7: Introductory MPI

John Burkardt
Information Technology Department
Virginia Tech

.....

FDI Summer Track V:
Parallel Programming

.....

https://people.sc.fsu.edu/~jburkardt/presentations/
mpi1_2008_vt.pdf

10-12 June 2008

Introductory MPI

- MPI: Why, Where, How?
- Overview of an MPI computation
- Designing an MPI computation
- The Source Code
- Compiling, linking, running.

MPI: Why, Where, How?

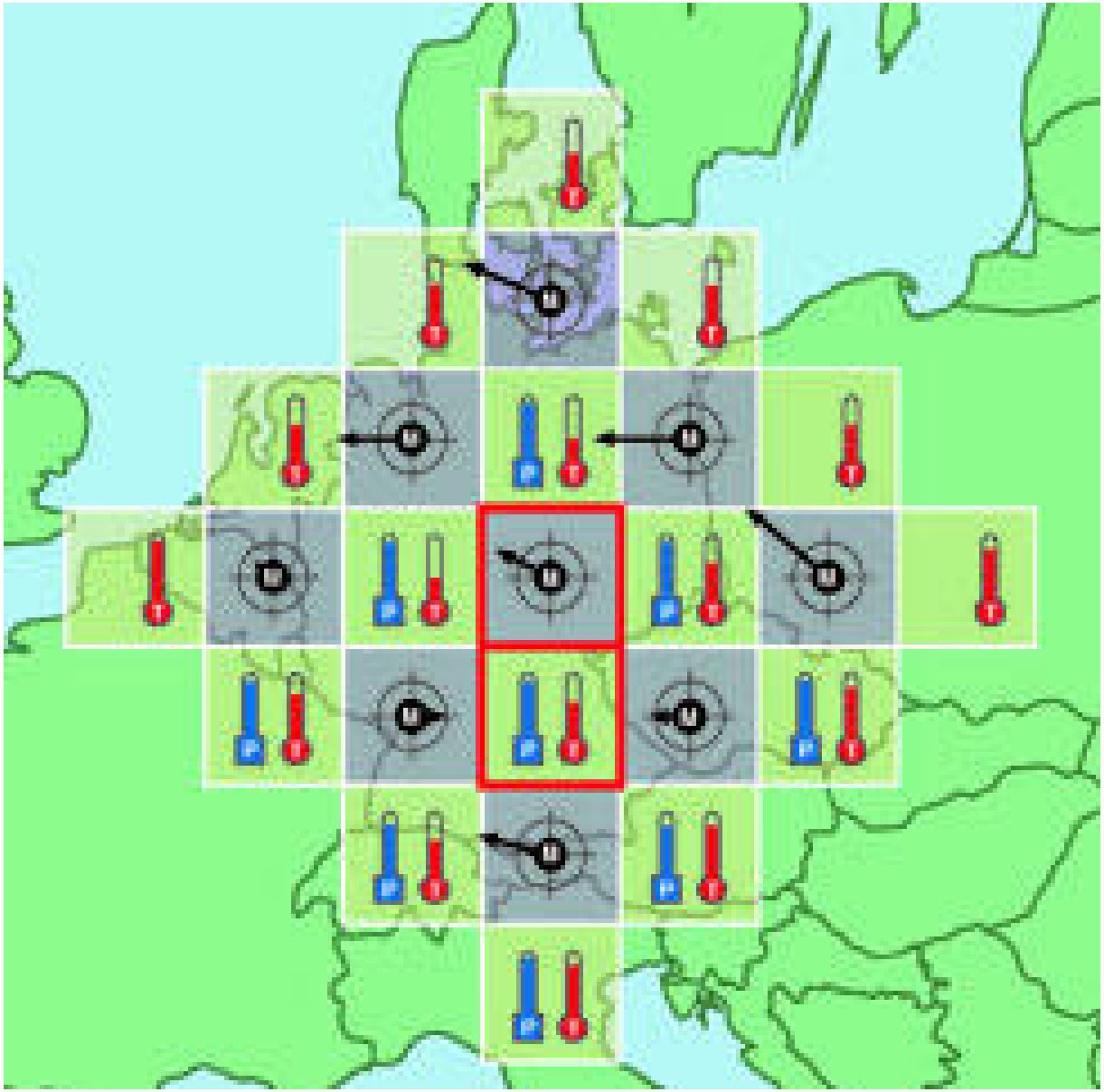
In 1917, Richardson's first efforts to compute a weather prediction were simplistic and mysteriously inaccurate.

Over time, it became clear that weather could be predicted, but that accuracy required huge amounts of data.

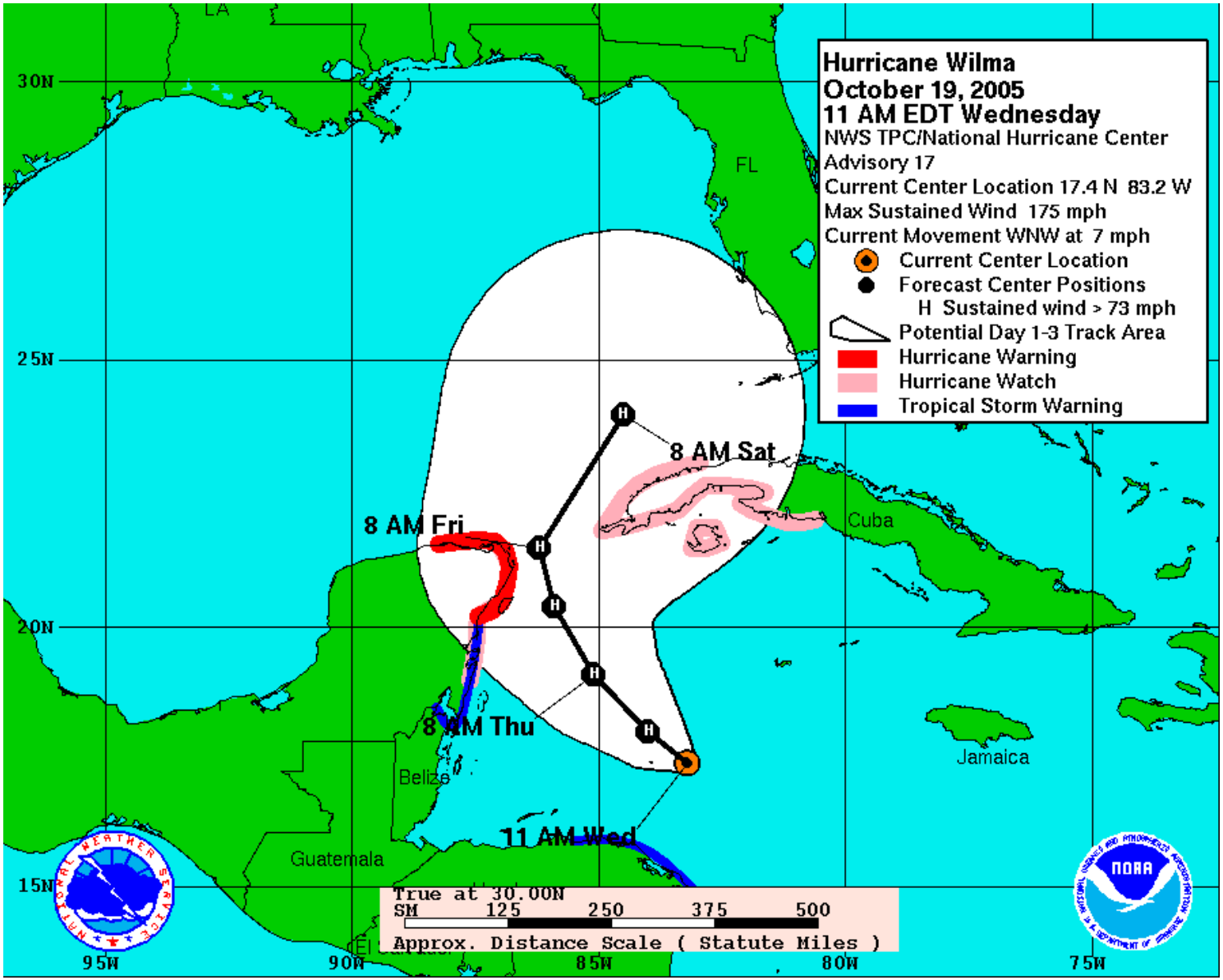
Soon there was so much data that making a prediction 24 hours in advance could take...24 hours of computer time.

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

Richardson's Weather Computation for 1917



Predicting the Path of Hurricane Wilma, 2005



MPI: Why, Where, How?

In the 1990's, the design and maintenance of specially designed supercomputers became unacceptably high.

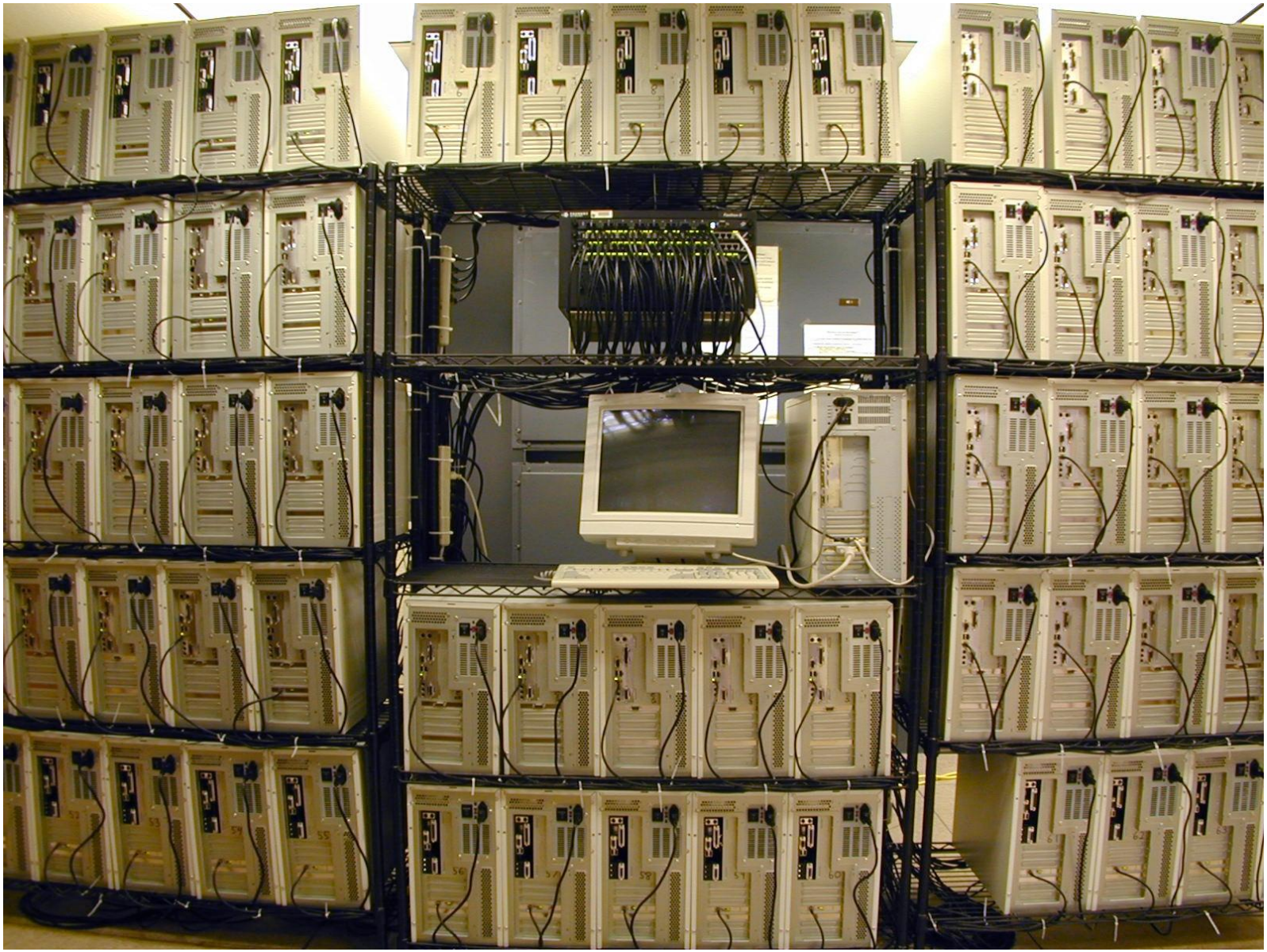
It also seemed to be the case that the performance of commodity chips was not improving fast enough.

However, 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.

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.

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
```

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:

```
qsub myprog.sh
```

When your job is completed, you will be able to access the output file as well as a log file describing how the job ran, on any one of the compute nodes.

MPI: Why, Where, How?

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

- **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.*

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: DE-HEAT

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: DE-HEAT

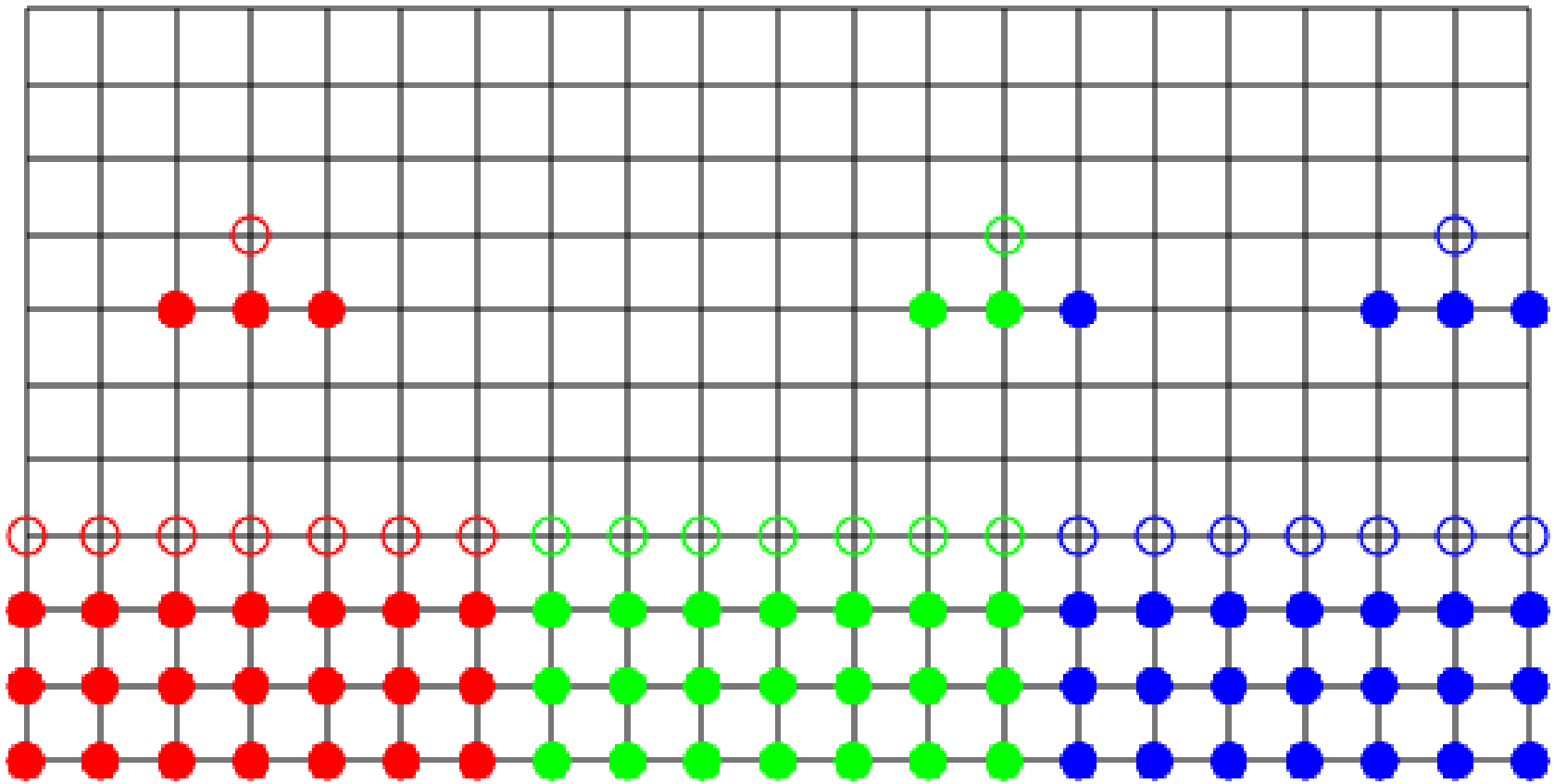
The discrete version of the differential equation is:

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

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: DE-HEAT



Overview of an MPI Computation: DE-HEAT

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.

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: DE-HEAT

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: DE-HEAT

What do you do?

We'll redefine **N** to be the number of nodes in our own single program, so that the total number of nodes is **P*N**.

We'll store our data in entries **H[1]** through **H[N]**.

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

We can figure out our range of X values as $[\frac{ID*N}{P*N-1}, \frac{(ID+1)*N-1}{P*N-1}]$;

To do our share of the work, we must update **H[1]** through **H[N]**.

To update **H[1]**, if **ID**=0, use the boundary condition, else a stencil that includes **H[0]** copied from lefthand neighbor.

To update **H[N]**, if **ID**=**P**-1, use the boundary condition, else a stencil that includes **H[N+1]** copied from righthand neighbor.

Overview of an MPI Computation: DE-HEAT

Some things to notice from our overview.

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.

In this case, the individual program that runs on one computer looks a lot like the sequential program that would solve the whole problem. That's not always how it works, though!

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 fact 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]
```

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. */
 }
```

COMPILING, Linking, Running

Now that we have a source code file, let's go through the process of using it.

The first step is to **compile** the program.

On the MPI machine, a special version of the compiler automatically knows how to include MPI:

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

Compiling on your laptop can be a convenient check for syntax errors. But you may need to find a copy of **mpi.h** for C/C++ or **mpif.h** for FORTRAN and place that in the directory.

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

Compiling, LINKING, Running

We can only **link** on a machine that has the MPI libraries. So let's assume we're on the MPI machine now.

To compile and link in one step:

```
mpicc myprog.c
```

To link a compiled code:

```
mpicc myprog.o
```

Either command creates the MPI executable **a.out**.

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.

Let's give our executable a more memorable name:

```
mv a.out myprog
```

To run interactively with 4 processors:

```
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 System X, called **myprog.sh**

Compiling, Linking, RUNNING

```
#!/bin/bash
#PBS -lwalltime=00:00:30
#PBS -lnodes=2:ppn=2
#PBS -W group_list=???
#PBS -q production_q
#PBS -A $$$
NUM_NODES=‘/bin/cat $PBS_NODEFILE | /usr/bin/wc -l \
  | /usr/bin/sed "s/ //g"‘
cd $PBS_O_WORKDIR
export PATH=/nfs/software/bin:$PATH
jmdrun -printhostname \
  -np $NUM_NODES \
  -hostfile $PBS_NODEFILE \
  ./myprog &> myprog.txt
exit;
```

Compiling, Linking, RUNNING

If you look, you can spot the most important line in this file, which says to run **myprog** and put the output into **myprog.txt**.

The command -**lnodes=2:ppn=2** says to use two nodes, and to use that there are two processors on each node, for a total of four processors. (System X uses dual core chips).

Compiling, Linking, RUNNING

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

```
qsub 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.

If your output does not show up in a reasonable time, you can issue the command **qstat** to see its status.

Compiling, Linking, RUNNING

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

```
qsub 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.

If your output does not show up in a reasonable time, you can issue the command **qstat** to see its status.

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, then the master process can collect all the estimates and average them for a final best estimate. Since this is very little work, we can let the master participate in the computation, as well.

So in the main program, we can isolate ALL the MPI work of initialization, communication (send N, return partial estimate of Q) and wrap up.

This helps to make it clear that an MPI program is really a sequential program...that somehow can communicate with other sequential programs.