

# Distributed Memory Programming With MPI

ISC 5316: Applied Computational Science II

.....

John Burkardt

Department of Scientific Computing

Florida State University

[http://people.sc.fsu.edu/~jburkardt/presentations/...  
...fsu\\_mpi\\_2011.pdf](http://people.sc.fsu.edu/~jburkardt/presentations/...fsu_mpi_2011.pdf)

25 & 27 October 2011

Lab on 01 November 2011, due 08 November 2011



# Distributed Memory Programming With MPI

- **MPI: Why, Where, How?**
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion



# MPI: We Work on Problems that are Too Big

Richardson tries to predict the weather.



# MPI: 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.

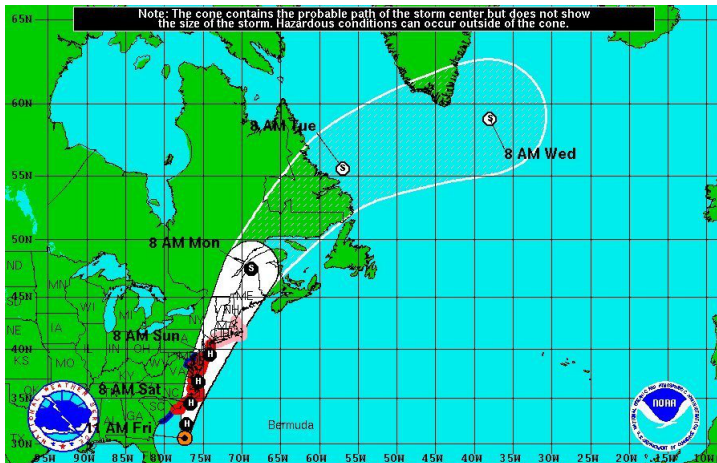
Over time, he was proved right, and the prediction of weather became one of the classic computational problems.

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


Events like Hurricane Irene in August 2011 (\$8 billion in damage) mean accurate weather prediction is worth paying for. (New York City shut the entire transportation system; but the real damage was in New England and Pennsylvania, where no one predicted massive rainstorms and floods that followed.)





# MPI: Big Problems Can Be Solved



**Hurricane Irene**  
 Friday August 26, 2011  
 11 AM EDT Advisory 25  
 NWS National Hurricane Center

**Current Information:**   
 Center Location 30.7 N 77.3 W  
 Max Sustained Wind 105 mph  
 Movement N at 14 mph

**Forecast Positions:**  
 Tropical Cyclone  Post-Tropical  
 Sustained Winds: D < 39 mph  
 S 39-73 mph H 74-110 mph M > 110mph

**Potential Track Area:**  
 Day 1-3  Day 4-5

**Watches:**  
 Hurricane  Trop.Storm

**Warnings:**  
 Hurricane  Trop.Storm



# MPI: For Years, Speeding Up the Clock was Enough

For many years, computer designers could keep up with the need for faster machines by improving current methods - shrinking circuits and making the electronic clock faster.

But in the 1990's, the cost of designing and building supercomputers of the traditional kind exploded.

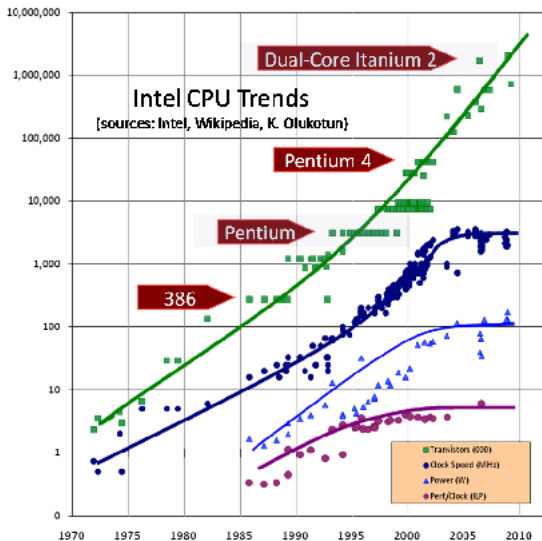
A real upper bound on performance was coming into view.

It was not possible to make the clock much faster, or the circuits much smaller.



# MPI: Sequential Computing Has Hit a Ceiling

CPU clock speeds have stopped at the 4 GigaHertz ceiling.



# MPI: 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?





## MPI: Early Clusters Were Ugly, but Worked



# MPI: 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.



# MPI: 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.



# MPI: Interactive Execution

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

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

```
mpirun -np 4 myprog > output.txt
```

This is useful for a quick check of a small program, but it does tie up the login node - other users will get mad if you try to run a big job this way!

The right place for your big MPI program to run is on the cluster itself, after being placed in a batch queue and scheduled by a job manager.



# MPI: Batch Execution

To execute on the cluster, your job must “stand in line” until the queueing system can arrange access to the number of processors you requested.

You talk to the queueing system using a “batch script”, a simple shell script that gives the characteristics of the job you want to run, and the sequence of commands you want to execute.

You submit the job to a batch system, perhaps like this:

```
msub myprog.sh
```

When your job is completed, two files are returned:

- an *output* file, such as **myprog.o6501**
- an *error* file, such as **myprog.e6501**

(For convenience, I always ask that these two files be combined.)



# MPI: Frequently Asked Questions

*Mr Garrison: "Please don't be afraid to ask any question you want in this class. Remember, there is no such thing as a stupid question, only stupid people."*

Running one program on **N** computers suggests some questions:

- **Q:** Do I need to learn a new computer language?
- **A:** *No, there is an MPI interface to C/C++/FORTRAN.*
- **Q:** Do I have to copy my program to many machines?
- **A:** *No, MPI makes copies and starts them together.*
- **Q:** How do we avoid doing the exact same thing **N** times?
- **A:** *MPI gives each computer a unique ID, and that's enough.*
- **Q:** Do we end up with **N** separate output files?
- **A:** *No, MPI collects them all together for you.*
- **Q:** So what's the hard part, then?
- **A:** *Rewriting your algorithm to work as cooperating processes*



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- **Overview of an MPI computation**
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- **Monte Carlo Integration in C++**
- Message Passing Options
- Conclusion



# OVERVIEW: Independent Cooperating Processes

Suppose that solving your problem can be thought of as solving  $N$  smaller problems simultaneously.

It's OK if the solution of these smaller problems is not completely independent; that is, from time to time, the program solving subproblem 17 might want to “talk” to the solvers of subproblems 16 and 18.

Then we may be able to use the original algorithm for the big problem, with some small changes that add communication, which might be thought of as filling in missing boundary information.

As long as the communication is limited, we can get a substantial speedup through parallel execution.





# OVERVIEW: The Heat Equation

We'll begin with a discussion of MPI computation "without MPI".

That is, we'll hold off on the details of the MPI language, but we will go through the motions of re-implementing a sequential algorithm using the capabilities of MPI.

The algorithm we have chosen is a simple example of **domain decomposition**, the time dependent heat equation on a wire (a one dimensional region).



## OVERVIEW: Statement of the Problem

Determine the values of  $H(x, t)$  over a range  $t_0 \leq t \leq t_1$  and space  $x_0 \leq x \leq x_1$ , given an initial value  $H(x, t_0)$ , boundary conditions, a heat source function  $f(x, t)$ , and a partial differential equation

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



# OVERVIEW: Discretized Heat Equation

The discrete version of the differential equation is:

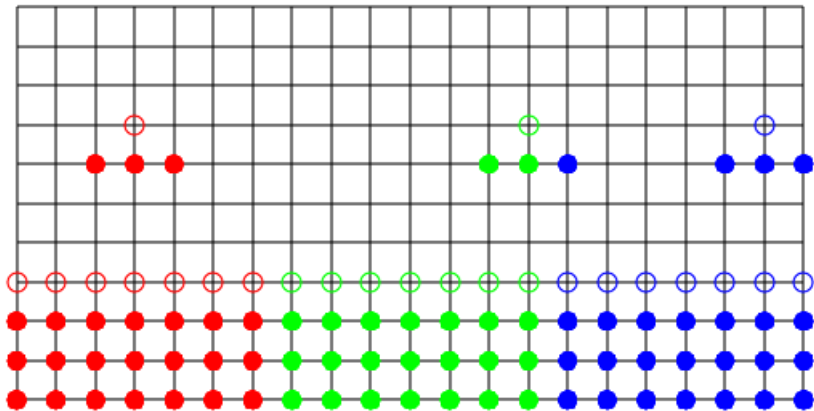
$$\frac{h(i, j + 1) - h(i, j)}{dt} - k \frac{h(i - 1, j) - 2h(i, j) + h(i + 1, j)}{dx^2} = f(i, j)$$

We have the values of  $h(i, j)$  for  $0 \leq i \leq N$  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  $0 < i < N$ .



# OVERVIEW: Red, Green and Blue Processes Cooperating



# OVERVIEW: 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](http://people.sc.fsu.edu/~jburkardt/c_src/heat_mpi/heat_mpi.html)



## OVERVIEW: 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.



# OVERVIEW: 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/>



# OVERVIEW: 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.





# OVERVIEW: Many Ways to Parallelize

Many problems are parallelizable, although the method of getting there depends on what you are doing.

To *sort a large array*, you think of doing it in parallel by sorting smaller arrays - and then exchanging some results.

To *solve a linear system*, you might break the matrix up into square sub-blocks, or strips of rows, and think of the related linear problem.

To *approximate an integral*, divide the range and sum up the result at the end.

To *compare a protein* to all the proteins in a database, have each process work with a portion of the database.



## OVERVIEW: One Program Binds Them All

Now we need to start thinking about how to write a single program that will still tell each process what its particular job is to be.

If we think of this from the process's point of view, it sounds like a Twilight Zone episode.

Your phone rings, and a voice tells you to wake up.

It rings again, and tells you that you are on a job involving **P**.

It rings again to tell you your ID number is **ID** (ID numbers run from 0 to **P**-1).

You go to your mailbox and open a program which contains your instructions.

*Is this any way to solve the HEAT problem?*



# OVERVIEW: One Process's View of the Heat Equation

You are responsible for  $K = N/P$  of the heat values. You're 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 get 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 get his first value.

Obviously, your neighbors will also be calling you!



# OVERVIEW: 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.



## OVERVIEW: 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 computer 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

- MPI: Why, Where, How?
- Overview of an MPI computation
- **Designing an MPI computation**
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion



## DESIGN: 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;
}
```



## DESIGN: 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!





## DESIGN: 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 )
```



## DESIGN: Actually, This is a Bad Idea

Our communication scheme is defective however. It comes very close to **deadlock**.

Deadlock is when a process is waiting for access to a device, or data or a message that we know will never actually arrive.

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

If you think about the implications, it's almost surprising that the code I have describe will work at all.

It will, but more slowly than it should!

Don't worry about this right now, but realize that with MPI you must also consider these communication issues.



## DESIGN: Once Data is Transmitted, Compute!

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

Actually,  $\mathbf{hnew}[1]$  in the first process, and  $\mathbf{hnew}[k]$  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.



# DESIGN: 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 ) );
}
/*
Process 0  sets left  node by BC
Process P-1 sets right node by BC
*/
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];
}
```



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- **The Heat Equation in C**
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion



# HEAT: 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 also available in a FORTRAN77 version. We will be able to send copies of these examples to an MPI machine for processing later.



# HEAT: 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;
}
```



# HEAT: 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;
}
```





# HEAT: 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 = time_min;
    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;

    time_delta = ( time_max - time_min ) / ( double ) ( j_max - j_min );
    x_delta    = ( x_max - x_min ) / ( double ) ( p * n - 1 );
}
```



# HEAT: Set H[0] and H[K+1]

```
for ( j = 1; j <= j_max; j++ )
{
    time_new = j * time_delta;
    /* 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 );
    }
}
```



# HEAT: Update the Temperatures

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

for ( i = 1; i <= k; i++ )
{
    h_new[i] = h[i]
    + ( time_delta * k / x_delta / x_delta ) * ( h[i-1] - 2.0 * h[i] + h[i+1] )
    + time_delta * 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
```



# HEAT: 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.



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- **Compiling, Linking, Running.**
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion



## RUN: Compiling, Perhaps on Your Laptop

The first step is to **compile** the program. An MPI program is written in a standard language, so if you are just checking for errors, you can do that on any machine - even your laptop.

```
gcc -c myprog.c
```

However:

- Your compiler needs the appropriate INCLUDE file.
- The resulting object code can't be used on another machine
- You can't check for linking errors without the MPI library

Compiling on your laptop can be a great way to check for syntax errors and quickly correct them. Sometimes editing directly on the HPC machine can be an awkward experience.



# RUN: Loading On a Laptop

In some cases, you may be able to go further on your laptop or desktop machine if you install a version of OpenMPI or MPICH.

In that case, the installation will automatically include the necessary libraries and include files. It will also mostly likely create customized scripts to be called for compilation and loading, which will typically be called **mpicc** and so on.

In that case, you could compile-only your file with a command like

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

or go straight to an executable with the command

```
mpicc myprog.c
```

---

OpenMPI: <http://www.open-mpi.org>

MPICH: <http://www.mcs.anl.gov/mpi/mpich2>



## RUN: Running On a Laptop

Again, assuming you have installed OpenMPI or MPICH, if your local system has multiple processors or cores, you may be able to run small MPI jobs immediately.

The typical way to do this involves the command **mpirun**.

For instance, if we renamed our executable program to **myprog**, and our local system had 4 processors, we could try to run it under MPI with the command

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

Here, the **-np 4** switch is specifying the number of processors on which MPI should run the program.





However, the whole point of writing a program in MPI is to get a huge amount of memory and a huge number of processors, and you do that by getting access to a computer cluster.

**Any** researcher in the FSU community can get such access, although students will 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.

---

FSU HPC Cluster Accounts: <http://hpc.fsu.edu>, "Your HPC Account: Apply for an Account"



# 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 setup 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
```

Now, to compile a program, we type one of the following:

```
mpicc -c myprog.c  
mpic++ -c myprog.cc  
mpif77 -c myprog.f  
mpif90 -c myprog.f90
```



# RUN: Link/Load on the FSU HPC

Linking combines your compiled code, the MPI libraries, and other system software. Loading creates an executable out of all that.

To link and load a C code you have already compiled:

```
mpicc myprog.o
```

or, to compile, link and load in just one step:

```
mpicc myprog.c
```

Either command creates the executable **a.out**. You should rename the executable to something meaningful:

```
mv a.out myprog
```

The libraries you need for MPI are complicated. The **mpicc** script hides all these details.



# RUN: Interactive Running on FSU HPC

You should not make a practice of running MPI programs interactively on the cluster. The login node is being shared with other users, who are also trying to work.

However, for sanity checks, for small programs that run a short time (10, 20 seconds), it is reasonable.

Assuming

- our executable program is named **myprog**,
- we are working in the directory containing that program,
- we have set up OpenMPI using the **source...** command

then we can run the program interactively with (say) 4 processors using the same **mpirun** command we mentioned for a laptop:

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

If the program goes crazy, or runs too long, kill it by typing **Control-C**, or **CTRL-C**.



# RUN: Executing in Batch on the FSU HPC

Most jobs on an MPI system go through a batch system.

That means you copy a script file, change a few parameters including the name of the program, and submit it.

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 **source...** command we needed earlier to set up OpenMPI.



# RUN: A Batch Script for the FSU HPC

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

*Commands to MOAB:*

```
#MOAB -N myprog          <-- Name is "myprog"  
#MOAB -q classroom      <-- Queue is "classroom"  
#MOAB -l nodes=1:ppn=4  <-- Limit to 4 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 4 ./myprog  <-- run with 4 processes
```



## RUN: Submitting the Job

The command **-l nodes=1:ppn=4** says we want to get 4 processors. For the classroom queue, there is a limit on the maximum number of processors you can ask for, and that limit is currently 32.

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 4 ./myprog > myprog_output.txt
```



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- **Monte Carlo Integration in Fortran77**
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion



# MC: Integral Estimate by Random Sampling

As a classroom exercise, we will try to put together a SIMPLE program to do numerical quadrature. To keep it even simpler, we'll do a Monte Carlo estimation, so there's little need to coordinate the efforts of multiple processes.

Here's the problem:

Estimate the integral of  $3 * x^2$  between 0 and 1.

Start by writing a sequential program, in which the computation is all in a separate function.



## MC: Program Outline

Choose a value for  $N$

Pick a seed for random number generator.

Set  $Q$  to 0

Loop  $N$  times:

    Pick a random  $X$  in  $[0,1]$ .

$$Q = Q + 3 X^2$$

End loop

Integral estimate is  $Q / N$



## MC: FORTRAN77 Program Segment

```
seed = 123456789
sample_num = 100000
q_exact = 1.0
q = 0.0
do sample = 1, sample_num
  x = r8_uniform_01 ( seed )    <-- Generate random x in [0,1]
  q = q + 3.0 * x * x
end do
q = q / sample_num

write ( *, * ) sample_num, q, abs ( q - q_exact )
```



## MC: Simple Parallelism

Once the sequential program is written, running, and running correctly, how much work do we need to do to turn it into a parallel program using MPI?

If we use the master-worker model, the master can collect all the estimates and average them for a final estimate. We can let the master participate in the computation, as well.

In the main program, we isolate ALL the MPI work of initialization, communication (send N, return partial estimate of Q) and wrapup.

We can think of an MPI program as a sequential program...  
...that can communicate with other sequential programs.



# MC: Initialization

```
program main
```

```
include 'mpif.h'
```

```
double precision f  
integer id, ierr, p  
double precision q, q_error, q_exact, q_total  
integer sample, sample_num, sample_total, seed  
double precision wtime, x
```

```
q_exact = 1.0  
sample_num = 100000
```

```
c  
c Initialize, get ID, number of processes, current time.  
c
```

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

```
if ( id .eq. 0 ) then  
    wtime = MPI_Wtime ( )  
end if
```



# MC: Each Process Calculates Q

```
c
c Each process, from ID=0 to P-1, uses a different seed.
c
c     seed = 123456789 + id
c
c This part looks the same as the sequential code
c (but runs with a different SEED).
c
c     q = 0.0
c     do sample = 1, sample_num
c       x = r8_uniform_01 ( seed )
c       q = q + 3.0 * x * x
c     end do
c     q = q / sample_num
c
c     write ( *, * ) sample_num, q, abs ( q - q_exact )
c
c Have each process sent results to process MASTER for reduction
c to final result.
c
c     call MPI_Reduce ( q, q_total, 1, MPI_DOUBLE_PRECISION, MPI_SUM,
c       & 0, MPI_COMM_WORLD, ierr )
```





# MC: The Master Process Reports Q\_TOTAL

```
c
c "Clean up" the result.
c
  if ( id .eq. 0 ) then
    q_total = q_total / dble ( p )
    q_error = abs ( q_total - q_exact )

    write ( *, * ) q_total, q_error

    wtime = MPI_Wtime ( ) - wtime

    write ( *, * ) ' Elapsed seconds = ', wtime

  end if
c
c Shut down MPI.
c
  call MPI_Finalize ( ierr )

  stop
end
```

---

C, C++, FORTRAN77 and FORTRAN90 versions of an MPI version are available.

See, for example [http://people.sc.fsu.edu/~jburkardt/f77\\_src/quad\\_mpi/quad\\_mpi.html](http://people.sc.fsu.edu/~jburkardt/f77_src/quad_mpi/quad_mpi.html)



## MC: The Mysterious “Reduction” Operation

In the main part of this program, you almost wouldn't know that MPI was involved. Each process churns out a bunch of random numbers, using them to compute an estimate  $q$  of the integral.

The MPI stuff happens when we need to collect all those  $q$  values and average them to get a super-accurate estimate for the integral.

We could do that using **MPI\_Send()** and **MPI\_Recv()**, but in this example, we have another example of a *reduction operation*: multiple pieces of data need to be summed to form a single result.

The function **MPI\_Reduce()** collects the value of  $q$  from each process and sums them to make  $q_{total}$ . We will come back and talk about this reduction operation in more detail later.



Monte Carlo calculations are often quite easy to do in MPI.

Many people are interested in Monte Carlo calculations, or similar computations in which the the same procedure is carried out many times, with different inputs, and at the end of the computation a small amount of summarizing is necessary to produce the output.

If the computations need to use random numbers, then one issue to be careful of is to ensure that each MPI process generates a distinct sequence of random numbers. If you understand your system random number generator well enough, you may be able to guarantee this by accessing an internal “seed” value.

Another possibility is to look into the SPRNG package.

---

<http://sprng.fsu.edu>, “The Scalable Parallel Random Number Generators Library”



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- **Your First Six Words in MPI**
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion



## SIX WORDS: Here They Are

You can go far in MPI if you start by learning six fundamental functions:

- **MPI\_Init()**
- **MPI\_Finalize()**
- **MPI\_Comm\_Rank()**
- **MPI\_Comm\_Size()**
- **MPI\_Send()**
- **MPI\_Recv()**

We will start by looking at their C implementations.



## SIX WORDS: MPI\_Init

```
error = MPI_Init ( &argc, &argv );
```

- Input, int **&argc**, address of the program argument counter;
- Input, char **&argv**, the address of the program argument list;
- Output, int **error**, is 1 if an error occurred;

Must be the first MPI routine called.



## SIX WORDS: MPI\_Finalize

```
error = MPI_Finalize ( );
```

- Output, int **error**, is 1 if an error occurred;

Must be the last MPI routine called.



## SIX WORDS: MPI\_Comm\_Rank

```
error = MPI_Comm_Rank ( communicator, &id );
```

- Input, int **communicator**, usually **MPI\_COMM\_WORLD**;
- Output, int **&id**, returns the ID of this process.
- Output, int **error**, is 1 if an error occurred;

How a process figures out its ID.

$0 \leq \text{ID} \leq \text{P}-1$ .





## SIX WORDS: MPI\_Comm\_Size

```
error = MPI_Comm_Size ( communicator, &p );
```

- Input, int **communicator**, usually **MPI\_COMM\_WORLD**;
- Output, int **&p**, returns the number of processes available;
- Output, int **error**, is 1 if an error occurred;

How a process finds out how many other processes there are.



## SIX WORDS: MPI\_Send

```
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;

This call sends data from one process to another.



## SIX WORDS: MPI\_Recv

```
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;

This call receives data sent from another process.



## SIX WORDS: C++ Functions Versions

A C++ program can get away with calling the C form of the MPI functions. But there is a whole object-oriented version of MPI. The functions and symbols specify MPI:: as their namespace:

- `MPI::Init( argc, argv )`
- `MPI::Finalize()`
- `id = MPI::COMM_WORLD.Get_rank()`
- `p = MPI::COMM_WORLD.Get_Size()`
- `MPI::COMM_WORLD.Send()`
- `MPI::COMM_WORLD.Recv()`

Notice that the “communicator”, which we’ll always assume is **COMM\_WORLD**, shows up as the object name in most of the function calls.



## SIX WORDS: C++ Symbols

The C++ symbolic constants use **MPI::** as their namespace:

- **MPI::FLOAT, MPI::DOUBLE, MPI::INT, MPI::BOOL** (data types)
- **MPI::SUM, MPI::PROD, MPI::MAX, MPI::MIN** (for reduction operations)
- **MPI::ANY\_TAG, MPI::ANY\_SOURCE** (“wildcards” for message receipt)



## SIX WORDS: The “status” variable

Sometimes a process is willing to receive any message from any sender. In that case, the **Recv()** function replaces the tag and source fields by **MPI::ANY\_TAG** and **MPI::ANY\_SOURCE**. Once you have actually received a message, you want to know who sent it, and what it was, and that’s what the **status** variable does.

In C++, a variable used for **status** information is declared as:

```
MPI::Status status;
```

Since **status** stores the source and tag of a message that has been received, these can be queried by

```
source = status.Get_source();  
tag = status.Get_tag();
```



## SIX WORDS: FORTRAN Versions

Differences between the C and FORTRAN versions of MPI:

- Most C functions return an integer error code **ierr**. Instead, the corresponding FORTRAN function has one additional subroutine argument called **ierr**;
- the C function **MPI\_Init()** accepts **argv** and **argc** as input arguments; the FORTRAN function does not, and so it has the single argument **ierr**;
- in C, the status variable is a structure; in FORTRAN, it is declared as **integer status(MPI\_STATUS\_SIZE)**; the tag and source values are stored in entries **MPI\_TAG** and **MPI\_SOURCE** of this array.



## SIX WORDS: FORTRAN Versions

There are no interesting differences between the FORTRAN77 and FORTRAN90 versions of MPI, except that

- FORTRAN77 uses **include "mpif.h"** to include MPI definitions;
- FORTRAN90 can use the same include statement, or a **use mpi** module statement;

If all this is clear, I think, we can now skip explicit descriptions of the corresponding FORTRAN MPI functions!





call MPI\_Init ( error )

- Output, integer **error**, is 1 if an error occurred;

Must be the first MPI routine called.



call MPI\_Finalize ( error )

- Output, integer **error**, is 1 if an error occurred;

Must be the last MPI routine called.



## SIX WORDS: MPI\_Comm\_Rank

call MPI\_Comm\_Rank ( communicator, id, error )

- Input, integer **communicator**, set this to **MPI\_COMM\_WORLD**;
- Output, integer **id**, returns the ID of this process.
- Output, integer **error**, is 1 if an error occurred;

This is how a process figures out its ID.



## SIX WORDS: MPI\_Comm\_Size

call MPI\_Comm\_Size ( communicator, p, error )

- Input, integer **communicator**, set this to **MPI\_COMM\_WORLD**;
- Output, integer **p**, returns the number of processes available;
- Output, integer **error**, is 1 if an error occurred;

How a process finds out how many other processes there are.



## SIX WORDS: MPI\_Send

call MPI\_Send ( data, count, type, to, tag, communicator, error )

- Input, (any type) **data(\*)**, the data;
- 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;

This call sends data from one process to another.



## SIX WORDS: MPI\_Recv

call MPI\_Recv ( data, count, type, from, tag, communicator, status, error )

- Output, (any type) **data**(\*), the data;
- 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;

This call receives data sent from another process.



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- **How Messages Are Sent and Received**
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion



# MESSAGES: Requires a Sender and a Receiver

The main feature of MPI is the use of messages to send data between processes.

There is a family of routines for sending messages, but the simplest is the pair **MPI\_Send** and **MPI\_Recv**.

Two processes must be in a common "communicator group" in order to communicate. This is simply a way for the user to organize processes into sub-groups. All processes can communicate in the shared group known as **MP\_COMM\_WORLD**.

In order for data to be transferred by a message, there must be a sending process that wants to send the data, and a receiving process that expects to receive it.





## MESSAGES: The Message Is Transferred

The sender calls **MPI\_Send**, specifying the data, size, type, tag, and communicator.

The sending process pauses, the data is transferred to a buffer on the receiving computer and MPI prepares to deliver it to the receiving process.

The receiving process must be expecting to receive a message, that is, it must execute a call to **MPI\_Recv** and be waiting for a message. The message it receives must correspond in size, type, tag, and communicator.

If so, the information is copied from the buffer into a designated memory location. The message has made it, and the receiver can move to the next instruction.

The sending process, still waiting, receives a confirmation of delivery, and it can proceed as well.



# MESSAGES: Information in Error and Status Variables

If an error occurs during the message transfer, both the sender and receiver return a nonzero flag value, either as the function value (in C and C++) or in the final **ierr** argument in the FORTRAN version of the MPI routines.

When the receiving process finishes the call to **MPI\_Recv**, the extra parameter **status** includes information about the message transfer.

The status variable is not usually of interest with simple **Send/Recv** pairs, but for other kinds of message transfers, it can contain important information



# MESSAGES: Communication = Synchronization

- 1 The sender process pauses at **MPI\_SEND**;
- 2 The message goes into a buffer on the receiver machine;
- 3 The receiver process does not receive the message until it reaches the corresponding **MPI\_RECV**.
- 4 The receiver process pauses at **MPI\_RECV** until the message has arrived.
- 5 Once the message has been received, the sender and receiver resume execution

Excessive idle time, waiting to receive a message, or to get confirmation that the message was received, can strongly affect the performance of an MPI program.

Good MPI programmers limit the number of messages sent. Also, 1 big message is better than 20 small ones.



# MESSAGES: The Buffer Can Fill Up

The simplest message transmissions involve a **buffer**, an area of memory for storing messages that have not yet been accepted.

MPI is just another piece of software, written by human beings, and is full of choices and compromises. One choice is the size of the buffer, and what happens if it fills up.

By default, if the buffer fills with messages that have not been received, nothing more can happen. No more messages can be sent that require the buffer. A process trying to send a message requiring the buffer will **pause** - waiting for the buffer to empty.

It is possible that all the processes pause because they are trying to send messages to a full buffer, in which case the whole program dies - well, worse, it doesn't die, it becomes a zombie, eating up computer time without any result.

You will not write such programs!



# MESSAGES: 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 the receiver may specify **MPI\_ANY\_SOURCE**.
- 5 **tag**, a user-chosen tag for the message, must match, or the receiver may specify **MPI\_ANY\_TAG**.
- 6 **comm**, the name of the communicator, must match (for us, always **MPI\_COMM\_WORLD**)



## MESSAGES: How STATUS Can Be Useful

By the way, if the **MPI\_RECV** allows a “wildcard” source by specifying **MPI\_ANY\_SOURCE** or a wildcard tag by specifying **MPI\_ANY\_TAG**, then the actual value of the tag or source is included in the **status** variable, and can be retrieved there.

```
source = status(MPI_SOURCE)      FORTRAN  
tag = status(MPI_TAG)
```

```
source = status.(MPI_SOURCE);    C  
tag = status.MPI_TAG);
```

```
source = status.Get_source ( );  C++  
tag = status.Get_tag ( );
```



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- **Prime Sum in C**
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion



## PRIME: Add the Primes in Parallel

Let's do the PRIME SUM problem in MPI. Here we want to add up the prime numbers from 2 to  $N$ .

Each of  $P$  processes will simply take about  $1/P$  of the range of numbers to check, and add up the primes it finds locally.

When it's done, it will send the partial result to process 0.

So processes 1 to  $P$  send a single message (simple) and process 0 has to expect any of  $P-1$  messages total.





# PRIME: Initialization

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

int main ( int argc, char *argv[] )
{
    int i, id, j, n = 1000, n_hi, n_lo;
    int p, prime, total, total_local;
    MPI_Status status;
    double wtime;

    MPI_Init ( &argc, &argv );
    MPI_Comm_size ( MPI_COMM_WORLD, &p );
    MPI_Comm_rank ( MPI_COMM_WORLD, &id );
```



# PRIME: Individual Computation

```
/*
 Determine the subrange [N_LO, N_HI] to be checked by this process.
*/
n_lo = ( ( p - id      ) * 1 + ( id      ) * n ) / p + 1;
n_hi = ( ( p - id - 1 ) * 1 + ( id + 1 ) * n ) / p;

total_local = 0.0;
for ( i = n_lo; i <= n_hi; i++ )
{
/*
 Find out if I is prime.
*/
  prime = 1;
  for ( j = 2; j < i; j++ )
  {
    if ( i % j == 0 )
    {
      prime = 0;
      break;
    }
  }
/*
 If I is prime, add it to the total.
*/
  if ( prime == 1 )
  {
    total_local = total_local + i;
  }
}
```



# PRIME: Combine Results

```
/*  
Workers send their partial result to process 0.  
*/  
if ( id != 0 )  
{  
    MPI_Send ( &total_local, 1, MPI_INT, 0, 1,  
              MPI_COMM_WORLD );  
}  
/*  
Process 0 expects to receive P-1 partial results to sum.  
*/  
else  
{  
    total = total_local;  
    for ( i = 1; i < p; i++ )  
    {  
        MPI_Recv ( &total_local, 1, MPI_INT, MPI_ANY_SOURCE,  
                  1, MPI_COMM_WORLD, &status );  
        total = total + total_local;  
    }  
}  
/*  
Print the total.  
*/  
if ( id == 0 )  
{  
    printf ( " Total is %d\n", total );  
}  
MPI_Finalize ( );  
return 0;  
}
```



# PRIME: Output

PRIME\_SUM - Master process:

Add up the prime numbers from 2 to 1000.

Compiled on Apr 21 2008 at 14:44:07.

The number of processes available is 4.

P0 [ 2, 250] Total = 5830 Time = 0.000137

P2 [ 501, 750] Total = 23147 Time = 0.000507

P3 [ 751, 1000] Total = 31444 Time = 0.000708

P1 [ 251, 500] Total = 15706 Time = 0.000367

The total sum is 76127



# PRIME: An Example of a Reduction Operation

Having all the processes compute partial results, which then have to be collected together is another example of a reduction operation.

Just as with OpenMP, MPI recognizes this common operation, and has a special function call which can replace all the sending and receiving code we just saw.



# PRIME: Rewrite using MPI\_Reduce

```
MPI_Reduce ( &total_local, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );  
  
if ( id == master )  
{  
    printf ( " Total is %d\n", total );  
}  
MPI_Finalize ( );  
return 0;
```

- **total\_local** is where it's stored on the sending process;
- **total** is where it is to be added on the receiving process;
- **1** is how many items are stored in **total\_local**;
- **MPI\_INT** is the data type;
- **MPI\_SUM** is the reduction operation;
- **0** is the id of the receiving process;



# PRIME: Syntax for MPI\_REDUCE

MPI\_Reduce ( local\_data, reduced\_value, count, type, operation, to, communicator )

- Input, (any type) **local\_data**, the local data;
- Output, (any type) **reduced\_value**, the variable to hold the result;
- Input, int **count**, number of data items;
- Input, int **type**, the data type;
- Input, int **operation**, the reduction operation **MPI\_SUM**, **MPI\_PROD**, **MPI\_MAX**...;
- Input, int **to**, the process ID which collects the local data into the reduced data;
- Input, int **communicator**, MPI\_COMM\_WORLD;



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- **Communication Styles**
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion





# STYLES:

We've seen two common styles of organizing an MPI program:

- **Master/Worker** - process 0 is in charge
  - **Helpful Master**, also helps in work
  - **Lazy Master**, only gives order, collects results
- **Symmetric** - no process is special (except, perhaps, for minor I/O or data collection)
  - **Embarrassingly Parallel**, almost no communication
  - **Coupled Parallel**, the processes communicate during the computation, not just at beginning and end



# STYLES:

The Master/Worker style of programming is a natural way to begin writing parallel programs.

It can be helpful, as an organizational device, to think of one process as being in charge.

Although the data is spread out over all the processes, the Master can take care of collecting results and printing them, of talking to the user, of controlling iterations and so on.

In the **PRIME\_SUM** program, we allowed process 0 to be in charge, and it was a **lazy master**!



# STYLES:

Another advantage of the Master/Worker style of programming is that is easier to think of data communication this way.

In the beginning, the master sends data to the workers. At the end, the master collects data from the workers. So the **MPI\_Send** and **MPI\_Recv** commands are very easy to comprehend.

This may not be the most efficient way to organize communication

- all the processes have to wait for a turn to talk to the master
- it's easier for the master to collect data from the processes in order of ID number, but they might be ready in any order.



## STYLES:

The Symmetric style of programming has a better chance of exploiting the parallelism in a problem, once we are comfortable with the parallel framework.

For instance, it is common to use a master/worker model to do quadrature. It makes sense: the master is there, in part, to decide which subinterval each worker should handle.

But each worker can figure out its subinterval without any help, just based on its own ID.

In the heat equation, there was no special process. In fact, the solution to the problem was **never** collected into one place; it was always distributed among the processes.



# STYLES:

One reason that the symmetric style of programming takes some practice is that symmetry has some strange effects.

In the symmetric style, as soon as you call **MPI\_Send**, to send some data to another process, you are also essentially telling some other process to send data to you!

Aside from being confusing, this sort of communication pattern can set up the deadlock problem we saw in the heat equation.

One way to handle this is to let the odd processes send to the even ones, and then vice versa. It is a simple way to guarantee that there is always both a talker and a listener!



# STYLES:

Another feature that is common to master/worker programming is the assignment of all the work at the beginning.

This means that there are two waves of communication, first the work assignments, and then later, the results.

If the computation involves many tasks, and they vary in difficulty in an irregular way, this method of task assignment might end up in a load imbalance, with one process getting all the hard work.

(This same issue can show up in **OpenMP** programs as well.)



# STYLES:

A **dynamic scheduling** scheme makes the master/worker communication more flexible.

The master divides the computation into many tasks, but initially only assigns part of the work, then enters a *listening loop* in which it waits for a message from any worker.

The result from the worker is collected. If there is more work, the master gives the worker the next task. Otherwise, the master tells the worker to shut down.

When all the tasks have been completed, all the workers have been shut down, and the master shuts down.

Our next example will include an example of dynamic scheduling.



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- **Matrix\*Vector in Fortran77**
- Monte Carlo Integration in C++
- Message Passing Options
- Conclusion





# $M^*v$ : A matrix-vector multiplication problem

We will now consider an example in which matrix-vector multiplication is carried out using MPI.

This is an artificial example, so don't worry about **why** we're going to divide the task up. Concentrate on **how** we do it.

We are going to compute  $A * x = b$ .

We start with the entire matrix **A** and vector **x** sitting on the "master process" (whichever process has lucky number 0).

We need to send *some* of this data to other processes, they carry out their part of the task, and process 0 collects the results back.



Because one process will be special, directing the work, this program will be an example of the “master-workers” model.

Entry  $b_i$  is the dot product of row  $i$  of the matrix with  $x$ :

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

If there were  $\mathbf{N}$  workers, each could do one entry of  $b$ .

There are only  $\mathbf{P} \ll \mathbf{N}$  processes available, and only  $\mathbf{P}-1$  can be workers, (our master is “lazy”) so we’ll do the job in batches.



Give all the workers a copy of  $x$ .

Then send row  $i$  of  $A$  to process  $i$ .

When process  $i$  returns  $b_i$ , send the next available row of  $A$ ,

The way we are setting up this algorithm allows processes to finish their work in any order. This approach is flexible.

In consequence, the master process doesn't know which process will be sending a response. It has to keep careful track of what data comes in, and when everything is done.



## M\*v: A Loose Master/Worker Model

In a master-worker model, you can really see how an MPI program, which is supposed to be a single program running on all machines, can end up looking more like *two* programs.

We divide the work up based entirely on the ID number.

This is also an unusual example because of the way the master program doesn't know when the next part of the answer will be computed, or who will compute it. How the master figures out who sent the latest message involves the mysterious **status** vector.



## M\*v: Master Pseudocode

If I am the master:

SEND N to all workers.

SEND X to all workers.

SEND out first batch of rows.

While ( any entries of B not returned )

RECEIVE message, entry ? of B, from process ?.

If ( any rows of A not sent )

SEND row ? of A to process ?.

else

SEND "FINALIZE" message to process ?.

end

end

FINALIZE



## M\*v: Worker Pseudocode

```
else if I am a worker
```

```
    RECEIVE N.
```

```
    RECEIVE X.
```

```
    do
```

```
        RECEIVE message.
```

```
        if ( message is "FINALIZE" ) then
```

```
            FINALIZE
```

```
        else
```

```
            it's a row of A, so compute dot product with X.
```

```
            SEND result to master.
```

```
        end
```

```
    end
```

```
end
```



## M\*v: Using BROADCAST

In some cases, the communication that is to be carried out doesn't involve a pair of processes talking to each other, but rather one process "announcing" some information to all the others.

This is often the case when the program is written using the *master/worker* model, in which case one process, (usually the one with ID 0) is put in charge. It takes care of interacting with the user, doing I/O, collecting results from the other processes, handling reduction operations and so on.

There is a "broadcast" function in MPI that makes it easy for the master process to send information to all other processes.

The single function does both sending and receiving!



## M\*v: C++ version of MPI\_Bcast

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

- Input on sender, Output on receivers, **data**, address of data;
- Input, int **count**, number of data items;
- Input, **type**, the data type;
- 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 (within the same communicator).





## M\*v: FORTRAN version of MPI\_Bcast

call MPI\_Bcast ( data, count, type, from, communicator, error )

- Input on sender, Output on receivers, **data**, address of data;
- Input, integer **count**, number of data items;
- Input, **type**, the data type;
- 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 (within the same communicator).



## $M^*v$ : An example algorithm

Compute  $A * x = b$ .

- a "task" is to multiply one row of  $A$  times  $x$ ;
- we can assign one task to each process. Whenever a process is done, give it another task.
- each process needs a copy of  $x$  at all times; for each task, it needs a copy of the corresponding row of  $A$ .
- process 0 will do no tasks; instead, it will pass out tasks and accept results.



# M\*v: Initial Send of Data

```
    if ( id == 0 )  
        numsent = 0  
c  
c BROADCAST X to all the workers.  
c  
    call MPI_BCAST ( x, cols, MPI_DOUBLE_PRECISION, 0,  
    & MPI_COMM_WORLD, ierr )  
  
c  
c SEND row I to process I with message tag I.  
c  
    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,  
    & i, MPI_COMM_WORLD, ierr )  
  
        numsent = numsent + 1  
  
    end do
```



# M\*v: Receive Results, Send More Work

```
c
c Wait to receive a result back from any processor;
c If more rows to do, send the next one back to that processor.
c
    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)
        row = status(MPI_TAG)
        b(row) = ans
c
c Tell the worker the next row to work on.
c
        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 )
c
c But if no more rows, send a dummy message with tag = "0"
c to tell the worker to shut down.
c
        else
```



# M\*v: Workers Compute Results until Shut Down

```
c
c Workers receive X, then compute dot products until
c done message received
c
  else

    call MPI_BCAST ( x, cols, MPI_DOUBLE_PRECISION, 0,
& MPI_COMM_WORLD, ierr )

90  continue

    call MPI_RECV ( buffer, cols, MPI_DOUBLE_PRECISION, 0,
& 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, 0,
& row, MPI_COMM_WORLD, ierr )

    go to 90

200 continue
```



This example showed how one could distribute assignments to workers, keeping them busy until all the work was done.

Such an approach might be appropriate if the individual tasks varied a lot in difficulty, or if the processes varied in speed, or if for any other reason it was wise to work on the problem in small sections.



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- **Monte Carlo Integration in C++**
- Message Passing Options
- Conclusion



# MC/C++: Initialization

*Omitted some stuff here*

```
# include "mpi.h"

int main ( int argc, char *argv[] )
{
    int id, p
    double q, q_error, q_exact, q_total;
    int sample, sample_num, sample_total, seed
    double wtime, x;

    q_exact = 1.0
    sample_num = 100000
    //
    // Initialize MPI.
    //
    MPI::Init ( argc, argv );
    //
    // Get this processor's ID.
    //
    id = MPI::COMM_WORLD.Get_rank ( );
    //
    // Get the number of processors.
    //
    p = MPI::COMM_WORLD.Get_size ( );

    wtime = MPI::Wtime ( );
```





# MC/C++: Each Process Calculates Q

```
//  
// Each process, from ID=0 to P-1, uses a different seed.  
//  
    seed = 123456789 + id;  
//  
// This part looks the same as the sequential code  
// (but runs with a different SEED).  
//  
    q = 0.0;  
    for ( sample = 1; sample <= sample_num; sample++ )  
    {  
        x = r8_uniform_01 ( &seed );  
        q = q + 3.0 * x * x;  
    }  
    q = q / sample_num;  
  
    cout << " " << sample_num << " " << q << " " << q - q_exact << "\n";  
//  
// Have each process sent results to process 0 for reduction to final result.  
//  
    MPI::COMM_WORLD.Reduce ( &q, &q_total, 1, MPI::DOUBLE, MPI::SUM, 0 );
```



# MC/C++: The Master Process Reports Q\_TOTAL

```
//  
// "Clean up" the result.  
//  
if ( id == 0 )  
{  
    q_total = q_total / ( double ) ( p );  
    q_error = q_total - q_exact;  
  
    cout << " " << q_total << " " << q_error << "\n";  
  
    wtime = MPI::Wtime ( ) - wtime;  
  
    cout << " Elapsed seconds = " << wtime << "\n";  
}  
//  
// Shut down MPI.  
//  
MPI::Finalize ( );  
  
return 0;  
}
```



# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- **Message Passing Options**
- Conclusion



## OPTIONS: Avoiding Simple Deadlock

In the heat equation example, pairs of processes exchange data.

For instance, process 6 wants to send its  $H[N]$  to process 7 (which will store it locally as  $H[0]$ ).

At the same time, process 7 wants to send its  $H[1]$  to process 6 (which will store it locally as  $H[N+1]$ ).

So processes 0 through  $P-2$  send  $H[N]$  to their right, while processes 1 through  $P-2$  receive  $H[0]$  from their left.

Processes 1 through  $P-2$  send  $H[0]$  to their left, while processes 0 through  $P-2$  receive  $H[N+1]$  from their right.



# OPTIONS: Avoiding Simple Deadlock

```
for ( j = 1; j <= j_max; j++ )
{
    time_new = j * time_delta;
    /* 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 );
    }
}
```



## OPTIONS: Avoiding Simple Deadlock

Although this is a natural way to write this exchange, it comes very close to causing deadlock, and is sure to cause delays.

And if you increase the number of processes, the delays will get worse!

The first **MPI\_Send** command puts processes 1 through **P-1** in the "send" state. They can't do anything until their messages have been received.

In particular, they can't receive messages. Luckily, process 0 doesn't have a neighbor to the left, so didn't send a message, and so can receive one.



## OPTIONS: Avoiding Simple Deadlock

Hence, instead of all the messages being sent simultaneously, we have the following sequential activity:

```
Process 0 acknowledges message from process 1,  
  THEN process 1 acknowledges message from process 2.  
    THEN ...  
      THEN process P-1 acknowledges message from process P.
```

Each acknowledgement must wait its turn, and as the number of processes increases, the wait grows as well.



## OPTIONS: Avoiding Simple Deadlock

As a programmer, you already have the tools to fix this problem. Have the even processes send to the odd processes, and then the other way around.

However, MPI provides a way to carry out this common exchange operation in a way that automatically avoids deadlock, using **MPI\_Sendrecv**.

The function is useful in the general case when pairs of processes have data to exchange.





# OPTIONS: Use MPI\_SendRecv()

```
/*
  Message 1:
  Process ID sends H[N] to Process ID+1 which stores it as H[0];
*/
if ( id < p - 1 )
{
  MPI_Sendrecv ( &h[n], 1, MPI_DOUBLE, id, 1,
                &h[0], 1, MPI_DOUBLE, id+1, 1,
                MPI_COMM_WORLD, status );
}

/*
  Message 2:
  Process ID+1 sends H[1] to process ID which stores it as H[N+1];
*/
MPI_Sendrecv ( &h[n+1], 1, MPI_DOUBLE, id+1, 2,
              &h[1], 1, MPI_DOUBLE, id, 2,
              MPI_COMM_WORLD, status );
}
```



## OPTIONS: MPI\_Sendrecv

MPI\_Sendrecv ( send\_data, send\_count, send\_type, send\_to, send\_tag, rcv\_data, rcv\_count, rcv\_type, rcv\_from, rcv\_tag, communicator, status)

- **send\_data**, the data to send;
- **send\_count**, the number of data items to send.
- **send\_type**, the type of the data sent;
- **send\_to**, the process to which the data is sent.
- **send\_tag**, a tag for the sent data.
- **rcv\_data**, the data to receive;
- **rcv\_count**, the number of data items to receive.
- **rcv\_type**, the type of the data received;
- **rcv\_to**, the process from which the data is received.
- **rcv\_tag**, a tag for the received data.
- **communicator**, the communicator;
- **status**, the status of the transmission.



## OPTIONS: Non-Blocking Messages

Using **MPI\_Send** and **MPI\_Recv** forces the sender and receiver to pause until the message has been sent and received.

In some cases, you may be able to improve efficiency by letting the sender send the message and proceed immediately to more computations.

On the receiver side, you might also want to declare the receiver ready, but then go immediately to computation while waiting to actually receive.

The non-blocking **MPI\_Isend** and **MPI\_Irecv** allow you to do this. However, the sending routine must not change the data in the array being sent until the data has actually been successfully transmitted. The receiver cannot try to use the data until it has been received.

This is done by calling **MPI\_Test** or **MPI\_Wait**.



## OPTIONS: Nonblocking Pseudocode

```
if I am the boss
{
  Isend ( X( 1:100) to worker 1, req1 )
  Isend ( X(101:200) to worker 2, req2 )
  Isend ( X(201:300) to worker 3, req3 )
  Irecv ( fx1 from worker1, req4 )
  Irecv ( fx2 from worker2, req5 )
  Irecv ( fx3 from worker3, req6 )

  while ( 1 ) {
    if ( Test ( req1 ) && Test ( req2 ) &&
        Test ( req3 ) && Test ( req4 ) &&
        Test ( req5 ) && Test ( req6 ) )
      break
  }
}
```



## OPTIONS: Nonblocking Pseudocode

```
else if I am a worker
{
  Irecv ( X, from boss, req ) <-- Ready to receive

  set up tables                    <-- work while waiting

  Wait ( req )                     <-- pause til data here.

  Compute fx = fun(X)              <-- X here, go to it.

  Isend ( fx to boss, req )
}
```



MPI\_Irecv ( data, count, type, from, tag, comm, req )

- **data**, the address of the data;
- **count**, number of data items;
- **type**, the data type;
- **from**, the process ID from which data is received;
- **tag**, the message identifier;
- **comm**, the communicator;
- **req**, the request array or structure.



MPI\_Test ( req, flag, status )

**MPI\_Test** reports whether the message associated with **req** has been sent and received.

- **req**, the address of the data;
- **flag**, is returned as TRUE if the sent message was received;
- **status**, the status array or structure.



MPI\_Wait ( req, status )

**MPI\_Wait** waits until the message associated with **req** has been sent and received.

- **req**, the address of the data;
- **status**, the status array or structure.





# Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Heat Equation in C
- Compiling, Linking, Running.
- Monte Carlo Integration in Fortran77
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C
- Communication Styles
- Matrix\*Vector in Fortran77
- Monte Carlo Integration in C++
- Message Passing Options
- **Conclusion**



## CONCLUSION: MPI Features

One of MPI's strongest features is that it is well suited to modern clusters of 100 or 1,000 or 10,000 processors.

A huge amount of memory can be requested, as long as it is distributed.

The main cost of MPI is that you need to rethink your algorithm so that it can be implemented as a collection of almost independent subprograms with limited communication.

In MPI, you are in complete control of what part of your program occurs in what process, and it is your responsibility to transmit any data needed by another process.



## Conclusion: Recall OpenMP

OpenMP is limited by the number of cores on a shared memory processor. The value of 48 on our HPC cluster is a relatively high number right now, although each year this number grows.

OpenMP does not usually require rewriting your code. Instead, you gradually “discover” loops that can be parallelized, mark them with directives, and turn them on with a compiler switch.

OpenMP has the opposite of MPI’s communication problem. OpenMP must be careful that two processes don’t overwrite the same data item with different values. This is why some OpenMP variables become “private”, a little like MPI’s distributed memory.

In OpenMP, parallelism only occurs at the low level of in loops and sections. In MPI, parallelism occurs at the program level.

MPI and OpenMP can be used together; for instance, if each MPI process is running on a separate multicore processor.



# CONCLUSION: Web References

- <http://www-unix.mcs.anl.gov/mpi/>, Argonne Labs;
- <http://www.mpi-forum.org>, the MPI Forum
- <http://www.netlib.org/mpi/>, reports, tests, software;
- <http://www.open-mpi.org>, an open source version of MPI;
- <http://www.nersc.gov/nusers/help/tutorials/mpi/intro>, a tutorial
- [http://people.sc.fsu.edu/~jburkardt/pdf/mpi\\_course.pdf](http://people.sc.fsu.edu/~jburkardt/pdf/mpi_course.pdf), a tutorial
- [http://people.sc.fsu.edu/~jburkardt/presentations/fsu\\_mpi\\_2011.pdf](http://people.sc.fsu.edu/~jburkardt/presentations/fsu_mpi_2011.pdf), these slides
- [http://people.sc.fsu.edu/~jburkardt/presentations/fsu\\_mpi\\_exercises\\_2011.pdf](http://people.sc.fsu.edu/~jburkardt/presentations/fsu_mpi_exercises_2011.pdf), related exercises



## CONCLUSION: Reference Books

- Gropp, **Using MPI**;
- **Mascagni, Srinivasan**, *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**;

