MPI Distributed Memory Programming

John Burkardt
Information Technology Department
Virginia Tech

....

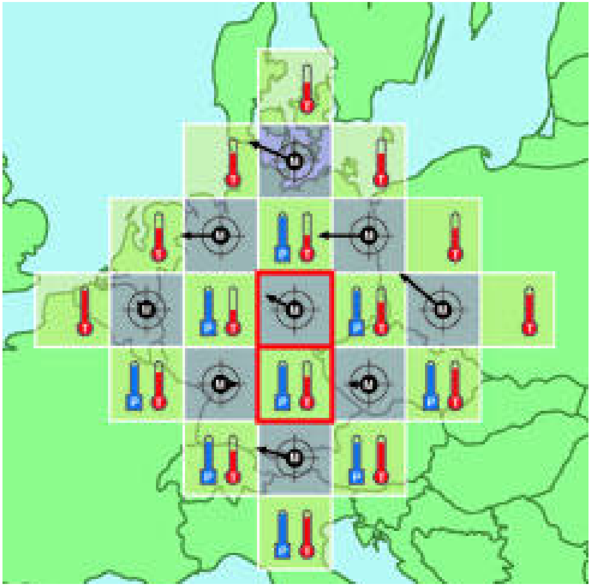
FDI Summer Track V:
Using Virginia Tech High Performance Computing
http://people.sc.fsu.edu/∼jburkardt/presentations/...
mpi_2009_vt.pdf

26-28 May 2009

MPI Distributed Memory Programming

- **Introduction**
- The HELLO Program
- Running HELLO in Batch
- The PRIME SUM Program
- The Logic of the HEAT Program
- Implementing the HEAT Program
- The HEAT Program
- How Messages Are Sent and Received
- Conclusion

Introduction: Weather Computation for 1917



Introduction: The Dream of Weather Forecasting

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

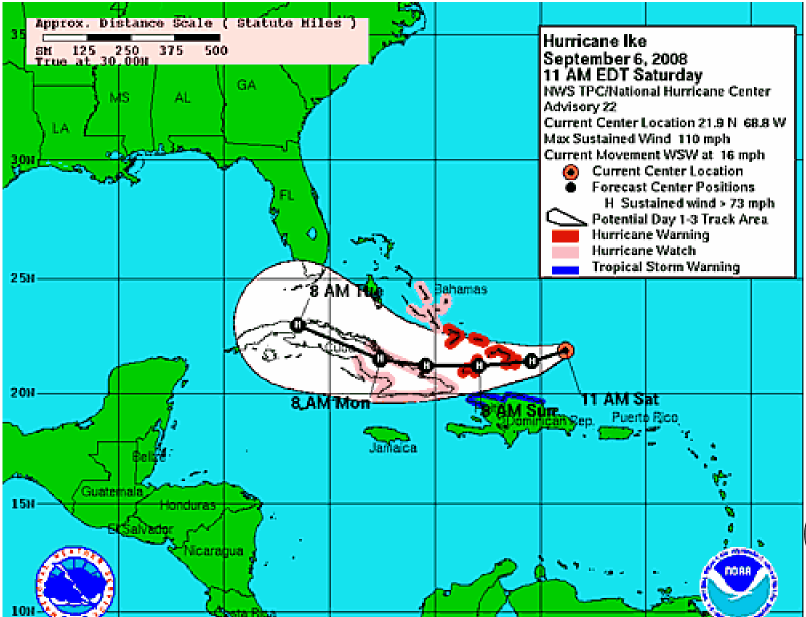
But he believed that with better algorithms and more data, it would be possible to predict the weather reliably.

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.

Introduction: Weather Computation for 2008



Hurricane Ike
September 6, 2008
11 AM EDT Saturday
NWS TPC/National Hurricane Center
Advisory 22
Current Center Location 21.9 N 68.8 W
Max Sustained Wind 110 mph
Current Movement WSW at 16 mph

◉	Current Center Location
●	Forecast Center Positions
	H Sustained wind > 73 mph
	Potential Day 1-3 Track Area
🟥	Hurricane Warning
▨	Hurricane Watch
🟦	Tropical Storm Warning

Introduction: Communication Getting Cheaper

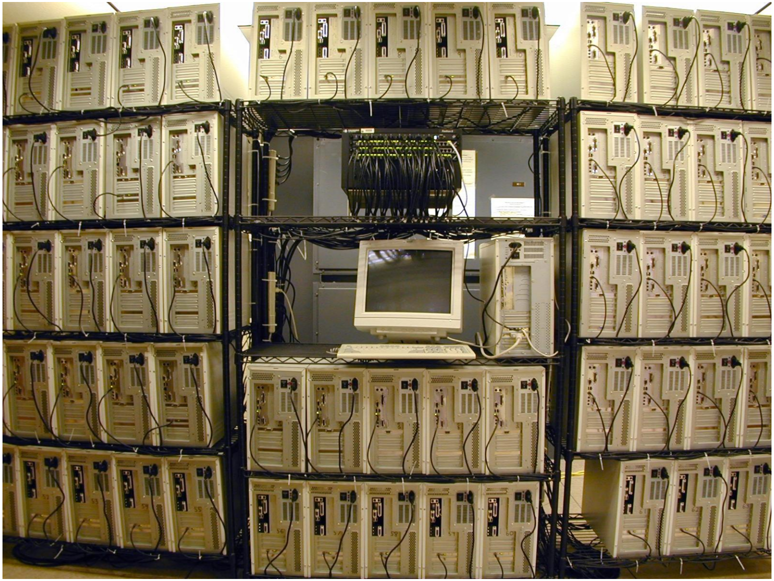
While computer processor clockspeeds hit an upper limit, Inter-computer communication has gotten faster and cheaper.

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

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

But where was the conductor?

Introduction: The Birth of Clusters



Introduction: MPI Manages Cluster Computations

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

MPI needs you to write and compile a “suitable” program.

MPI initializes the computation by copying your program to many processors, giving each a unique ID, and starting them up.

Each copy needs to be smart enough to use its ID to determine what part of the task it will work on.

MPI enables any copy of the program to exchange data with any other copy, using ID numbers for addresses.

At the end, MPI shuts down all the programs, and collects the output into one file.

This basic procedure is the key to MPI.

Introduction: The Programmer's Task

The MPI programmer must distribute the data among the processors, and move some intermediate results from one processor to another, as needed.

The MPI programmer must manage:

- **more computational power** (more MegaFLOPS! That's good!)
- **more memory** (more MegaWords! That's good!)
- **more communication** (This is bad!)

Each communication can cost as much time as 500 computations!

MPI Distributed Memory Programming

- Introduction
- **The HELLO Program**
- Running HELLO in Batch
- The PRIME SUM Program
- The Logic of the HEAT Program
- Implementing the HEAT Program
- The HEAT Program
- How Messages Are Sent and Received
- Conclusion

HELLO

Every programming language introduces itself with a "Hello, world!" program that gives you some idea of how complicated life is going to be!

This simple program is a chance to get a feeling for the things that you will see over and over again in that programming language.

It seems that even just to say "Hello" can be pretty complicated!

HELLO: F90 Code

```
program main

  use mpi
  integer id, ierr, p

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

  write ( *, * ) '  Hello world, from process ', id

  call MPI_Finalize ( ierr )

  stop
end
```

HELLO: MPI_Init & MPI_Finalize

ierr = MPI_Init (&argc, &argv)
subroutine MPI_Init (ierr)

- **argc**, the program argument counter;
- **argv**, the program argument list

ierr = MPI_Finalize ( )
subroutine MPI_Finalize (ierr)

must be the last MPI routine called.

HELLO: MPI_Comm_Rank & MPI_Comm_Size

ierr = MPI_Comm_Rank (communicator, &id)
subroutine MPI_Comm_Rank (communicator, id, ierr)

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

ierr = MPI_Comm_Size (communicator, &p)
subroutine MPI_Comm_Size (communicator, p, ierr)

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

HELLO: C Code

```
# include <stdlib.h>
# include <stdio.h>
# include "mpi.h"
{
  int id;
  int ierr;
  int p;

  ierr = MPI_Init ( &argc, &argv );
  ierr = MPI_Comm_size ( MPI_COMM_WORLD, &p );
  ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &id );
  printf ( "  Process %d says 'Hello, world!'\n", id );
  ierr = MPI_Finalize ( );
  return 0;
}
```

HELLO: Creating an Executable 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 hello.c
mpiCC hello.C   or   mpiCC hello.cc
mpif77 hello.f
mpif90 hello.f90
```

The commands **mpicc**, **mpiCC**, **mpif77** and **mpif90** are customized calls to the compiler which add information about MPI include files and libraries.

Batch jobs

The compile command creates an executable program called **a.out**. It's probably best to rename it using the **mv** command:

```
  mv a.out hello
```

Once you have created the executable program on the cluster, you are *almost* ready to go!

On some computer systems, it is possible at this point to run your MPI program interactively, just by typing its name and some information about how many processes you want to use.

Most systems, including Virginia Tech's System X, require i that your job be put into a queue along with all the jobs requested by other users, so the jobs can be run in an orderly fashion.

This is called the **batch** or **queueing** system.

MPI Distributed Memory Programming

- Introduction
- The HELLO Program
- **Running HELLO in Batch**
- The PRIME SUM Program
- The Logic of the HEAT Program
- Implementing the HEAT Program
- The HEAT Program
- How Messages Are Sent and Received
- Conclusion

Batch jobs: The Job Script File

To run the executable program **hello** on the cluster, you write a job script, which might be called *hello.sh*,

The job script file describes the account information, time limits, the number of processors you want, input files, and the program to be run.

The job script file can look pretty confusing, but the good news is that there are only a few important lines!

Batch jobs: Example Job Script File

```
#!/bin/bash
#PBS -lwalltime=00:00:30
#PBS -lnodes=2:ppn=2
#PBS -W group_list=tcf_user
#PBS -q production_q
#PBS -A hpcb0001

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 -np $NUM_NODES -hostfile $PBS_NODEFILE \
  ./hello

exit;
```

Batch jobs: Important items in job script file

In this job script, the important items are:

- **walltime** lists your job time limit in seconds
- **nodes=2:ppn=2** asks for 2 nodes, and 2 processors per node. Increase the number of nodes for more total processors.
- **hpcb0001** is the account under which you are running.
- **./hello** runs the program; the queueing system saves the output for you
- **./hello &> hello_output.txt** runs your program and saves the output to the file **hello_output.txt**.

Batch jobs: Submit the job script

To run your job, you use the **qsub** command to send your job script file:

You submit the job, perhaps like this:

```
qsub hello.sh
```

The queueing system responds with a short message:

```
111484.queue.tcf-int.vt.edu
```

The important information is your job's ID **111484**.

Batch jobs: Wait for the script to run

Your job probably won't execute immediately. To check on the status of ALL the jobs for everyone, type

```
showq
```

Since the **showq** command lists each job by number and username, you can check for just your job number:

```
showq | grep 111484
```

or only jobs associated with your username (for this class, our usernames are **hpc01**, **hpc02** and so on:

```
showq | grep hpc16
```

Batch jobs: Output files

When your job is done, the queueing system gives you two files:

- an *output* file, such as **hello.o111484**
- an *error* file, such as **hello.e111484**

If your program failed unexpectedly, the error file contains messages explaining the sudden death of your program.

Otherwise, the interesting information is in the output file, which contains all the data which would have appeared on the screen if you'd run the program interactively.

Of course, if your program also writes data files, these simply appear in your home directory when the program is completed.

Batch jobs: Examining the output

Our program output is in **hello.o111484**, or, we we might have redirected the output to **hello_output.txt**.

To see the output, type either:

```
more hello.o111484
more hello_output.txt
```

MPI output from different processes may be “shuffled”:

```
Process 3 says ‘Hello, world!’
Process 1 says ‘Hello, world!’
Process 0 says ‘Hello, world!’
Process 2 says ‘Hello, world!’
```

MPI Distributed Memory Programming

- Introduction
- The HELLO Program
- Running HELLO in Batch
- **The PRIME SUM Program**
- The Logic of the HEAT Program
- Implementing the HEAT Program
- The HEAT Program
- How Messages Are Sent and Received
- Conclusion

PRIME SUM

Our Hello World program was easy and we really did run four copies of the program. But we didn't do any communication!

It's time to do a computation in which the individual programs must cooperate and communicate.

Let's add up the prime numbers from 2 to N.

Each **P** process chooses a subrange to add up, then sends the total to process 0.

PRIME SUM: Initialization

```
# 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: Computation

```
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: Communication

```
    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: 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.
```

PRIME SUM: MPI_Send

ierr = MPI_Send (data, count, type, to, tag, communicator)
subroutine MPI_Send (data, count, type, to, tag, communicator, ierr)

- **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**;

PRIME SUM: MPI_Recv

ierr = MPI_Recv (data, count, type, from, tag, communicator, status)
subroutine MPI_Recv (data, count, type, from, tag, communicator, status, ierr)

- **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).

PRIME SUM: Reduction Operations

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: Communication (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;
```

PRIME SUM: MPI_REDUCE

ierr = MPI_Reduce (local_data, reduced_value, count, type, operation, to, communicator)
subroutine MPI_Reduce (local_data, reduced_value, count, type, operation, to, communicator, ierr)

- **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**;

MPI Distributed Memory Programming

- Introduction
- The HELLO Program
- Running HELLO in Batch
- The PRIME SUM Program
- **The Logic of the HEAT Program**
- Implementing the HEAT Program
- The HEAT Program
- How Messages Are Sent and Received
- Conclusion

Program Logic

In the beginning, we can think about writing a distributed memory program without worrying about the details of arrays and function calls.

We will look at the **logic** involved in planning a computer algorithm, assuming that multiple processors will be available, and that parts of the problem data and solution will be local to a particular processor.

The problem we wish to solve is the equation for the changes over time in temperature along a long wire,

Program Logic: Continuous Heat Equation

Determine the values of $H(x, t)$ over a range $t_0 <= t <= t_1$ and space $x_0 <= x <= x_1$,

We are given:

- the temperature at the starting time, $H(*, t_0)$,
- the temperatures at the ends of the wire, $H(x1, *)$ and $H(x2, *)$,
- a heat source function $f(x, t)$

and a partial differential equation

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

Program Logic: Discrete Heat Equation

The partial differential equation for the function $H(x,t)$

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

becomes a discrete equation for the table of values $H(i,j)$ evaluated at the mesh nodes $(x(i), t(j))$:

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

Program Logic: Step Ahead in Time

A solution procedure simply fills in the missing entries of the array *H*. We start out knowing all the values along the "bottom" (initial time), and the "left" and "right" (the ends of the wire).

The discrete equation can be used to fill in all the missing data.

For instance, knowing H(11,0), H(12,0) and H(13,0), we can "fill in" the value of H(12,1).

```
               H(12,1)
                  ^
                 |||
                 |||
H(11,0)-----H(12,0)-----H(13,0)
```

Program Logic: Distribution and Communication?

If we know all the values in one row of the H table, the discrete equation can be used to determine all the values in the *next* row – except for the first and last entries.

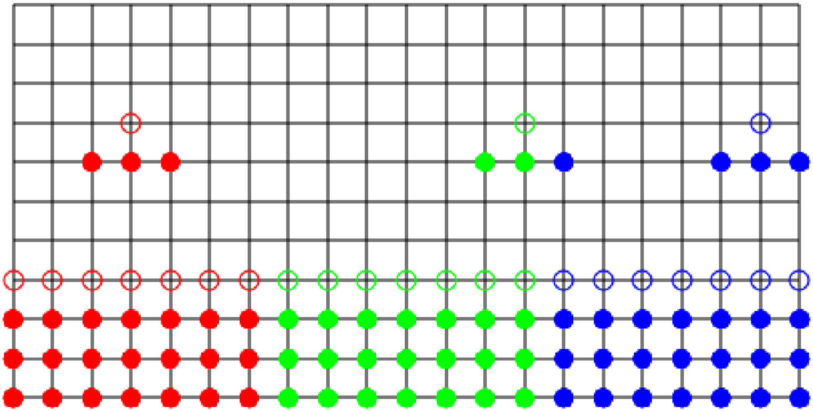
But the first and last values are available as boundary conditions.

Therefore, we can fill in the entire table, one row at a time.

The question to keep in mind is:

Will we be able to rearrange this solution procedure so that it works under MPI, and uses limited communication?

Program Logic: Proposed Layout



Program Logic: How Will Time Stepping Work?

This computation could be done by three processors, which we can call **red**, **green** and **blue**, or perhaps “0”, “1”, and “2”.

Instead of one complete H array of length N (21 for our picture), each process will have a partial array, called h, of length n (7 for our picture).

Each array h will also have entries 0 and n+1, for the left and right immediate neighboring values.

If the two extra values are kept up to date, each process can update all the local values.

Once local values are updated, each process must communicate with its neighbors.

Program Logic: Distributed Memory

H has 21 elements; each process is responsible for 7 of them,
Each process copies entry 0 from the left and 8 from the right.

```
  <-----------------THE H ARRAY------------------------->
  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21


0 1 2 3 4 5 6 7 8    <-- Red's h array
              | |
              0 1 2  3  4  5  6  7  8  <-- Green's h array
                                 |  |
         Blue's h array -->      0  1  2  3  4  5  6  7  8
```

The information in H has been **distributed**.

Program Logic: Imagine You are One Of the Processors

Can we formulate this calculation for the ID-th process out of a total of P processes?

Process ID needs to know if it is the very first or last process, because then it actually has a boundary condition on one side instead of a neighbor.

Process ID must know the value of N, divide that by P, and work out how many values it is responsible for.

Process ID must initialize the h array.

It must update entries h(1) through h(n).

It must send h(1) left, h(n) right, and get updated values of h(0) and h(n+1).

MPI Distributed Memory Programming

- Introduction
- The HELLO Program
- Running HELLO in Batch
- The PRIME SUM Program
- The Logic of the HEAT Program
- **Implementing the HEAT Program**
- The HEAT Program
- How Messages Are Sent and Received
- Conclusion

Implementation: The Basic Calls

```
# 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 computational stuff will be

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

Implementation: The Time Loop

On each time step, a process must do the following:

Exchange data with neighbors to update the "ends" of **h** (that is **h[0]** and **h[n+1]**);

Compute **hnew**, the temperature at the next time, for the "middle" positions of **h**, 1 through n.

Replace the middle values in **h** by the new values from **hnew**.

Implementation: Exchanging data

The exchange of **h** values requires that:

- processes 1 through **P**-1 send **h[1]** "to the left".
- processes 0 through **P**-2 receive these values as **h[n+1]**
- processes 0 through **P**-2 send **h[n]** "to the right"
- processes 1 through **P**-1 receive these values as **h[0]**.

Boundary conditions allow process 0 to sets its **[h0]** and process **P**-1 to set its **h[n+1]**.

Implementation: Exchanging data

Each exchange requires an MPI_Send and a matching MPI_Recv:

```
if ( 0 < id )
  MPI_Send ( id-1:h[n+1] <=== id:   h[1] )

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

if ( id < p-1 )
  MPI_Send ( id:   h[n]    ===> id+1:h[0] )

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

Implementation: The Time Step

Once the communication has been done, (and the first and last process have used their boundary condition information), each process has up-to-date information in **h** with which to compute the "middle" values of **hnew**, the temperature at the next time.

So this part of the computation looks like any ordinary sequential code.

Once **hnew** is computed, we overwrite **h** and we are ready for the next time step.

Implementation: The Time Step

```
  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 ) );

\*  Replace old H by new. *\

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

MPI Distributed Memory Programming

- Introduction
- The HELLO Program
- Running HELLO in Batch
- The PRIME SUM Program
- The Logic of the HEAT Program
- Implementing the HEAT Program
- **The HEAT Program**
- How Messages Are Sent and Received
- Conclusion

The HEAT Program: MPI Basics

```
# 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;
}
```

The HEAT Program: Initialization

```
/* 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 );
```

The HEAT Program: Data Exchange

```
  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 );
    }
```

The HEAT Program: Update

```
/* 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. */
 }
```

The HEAT Program: Utility 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;
}
```

Distributed Memory Programming With MPI

- Introduction
- The HELLO Program
- The PRIME SUM Program
- The Logic of the HEAT Program
- Implementing the HEAT Program
- The HEAT Program
- **How Messages Are Sent and Received**
- 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

```
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 ( );
```

MPI Distributed Memory Programming

- Introduction
- The HELLO Program
- Running HELLO in Batch
- The PRIME SUM Program
- The Logic of the HEAT Program
- Implementing the HEAT Program
- The HEAT Program
- How Messages Are Sent and Received
- **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.