI_COMM_WORLD );
  }
/* Receive H[K+1] from ID+1. */

  if ( id < p-1 ) {
    tag = 1;
    MPI_Recv ( &h[k+1], 1,  MPI_DOUBLE, id+1, tag, MPI_COMM_WORLD, &status );
  }
/* Send H[K] to ID+1. */

  if ( id < p-1 ) {
    tag = 2;
    MPI_Send ( &h[k], 1, MPI_DOUBLE, id+1, tag, MPI_COMM_WORLD );
  }
/* Receive H[0] from ID-1. */

    if ( 0 < id ) {
      tag = 2;
      MPI_Recv ( &h[0], 1, MPI_DOUBLE, id-1, tag, MPI_COMM_WORLD, &status );
    }

}
```

PROGRAM: Update the Temperatures

```
/* Update the temperature based on the four point stencil. */

    for ( i = 1; i <= k; i++ )
    {
      h_new[i] = h[i]
      + dt * k * ( h[i-1] - 2.0 * h[i] + h[i+1] ) / dx / dx
      + dt * rhs ( x[i], time );
    }
/*  Correct settings of first H in first interval, last H in last interval. */

    if ( 0 == id ) {
      h_new[1] = boundary_condition ( 0.0, time_new );
    }
    if ( id == p - 1 ) {
      h_new[k] = boundary_condition ( 1.0, time_new );
    }

/* Update time and temperature. */

    time = time_new;

    for ( i = 1; i <= k; i++ ) {
      h[i] = h_new[i];
    }

 }  <-- End of time loop
}  <-- End of UPDATE function
```

PROGRAM: Where is the Solution?

Since each of the P processes has computed the solution at K nodes, our solution is "scattered" across the machines.

If we needed to print out the solution at the final time as a single list, we could do that by having each process print its part (can be chaotic!) or they can each send their partial results to process 0, which can create and print a single unified result.

PROGRAM: Communication Issues

Our program, as written, still has some problems. If we imagine all the processes working about equally fast, then they all issue a SEND, and wait for it to be completed.

But the SEND's can't complete until someone on the other side receives the message. Nobody can, because they're all still waiting for their SEND's to complete as well.

Well, that's not true, because process **p-1** didn't have to send. So that process receives, freeing up process **p-2**. So then process **p-2** can receive its message from process **p-3** and so on.

This can be fixed by using **MPI_Isend()** and **MPI_Irecv()** which send the data immediately without waiting for confirmation of receipt.

Immediate SEND/RECV: MPI_ISend (C version)

error = MPI_Isend (&data, count, type, to, tag, communicator, &request);

- Input, (any type) **&data**, the address of the data;
- Input, int **count**, the number of data items;
- Input, int **type**, the data type (**MPI_INT**, **MPI_FLOAT**...);
- Input, int **to**, the process ID to which data is sent;
- Input, int **tag**, a message identifier, (0, 1, 1492 etc);
- Input, int **communicator**, usually **MPI_COMM_WORLD**;
- Output, MPI_Request **&request**, tracks the communication status;
- Output, int **error**, is 1 if an error occurred;

Immediate SEND/RECV: MPI_Isend (FORTRAN version)

call MPI_Isend (data, count, type, to, tag, communicator, request, error)

- Input, (any type) **data**, the data, scalar or array;
- Input, integer **count**, the number of data items;
- Input, integer **type**, the data type (**MPI_INTEGER**, **MPI_REAL**...);
- Input, integer **to**, the process ID to which data is sent;
- Input, integer **tag**, a message identifier, that is, some numeric identifier, such as 0, 1, 1492, etc;
- Input, integer **communicator**, set this to **MPI_COMM_WORLD**;
- Output, integer **request**, tracks the communication status;
- Output, integer **error**, is 1 if an error occurred;

Immediate SEND/RECV: MPI_Irecv (C version)

error= MPI_Irecv (&data, count, type, from, tag, communicator, &request);

- Input, (any type) **&data**, the address of the data;
- Input, int **count**, number of data items;
- Input, int **type**, the data type;
- Input, int **from**, the sender's ID or **MPI_ANY_SOURCE**;
- Input, int **tag**, the message tag or **MPI_ANY_TAG**;
- Input, int **communicator**, usually **MPI_COMM_WORLD**;
- Output, MPI_Request **&request**, tracks the communication status;
- Output, int **error**, is 1 if an error occurred;

Immediate SEND/RECV: MPI_Irecv (FORTRAN version)

call MPI_Irecv(data, count, type, from, tag, communicator, request, error)

- Output, (any type) **data**, the data (scalar or array);
- Input, integer **count**, number of data items expected;
- Input, integer **type**, the data type;
- Input, integer **from**, the sender's ID or **MPI_ANY_SOURCE**;
- Input, integer **tag**, the message tag or **MPI_ANY_TAG**;
- Input, integer **communicator**, (must match what is sent);
- Output, integer **request**, tracks the communication status;
- Output, integer **error**, is 1 if an error occurred;

Distributed Memory Programming with MPI

CONCLUSION: Comparison

Now you've seen two methods for programming an algorithm that can be carried out in parallel.

In OpenMP, we can think that we are improving a sequential algorithm by having loops that execute very fast because multiple threads are cooperating. Because the memory is shared, we have to be careful that individual threads don't interfere with each other. Data is public by default.

In MPI, we essentially have complete, separate sequential programs that are running in parallel. Memory is distributed (not shared), and so data is private by default. To share a data item, we must explicitly send a message.

CONCLUSION: Comparison

OpenMP essentially runs on one computer, and so parallelism is limited by the number of cores, and memory is limited to what is available on one computer.

MPI can run on hundreds or thousands of computers. The hardware limitation for MPI is the communication speed.

OpenMP is a simple system. It is easy to start with an existing program and make small, stepwise modifications, turning it into a parallel program one loop at a time.

MPI is a complex system. Typically, you need to rethink parts of your algorithm, create new data, and add function calls. Your MPI program may look substantially different from your original.

CONCLUSION: Web References

- **www-unix.mcs.anl.gov/mpi/**, Argonne Labs;
- **www.mpi-forum.org**, the MPI Forum
- **www.netlib.org/mpi/**, reports, tests, software;
- **www.open-mpi.org** , an open source version of MPI;
- **www.nersc.gov/nusers/help/tutorials/mpi/intro**
- Gropp, **Using MPI**;
- 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**;