Distributed Memory Programming With MPI

John Burkardt
Interdisciplinary Center for Applied Mathematics &
Information Technology Department
Virginia Tech

.....

Applied Computational Science II
Department of Scientific Computing
Florida State University

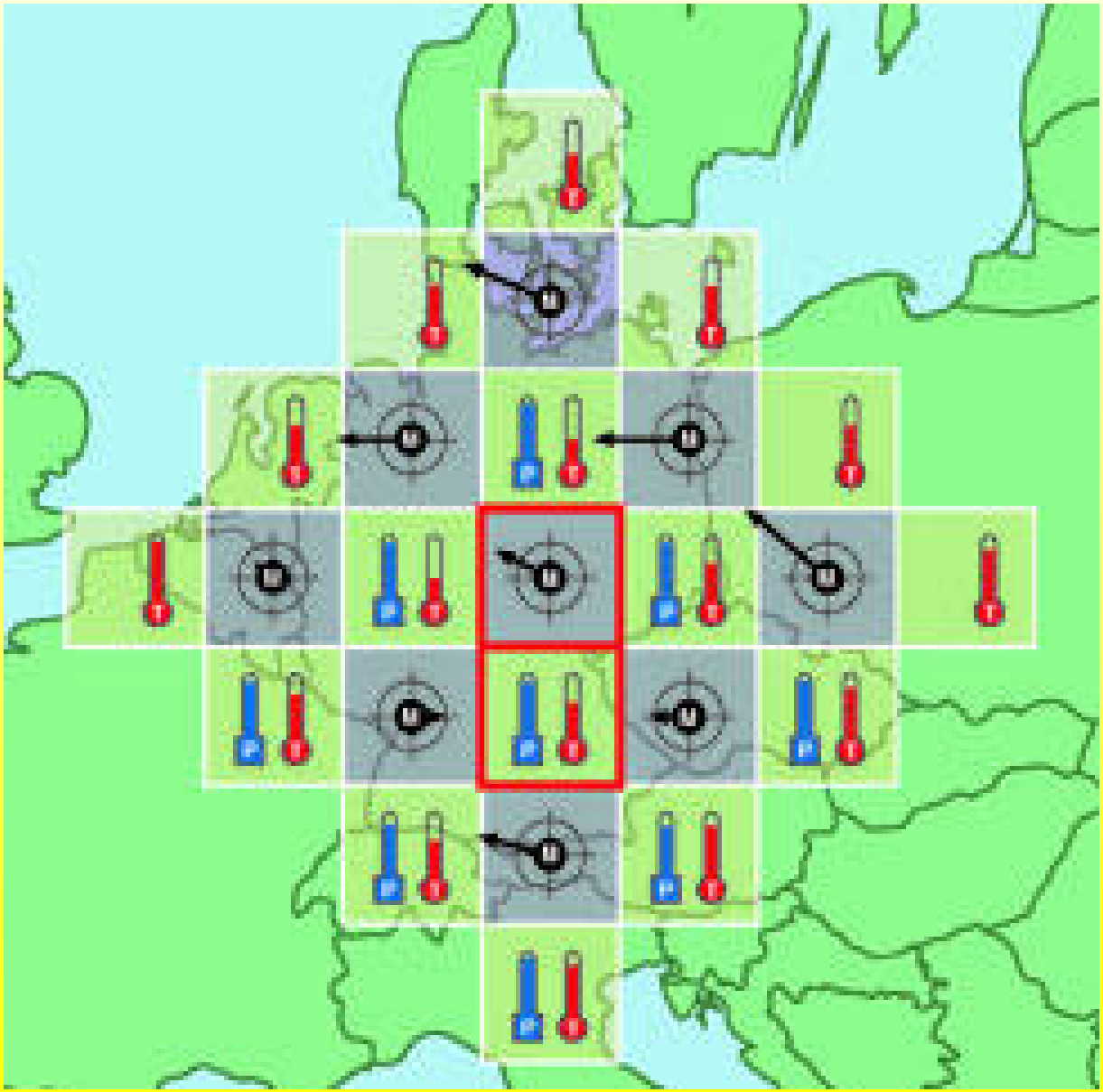
http://people.sc.fsu.edu/∼jburkardt/presentations/...
...fsu_2008_mpi.pdf

16 & 18 September 2008

Distributed Memory Programming With MPI

- **MPI: Why, Where, How?**
- Overview of an MPI computation
- Designing an MPI computation
- The Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

Richardson's Weather Computation for 1917



MPI: Why, Where, How?

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

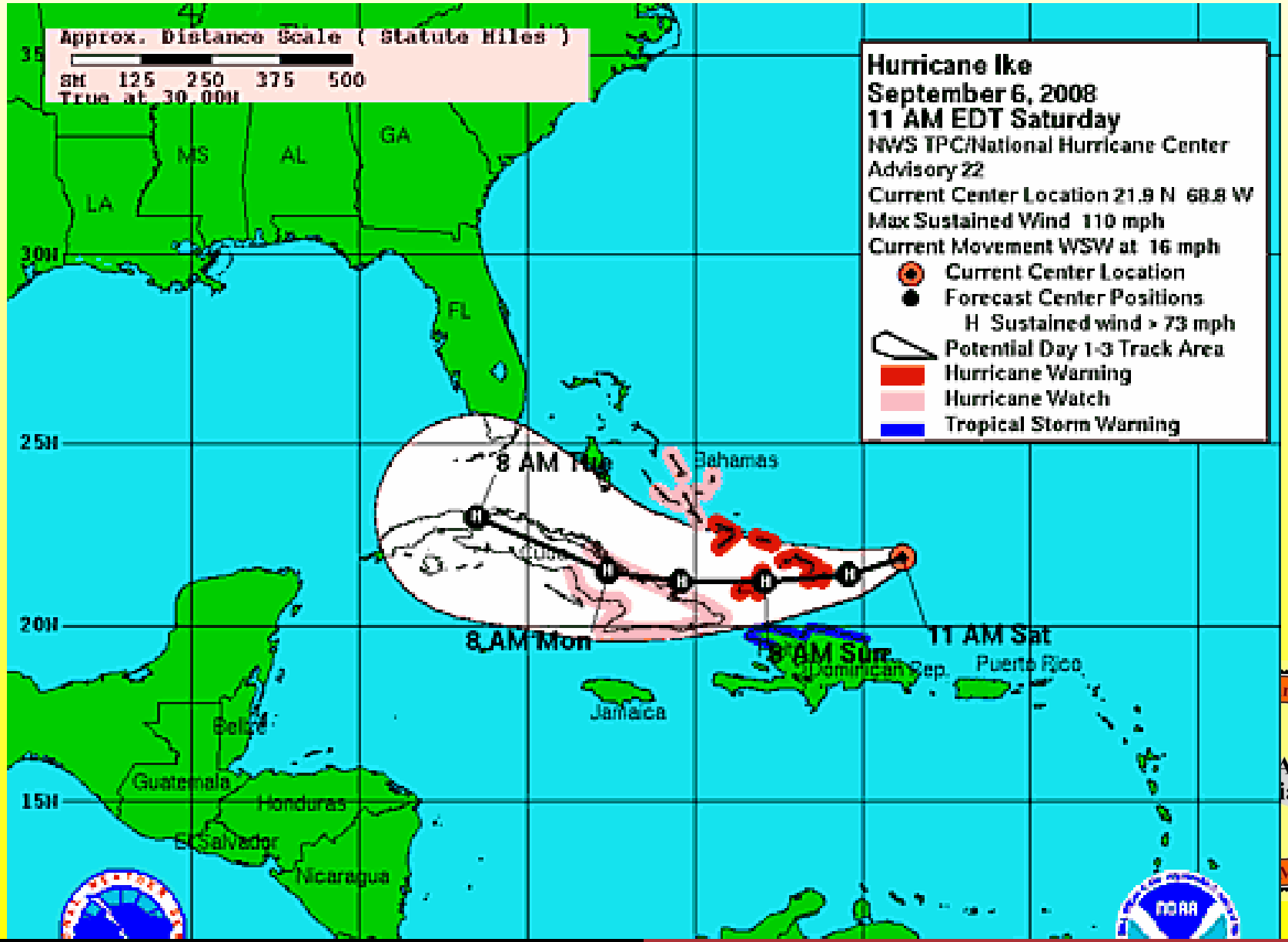
But he believed that with better algorithms and more data, it would be possible to predict the weather reliably.

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.

Weather events like Hurricane Wilma in 2005 ($30 billion in damage) meant accurate weather prediction was worth paying for.

Predicting the Path of Hurricane Ike, 2008



MPI: Why, Where, How?

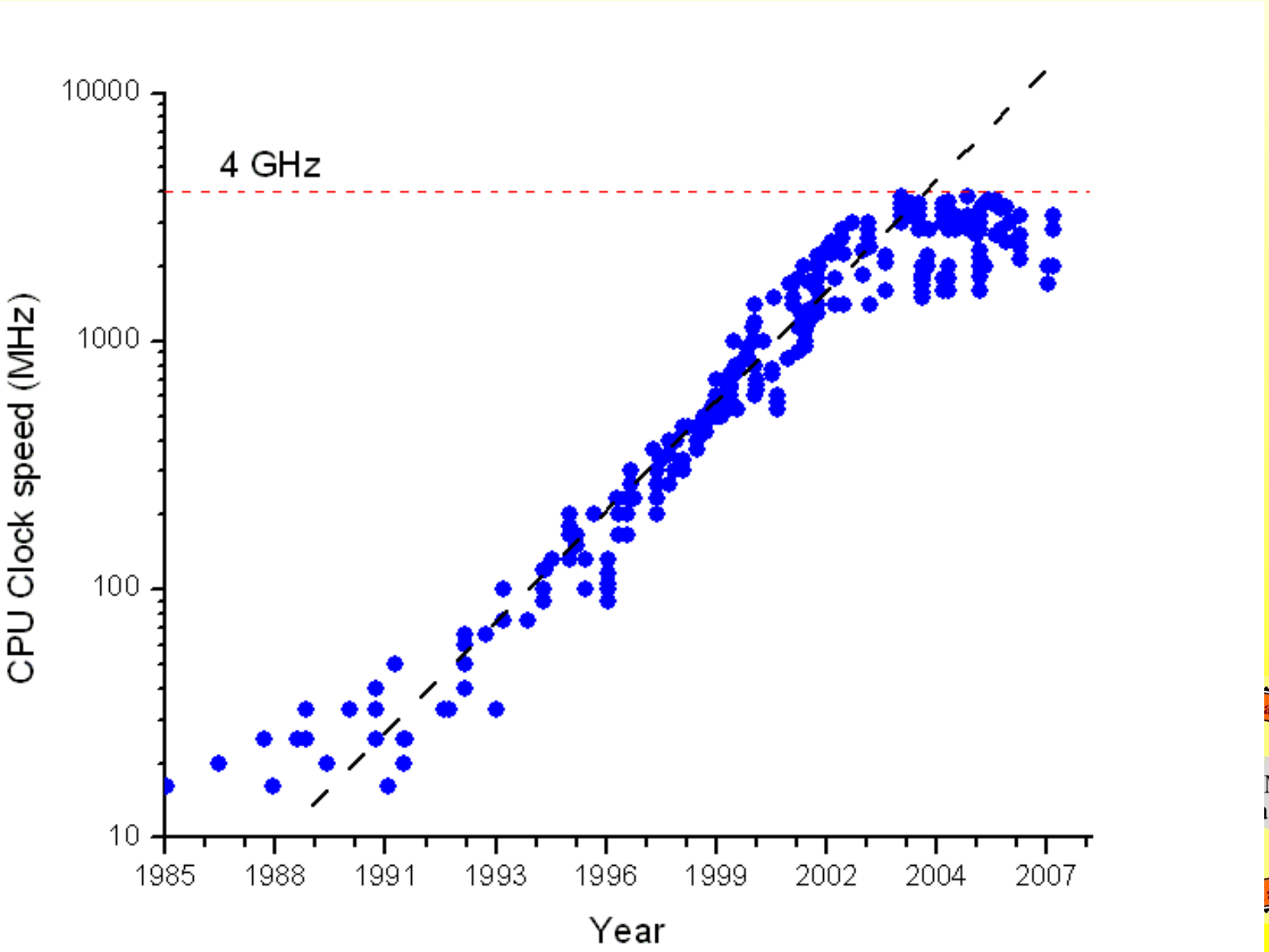
For many years, computer designers could keep up with the need for faster machines by improving current methods - shrinking circuits and making the electronic clock faster.

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: Why, Where, How?



MPI: Why, Where, How?

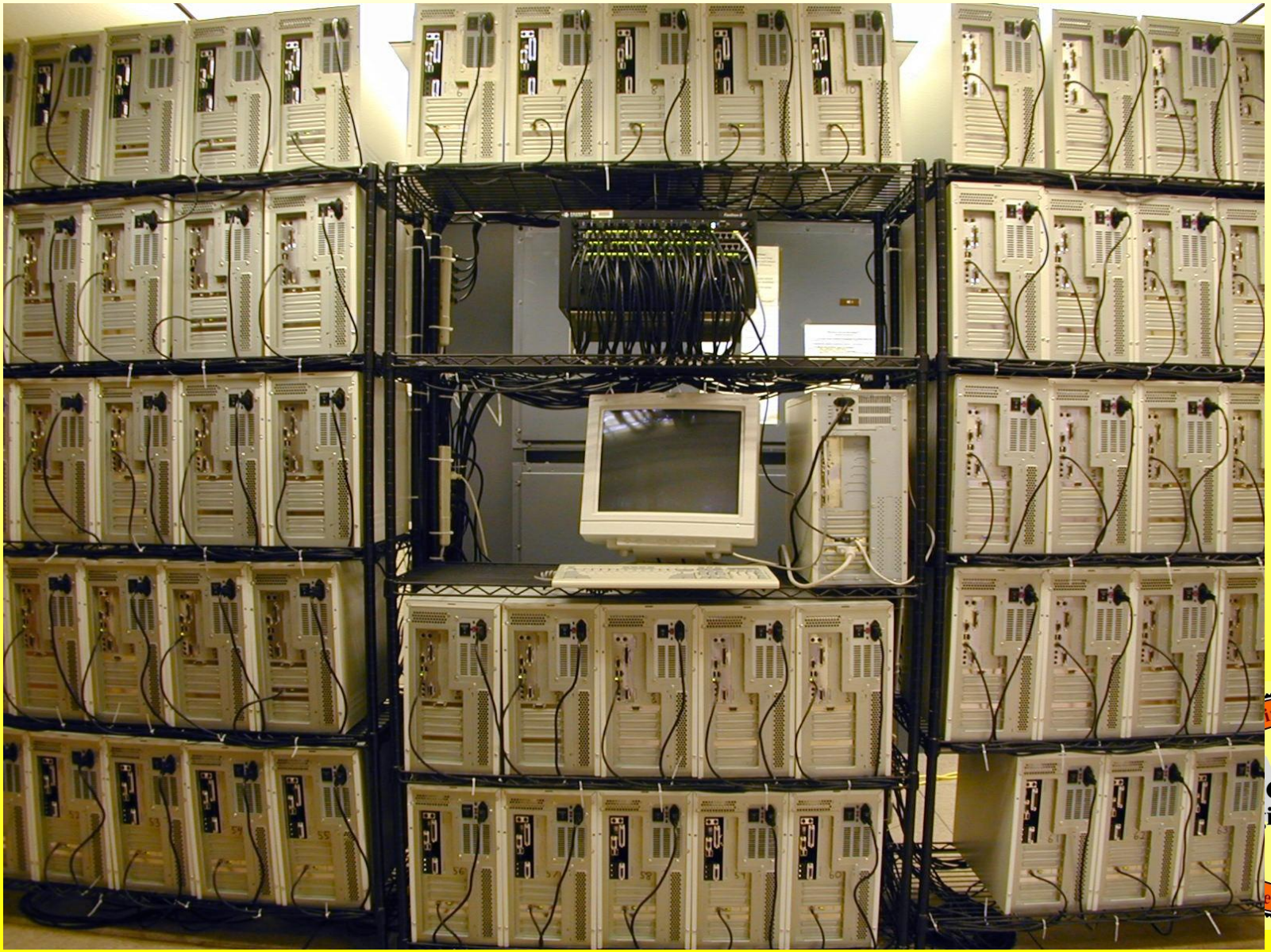
Inter-computer communication had gotten faster and cheaper.

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

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

But where was the conductor?

Cheap, Ugly, Effective



MPI: Why, Where, How?

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

MPI is told the number of computers available, and the program to be run.

MPI then

- distributes a copy of the program to each computer;
- assigns each computer a process id;
- synchronizes the start of the programs;
- transfers *messages* between the processors;
- manages an orderly shutdown of the programs at the end.

MPI: Why, Where, How?

Suppose a user has written a C program *myprog.c* that includes the necessary MPI calls (we'll worry about what those are later!)

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 compiler which adds information about MPI include files and libraries.

MPI: Why, Where, How?

On some systems, the executable 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
```

MPI: Why, Where, How?

Interactive execution may be ruled out if you want to run for a long time, or with a lot of processors.

In that case, you write a job script that describes time limits, input files, and the program to be run.

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

```
msub myprog.sh
```

When your job is completed, two files are returned:

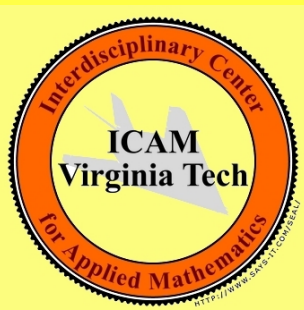
- an *output* file, such as **myprog.o6501**
- an *error* file, such as **myprog.e6501**

(It's easy and simpler to have these two files merged.)

MPI: Why, Where, How?

Since your program ran on **N** computers, you might have some natural concerns:

- **Q:** Do I need to learn a new computer language?
- **A:** *No, Use C/C++/Fortran with an MPI library.*
- **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.*



Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- **Overview of an MPI computation**
- Designing an MPI computation
- The Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

Overview of an MPI Computation

How can parallel processing help you solve a problem?

Suppose that your problem can be replaced by **N** smaller problems, which are not completely independent.

The computers can work simultaneously on the smaller problems, each of which can be solved much faster than the full problem.

Because the small problems are "related" to each other, there will be some communication during the computations.

But the key is that each computer thinks it's just solving a regular problem. We know how to make a computer do that.

The hard part is thinking about how to break down the big problem and put the little solutions together.

Overview of an MPI Computation

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 of an MPI Computation: Heat Equation

Determine the values of $H(x, t)$ over a range $t_0 <= t <= t_1$ and space $x_0 <= x <= x_1$, 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 of an MPI Computation: Heat Equation

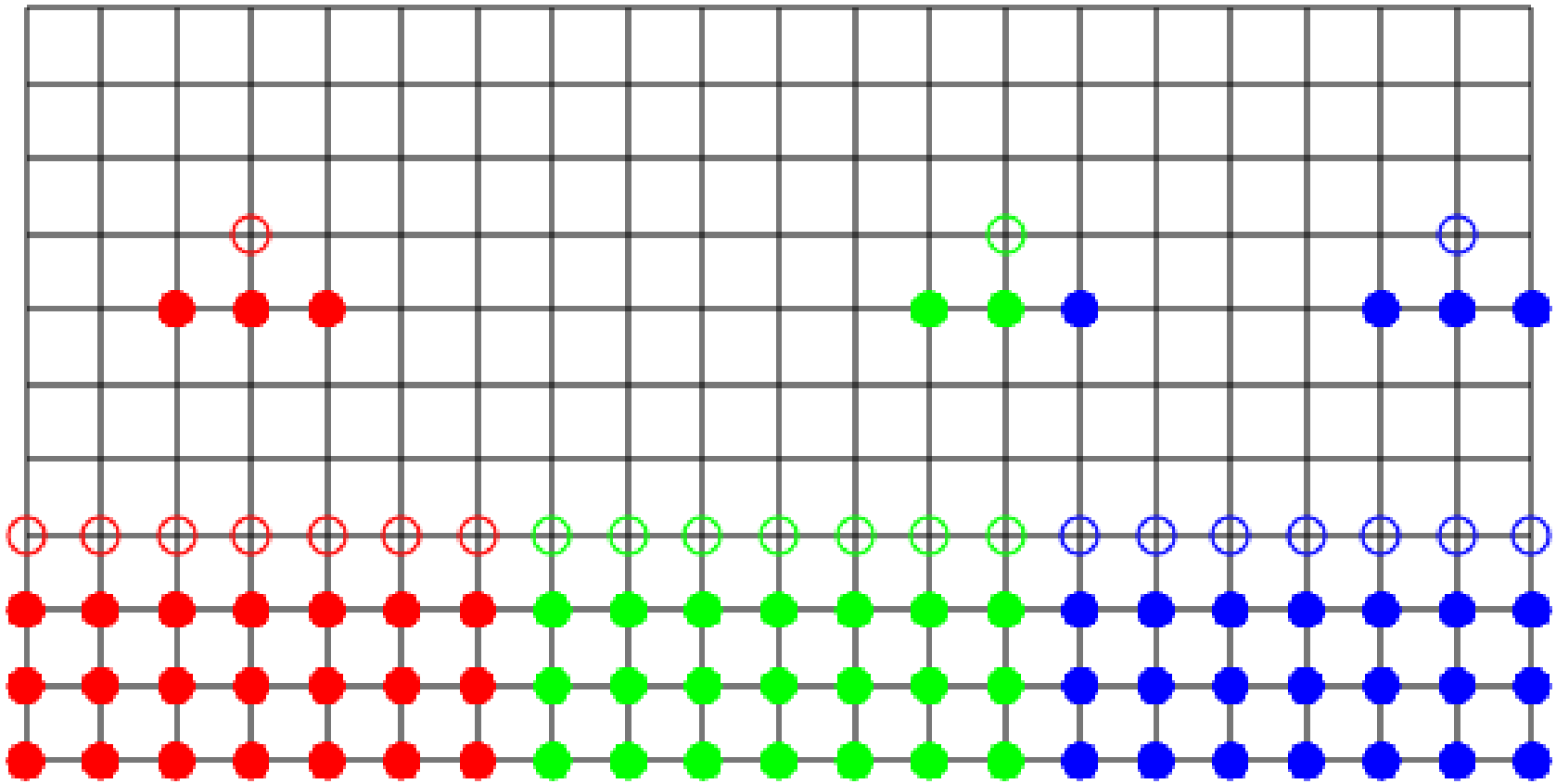
The discrete version of the differential equation is:

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

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 $0 < i < N$.

Overview of an MPI Computation: Heat Equation



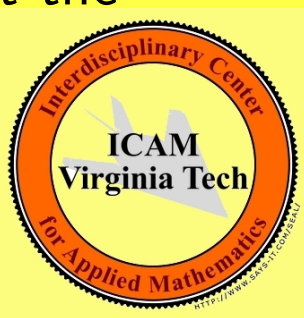
Overview of an MPI Computation: Heat Equation

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

Each processor has a part of the h array.

The **red** processor, 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.



Overview of an MPI Computation: Heat Equation

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 the same idea is true - we can break big problems into smaller ones if we can handle the communication through the boundaries.

Overview of an MPI Computation: Heat Equation

A region of 6,770 elements, subdivided by PMETIS.

Overview of an MPI Computation: Heat Equation

So why does domain decomposition work for us?

Domain decomposition is simply a way to break our big problem up into smaller problems.

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

In order to relate the small problems to our big problem, we need to make sure the interfaces between regions are correctly handled through communication.

Overview of an MPI Computation: Heat Equation

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 computer work with a portion of the database.

One Program Binds Them All

At the next level of abstraction, we have to address the issue of writing one program that all the processors can carry out.

This is going to sound like a Twilight Zone episode.

You've just woken up, and been told to help on the HEAT problem.

You know there are **P** processors on the job.

You look in your wallet and realize your ID number is **ID** (ID numbers run from 0 to **P**-1.

Who do you need to talk to? What do you do?

Overview of an MPI Computation: Heat Equation

Who do you need to talk to?

If your ID is 0, you will need to share some information with your right handneighbor, processor 1.

If your ID is **P**-1, you'll share information with your lefthand neighbor, processor **P**-2.

Otherwise, talk to both **ID**-1 and **ID**+1.

In the communication, you "trade" information about the current value of your solution that touches the border with your neighbor.

You need your neighbor's border value in order to complete the stencil that lets you compute the next set of data.

Overview of an MPI Computation: Heat Equation

What do you do?

We'll redefine **N** to be the number of nodes in our own single program, and store our data in entries **H[1]** through **H[N]**.

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

Our range of X values is $[\frac{ID*N}{P*N-1}, \frac{(ID+1)*N-1}{P*N-1}]$;

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

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

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

Overview of an MPI Computation: Heat Equation

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

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

The data for this problem is truly distributed. No single processor 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.

Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- **Designing an MPI computation**
- The Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

How to Say it in MPI: 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 );
```

Here's where the good stuff goes!

```
  MPI_Finalize ( );
  return 0;
}
```

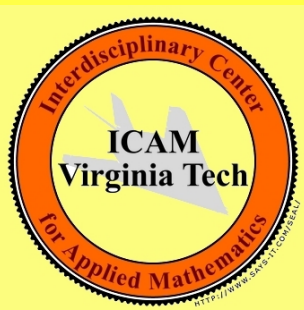
How to Say it in MPI: The "Good Stuff"

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

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

Similarly, **h[n]** gets tossed "to the right" into the ghost slot **h[0]** of the next higher processor.

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!



How to Say it in MPI: The "Good Stuff"

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

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

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

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

How to Say it in MPI: The "Good Stuff"

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

Remember deadlock? *when a process waits for access to a device, or data or a message that will never 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.

How to Say it in MPI: The "Good Stuff"

All processes can use the four node stencil now to compute the updated value of **h**.

Actually, **hnew[1]** in the first process, and **hnew[n]** 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.

How to Say it in MPI: The "Good Stuff"

```
  for ( i = 1; i <= n; 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 ( 0 == id   ) hnew[1] = bc ( x[1], t );
  if ( id == p-1 ) hnew[n] = bc ( x[n], t );

\*  Replace old H by new. *\

  for ( i = 1; i <= n; i++ )  h[i] = hnew[i]
```

Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- **The Source Code**
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

THE SOURCE CODE

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 Equation Source Code (Page 1)

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

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

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

  update ( id, p );

  MPI_Finalize ( );

  return 0;
}
```

Heat Equation Source Code (Page 2)

```
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 Equation Source Code (Page 3)

```
/* Set the X coordinates of the N nodes. */

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

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

  time = time_min;
  h = ( double * ) malloc ( ( n + 2 ) * sizeof ( double ) );
  h_new = ( double * ) malloc ( ( n + 2 ) * sizeof ( double ) );
  h[0] = 0.0;
  for ( i = 1; i <= n; i++ )
  {
    h[i] = initial_condition ( x[i], time );
  }
  h[n+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 Equation Source Code (Page 4)

```
    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[N+1] from ID+1. */

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

      if ( id < p-1 ) {
        tag = 2;
        MPI_Send ( &h[n], 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 Equation Source Code (Page 5)

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

    for ( i = 1; i <= n; 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 ( x[1], time_new );

    if ( id == p - 1 ) h_new[n] = boundary_condition ( x[n], time_new );

/* Update time and temperature. */

    time = time_new;

    for ( i = 1; i <= n; i++ ) h[i] = h_new[i];

/* End of time loop. */
 }
```

Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Source Code
- **Compiling, linking, running.**
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

COMPILING, Linking, Running

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.

COMPILING, Linking, Running

Because it's so nice to do as much initial work on your local machine as possible, there are MPI "stub" libraries available.

These libraries contain dummy routines with the names and arguments of the real MPI library.

It's simple, but enough to fool the loader, that is, the next step after simple compilation. So with a stub library, you would be able to make sure you were calling MPI routines in the right way, perhaps with a command like this:

```
gcc myprog.c mpi_stubs.c
```

If your code can run properly when using only one machine, you may even be able to run the code using the stub library.

We'll see an example in the lab exercises.

COMPILING, Linking, Running

To compile on the HPC machine, transfer the file there using **sftp**, and 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:

```
source /usr/local/profile.d/openmpi-gnu.sh
```

Now, to compile a program, we type:

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

Compiling, LINKING, Running

Linking combines your compiled code with the MPI libraries to make an executable program.

To link a code you have already compiled:

```
mpicc myprog.o
```

To compile and link in one step:

```
mpicc myprog.c
```

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

```
mv a.out myprog
```

Compiling, Linking, RUNNING

Sometimes it is legal to run your program interactively on an MPI machine, if your program is small in time and memory.

Assuming

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

then we can run the program interactively with (say) 4 processors using the command:

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

Compiling, Linking, RUNNING

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.

Here is a script file for the FSU HPC system, called **myprog.sh**

Compiling, Linking, RUNNING

```
#!/bin/bash
#MOAB -N myprog          <-- Name job "myprog"
#MOAB -q backfill        <-- Run job in this queue
#MOAB -l nodes=4:ppn=1   <-- 4 nodes, 1 processor each
#MOAB -l walltime=00:00:30
#MOAB -j oe              <-- join output and error
source /usr/local/profile.d/openmpi-gnu.sh
cd $PBS_O_WORKDIR        <-- move to directory
mpirun -np 4 ./myprog    <-- run with 4 processes
```

Compiling, Linking, RUNNING

The command **-l nodes=4:ppn=1** says to use 4 nodes, and 1 processor per node.

The product **nodes * ppn** is the number of processors we ask for.

Leave **ppn** at 1, and increase **nodes** to get more processors.

To increase **ppn** is to request that the system provide you with processors that are guaranteed to be on the same node. That makes their communication faster, but it's harder for the system to come up with available processors.

The maximum value of **ppn** at the FSU HPC site is 8.

Compiling, Linking, RUNNING

So to use the batch system, you first compile your program, then send the job to be processed:

```
msub myprog.sh
```

The system will accept your job, and report to you a queueing number that can be used to locate the job while it is waiting, and which will be part of the name of the log files at the end.

Compiling, Linking, Running...and WAITING

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

```
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   burkardt   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 burkardt
```

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

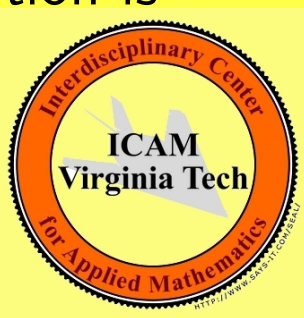
Exercise

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 processors.

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.



Exercise

```
Choose a value for N

Pick a seed for random number generator.

  Set Q to 0

  Do N times:

    Pick a random X in [0,1].

    Q = Q + 3 X^2

  end iteration

Estimate is Q / N
```

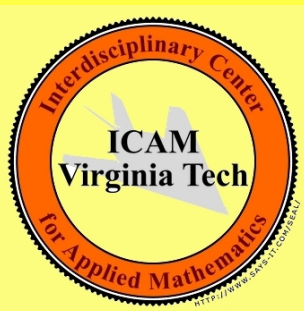
Exercise

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.



Monte Carlo Integration (Page 1)

```
program main

include 'mpif.h'

integer dim_num
parameter ( dim_num = 4 )

double precision f
integer id, ierr, master
parameter ( master = 0 )
integer p
double precision q, q_error, q_exact
parameter ( q_exact = 1.0D+00 )
double precision q_total
integer sample, sample_num
parameter ( sample_num = 1000 )
integer sample_total, seed
double precision wtime, wtime1, wtime2, x(dim_num)

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. master ) then
  wtime1 = MPI_Wtime ( )
end if
```

Monte Carlo Integration (Page 2)

```
c
c  Each process must use a different seed.
c
      seed = 123456789 + id
      q = 0.0D+00
      do sample = 1, sample_num
        call r8vec_uniform_01 ( dim_num, seed, x )
        q = q + f ( dim_num, x )
      end do
      q = q / dble ( sample_num )
      q_error = abs ( q - q_exact )

      write ( *, '(2x,i8,2x,i8,2x,i8,2x,f16.10,2x,g16.6)' )
     &  id, sample_num, dim_num, q, q_error
c
c  Have each process sent results to process MASTER for reduction
c  to final result.
c
      call MPI_Reduce ( q, q_total, 1, MPI_DOUBLE_PRECISION, MPI_SUM,
     &  master, MPI_COMM_WORLD, ierr )
```

Monte Carlo Integration (Page 3)

```
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 )
        sample_total = p * sample_num
        write ( *, '(2x,a8,2x,i8,2x,i8,2x,f16.10,2x,g16.6)' )
     &    '   Total', sample_total, dim_num, q_total, q_error

        wtime2 = MPI_Wtime ( )
        wtime = wtime2 - wtime1
        write ( *, '(a,f14.6)' ) '  Elapsed wall clock seconds = ',
     &    wtime

      end if

      call MPI_Finalize ( ierr )

      stop
      end
```

Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Source Code
- Compiling, linking, running.
- **Your First Six Words in MPI**
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

Your First Six "Words" in MPI

You can write useful programs using the six fundamental routines:

- **MPI_Init**
- **MPI_Finalize**
- **MPI_Comm_Rank**
- **MPI_Comm_Size**
- **MPI_Send**
- **MPI_Recv**

MPI Language Lesson: MPI_Init

MPI_Init (&argc, &argv)

- **&argc**, the address of the program argument counter;
- **&argv**, the address of the program argument list

Must be the first MPI routine called.

MPI Language Lesson: MPI_Finalize

MPI_Finalize ( )

Must be the last MPI routine called.

MPI Language Lesson: MPI_Comm_Rank

MPI_Comm_Rank (communicator, &id)

- **communicator**, set this to **MPI_COMM_WORLD**;
- **&id**, returns the MPI ID of this process.

This is how a processor figures out its ID.

MPI Language Lesson: MPI_Comm_Size

MPI_Comm_Size (communicator, &p)

- **communicator**, set this to **MPI_COMM_WORLD**;
- **&p**, returns the number of processors available.

This is how a processor finds out how many other processors there are.

MPI Language Lesson: MPI_Send

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

- **data**, the address of the data;
- **count**, the number of data items;
- **type**, the data type (**MPI_INT**, **MPI_FLOAT**...);
- **to**, the processor ID to which data is sent;
- **tag**, a message identifier ("0", "1", "1492" etc);
- **communicator**, set this to **MPI_COMM_WORLD**;

MPI Language Lesson: MPI_Recv

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

- **data**, the address of the data;
- **count**, number of data items;
- **type**, the data type (must match what is sent);
- **from**, the processor ID from which data is received (must match the sender, or if don't care, **MPI_ANY_SOURCE**;
- **tag**, the message identifier (must match what is sent, or, if don't care, **MPI_ANY_TAG**);
- **communicator**, (must match what is sent);
- **status**, (auxilliary diagnostic information).

Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- **How Messages Are Sent and Received**
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

How Messages Are Sent and Received

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

There is a family of routines for sending messages, but the simplest is the pair **MPI_Send** and **MPI_Recv**.

Two processors must be in a common "communicator group" in order to communicate. This is simply a way for the user to organize processors into sub-groups. All processors 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 program that wants to send the data, and a receiving program that expects to receive it.

How Messages Are Sent and Received

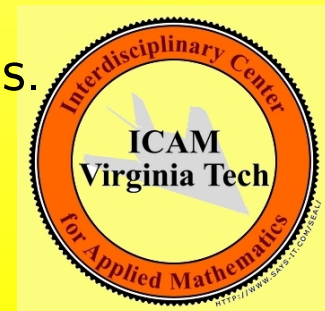
The sender calls **MPI_Send**, specifying the data, an identifier for the message, and the name of the communicator group.

On executing the call to **MPI_Send**, the sending program pauses, the message is transferred to a buffer on the receiving computer system and the MPI system there prepares to deliver it to the receiving program.

The receiving program must be expecting to receive a message, that is, it must execute a call to **MPI_Recv** and be waiting for a response. The message it receives must correspond in size, arithmetic precision, message identifier, and communicator group.

Once the message is received, the receiving process proceeds.

The sending process gets a response that the message was received, and it can proceed as well.



How Messages Are Sent and Received

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 program 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

How Messages Are Sent and Received

1. The sender program pauses at **MPI_SEND**;
2. The message goes into a buffer on the receiver machine;
3. The receiver program does not receive the message until it reaches the corresponding **MPI_RECV**.
4. The receiver program 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.

How Messages Are Sent and Received

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.

NOTHING happens. That is, no more messages can be sent that require the buffer. A program trying to **MPI_Send** more data using the buffer will **pause** - waiting for the buffer to empty.

The buffer has a fixed size. You can send a single message that is too large to fit into the buffer. Your program will go into a coma!

Remedies: Send really big messages in chunks, or use "immediate" (unbuffered) sends and receives: **MPI_ISend** and **MPI_IRecv**.

How Messages Are Sent and Received

```
MPI_Send ( data, count, type, to,   tag, comm )
                    |      |          |     |
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. **from**, must be the process id of the sender, or the receiver may specify **MPI_ANY_SOURCE**.
4. **tag**, a user-chosen ID for the message, must match, or the receiver may specify **MPI_ANY_TAG**.
5. **comm**, the name of the communicator, must match (for us, always **MPI_COMM_WORLD**

How Messages Are Sent and Received

By the way, if the **MPI_RECV** allows a "wildcard" source by specifying **MPI_ANY_SOURCE** or a wildcard tab 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 Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- **Prime Sum in C+MPI**
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

The Prime Sum Example in MPI

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** processors 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 processor 0.

So processors 1 to **P** send a single message (simple) and processor 0 has to expect any of **P**-1 messages total.

Prime Sum Example: Page 1

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

int main ( int argc, char *argv[] )
{
  int i, id, j, master = 0, 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 Sum Example: Page 2

```
n_lo = ( ( p - id     ) * 1 + ( id     ) * n ) / p + 1;
n_hi = ( ( p - id - 1 ) * 1 + ( id + 1 ) * n ) / p;

wtime = MPI_Wtime ( );
total_local = 0.0;
for ( i = n_lo; i <= n_hi; i++ ) {
  prime = 1;
  for ( j = 2; j < i; j++ ) {
    if ( i % j == 0 ) {
      prime = 0;
      break; } }
  if ( prime == 1 )
    total_local = total_local + i;
}
wtime = MPI_Wtime ( ) - wtime;
```

Prime Sum Example Page 3

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

Prime Sum Example: Output

```
n825(0): PRIME_SUM - Master process:
n825(0):   Add up the prime numbers from 2 to 1000.
n825(0):   Compiled on Apr 21 2008 at 14:44:07.
n825(0):
n825(0): The number of processes available is 4.
n825(0):
n825(0): P0 [   2,  250] Total =  5830 Time = 0.000137
n826(2): P2 [ 501,  750] Total = 23147 Time = 0.000507
n826(2): P3 [ 751, 1000] Total = 31444 Time = 0.000708
n825(0): P1 [ 251,  500] Total = 15706 Time = 0.000367
n825(0):
n825(0):         The total sum is 76127

All nodes terminated successfully.
```

The Prime Sum Example in MPI

Having all the processors 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 Sum Example Page 3 REVISED

```
MPI_Reduce ( &total_local, &total, 1, MPI_INT, MPI_SUM,
  master, MPI_COMM_WORLD );

if ( id == master ) printf ( "  Total is %d\n", total );
MPI_Finalize ( );
return 0;
```

MPI Language Lesson: MPI_REDUCE

MPI_Reduce (local_data, reduced_value, count, type, operation, to, communicator)

- **local_data**, the address of the local data;
- **reduced_value**, the address of the variable to hold the result;
- **count**, number of data items;
- **type**, the data type;
- **operation**, the reduction operation **MPI_SUM, MPI_PROD, MPI_MAX...**;
- **to**, the processor ID which collects the local data into the reduced data;
- **communicator**;

Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- **Communication Styles**
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- Conclusion

Communication 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

Communication 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**!

Communication 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.

Communication 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.

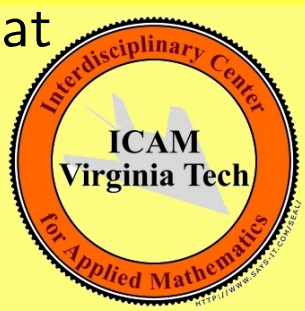
Communication Styles

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!



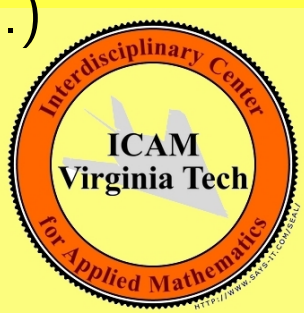
Communication Styles

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 processor getting all the hard work.

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



Communication 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 Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- **Matrix*Vector in Fortran77+MPI**
- Message Passing Options
- Conclusion

Matrix * Vector Example

We will now consider an example in which matrix 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 processor" (whichever processor has lucky number 0).

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

Matrix * Vector Example

Because one processor 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 **N** workers, each could do one entry of b.

There are only **P** << **N** processors available, and only **P-1** can be workers, (our master is "lazy") so we'll do the job in batches.

Matrix * Vector Example

Give all the workers a copy of x.

Then send row i of A to processor i.

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

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

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

Matrix * Vector Example

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.

Matrix * Vector: 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 processor ?.
    If ( any rows of A not sent )
      SEND row ? of A to processor ?.
    else
      SEND "FINALIZE" message to processor ?.
    end
  end
  FINALIZE
```

Matrix * Vector: 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
```

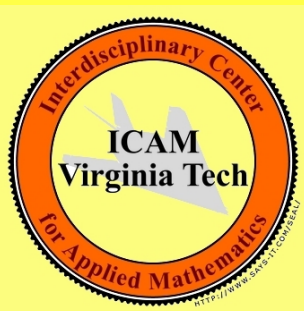
Matrix * Vector: Using BROADCAST

In some cases, the communication that is to be carried out doesn't involve a pair of processors talking to each other, but rather one processor "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 processor, (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 processors, 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 processors.

The single function does both sending and receiving!



MPI Language Lesson: MPI_Bcast

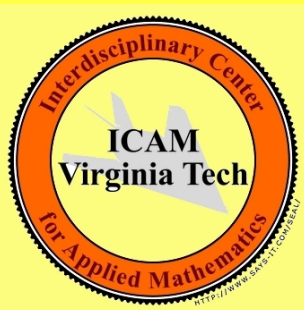
MPI_Bcast (data, count, type, from, communicator)

- **data**, the address of the data;
- **count**, number of data items;
- **type**, the data type;
- **from**, the processor ID which sends the data;
- **communicator**;

Matrix * Vector: 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 processor. Whenever a processor is done, give it another task.
- each processor needs a copy of x at all times; for each task, it needs a copy of the corresponding row of A.
- processor 0 will do no tasks; instead, it will pass out tasks and accept results.



Matrix * Vector in FORTRAN77 (Page 1)

```
      if ( my_id == master )

        numsent = 0
c
c  BROADCAST X to all the workers.
c
        call MPI_BCAST ( x, cols, MPI_DOUBLE_PRECISION, master,
     &    MPI_COMM_WORLD, ierr )

c
c  SEND row I to worker process I; tag the message with the row number.
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
```

Matrix * Vector in FORTRAN77 (Page 2)

```
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)
          anstype = status(MPI_TAG)
          b(anstype) = ans

          if ( numsent .lt. rows ) then

            numsent = numsent + 1

            do j = 1, cols
              buffer(j) = a(numsent,j)
            end do

            call MPI_SEND ( buffer, cols, MPI_DOUBLE_PRECISION,
     &         sender, numsent, MPI_COMM_WORLD, ierr )

          else

            call MPI_SEND ( MPI_BOTTOM, 0, MPI_DOUBLE_PRECISION,
     &        sender, 0, MPI_COMM_WORLD, ierr )

          end if
```

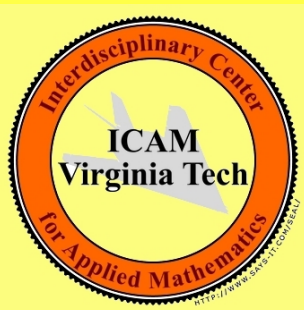
Matrix * Vector in FORTRAN77 (Page 3)

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

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

 90     continue

        call MPI_RECV ( buffer, cols, MPI_DOUBLE_PRECISION, master,
     &    MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr )

        if ( status(MPI_TAG) .eq. 0 ) then
          go to 200
        end if

        row = status(MPI_TAG)

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

        call MPI_SEND ( ans, 1, MPI_DOUBLE_PRECISION, master,
     &     row, MPI_COMM_WORLD, ierr )

        go to 90

 200    continue
```

Matrix * Vector: 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 processor. Whenever a processor is done, give it another task.
- each processor needs a copy of x at all times; for each task, it needs a copy of the corresponding row of A.
- processor 0 will do no tasks; instead, it will pass out tasks and accept results.



Distributed Memory Programming With MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- **Message Passing Options**
- Conclusion

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.

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[N+1] from ID+1. */

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

    if ( id < p-1 ) {
      tag = 2;
      MPI_Send ( &h[n], 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 );
    }
```

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.

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.

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 the **MPI_Sendrecv** function.

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

Avoiding Simple Deadlock

```
/*
  Process ID sends H[N] to Process ID+1 which stores it as H[0];
*/
  if ( id < p - 1 )
  {
    send_tag = 1;
    recv_tag = 1;

    MPI_Sendrecv ( &h[n],   1, MPI_DOUBLE, id,   send_tag,
                   &h[0],   1, MPI_DOUBLE, id+1, recv_tag,
                   MPI_COMM_WORLD, status );
/*
  Process ID+1 sends H[1] to process ID which stores it as H[N+1];
*/
    send_tag = 2;
    recv_tag = 2;

    MPI_Sendrecv ( &h[n+1], 1, MPI_DOUBLE, id+1, send_tag,
                   &h[1],   1, MPI_DOUBLE, id,   recv_tag,
                   MPI_COMM_WORLD, status );
  }
```

MPI Language Lesson: MPI_Sendrecv

MPI_Sendrecv (send_data, send_count, send_type, send_to, send_tag, recv_data, recv_count, recv_type, recv_from, recv_tag, communicator, status)

MPI_Sendrecv exchanges data between pairs of processes.

- **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.
- **recv_data**, the data to receive;
- **recv_count**, the number of data items to receive.
- **recv_type**, the type of the data received;
- **recv_to**, the process from which the data is received.
- **recv_tag**, a tag for the received data.
- **communicator**, the communicator;
- **status**, the status of the transmission.

Non-Blocking Message Passing

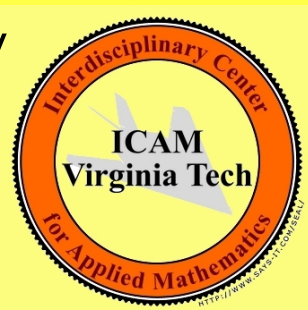
Using **MPI_Send** and **MPI_Recv** forces the sender and receiver to pause until the message has been sent and received.

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**.



Nonblocking Send/Receive 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
  }
```

Nonblocking Send/Receive 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 Language Lesson: MPI_Irecv

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 processor ID from which data is received;
- **tag**, the message identifier;
- **comm**, the communicator;
- **req**, the request array or structure.

MPI Language Lesson: MPI_Test

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 Language Lesson: MPI_Wait

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 Source Code
- Compiling, linking, running.
- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Communication Styles
- Matrix*Vector in Fortran77+MPI
- Message Passing Options
- **Conclusion**

Conclusion

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

In most cases, an MPI implementation of an algorithm is quite different from the serial implementation.

In MPI, communication is explicit, and you have to take care of it. This means you have more control; you also have new kinds of errors and inefficiencies to watch out for.

MPI can be difficult to use when you want tasks of different kinds to be going on.

MPI and OpenMP can be used together; for instance, on a cluster of multicore servers.

References: Web

- **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/∼burkardt/pdf/mpi_course.pdf**, a tutorial

References: Books

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