Distributed Memory Programming With MPI

CIS4930/CIS5930: Parallel and Distributed Monte Carlo Methods

.....

John Burkardt
Department of Scientific Computing
Florida State University

http://people.sc.fsu.edu/~jburkardt/presentations/...
...mpi_2012_fsu.pdf

13 July 2012

Distributed Memory Programming With MPI

- **Approximating an Integral**
- MPI and Distributed Computing
- An MPI Program for Integration
- Coding Time!
- Run Time
- Where Do We Go From Here?

EXAMPLE:

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

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

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

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

EXAMPLE:

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

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

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

EXAMPLE:

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

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

The exact value of this integral is 1.

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

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

for which the exact value is also 1.

EXAMPLE: C Version

```
 1    f1 = 0.0;
 2    f2 = 0.0;
 3    for ( i = 0; i < n; i++)
 4    {
 5      for ( j = 0; j < m; j++ )
 6      {
 7        x[j] = (double) rand () / (double) RAND_MAX;
 8      }
 9      value  = f ( m, x );
10      f1 = f1 + value;
11      f2 = f2 + value * value;
12    }
13    f1 = f1 / (double) n;
14    f2 = f2 / (double) ( n - 1 );
15    stdev = sqrt ( f2 - f1 * f1 );
16    sterr = stdev / sqrt ( (double) n );
```

EXAMPLE: Output for M = 2

N	F1	stdev	sterr	error
1	1.2329416	inf	inf	0.232942
10	0.7625974	0.9751	0.3084	0.237403
100	1.0609715	0.8748	0.0875	0.060971
1000	1.0037517	0.8818	0.0279	0.003751
10000	0.9969711	0.8703	0.0087	0.003028
100000	0.9974288	0.8787	0.0028	0.002571
1000000	1.0005395	0.8824	0.0009	0.000539

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

EXAMPLE: Output for M = 10

N	F1	stdev	sterr	error
1	0.0643729	inf	inf	0.935627
10	1.1999289	2.4393	0.7714	0.199929
100	1.2155225	6.2188	0.6219	0.215523
1000	1.2706223	6.2971	0.1991	0.270622
10000	0.9958461	4.4049	0.0440	0.004153
100000	1.0016405	4.3104	0.0136	0.001640
1000000	1.0056709	4.1007	0.0041	0.005670

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

EXAMPLE: Output for M = 20

N	F1	stdev	sterr	error
1	0.0055534	inf	inf	0.99444
10	0.3171449	0.9767	0.3089	0.68285
100	0.2272844	0.9545	0.0954	0.77271
1000	1.7362339	17.6923	0.5595	0.73623
10000	0.7468981	7.7458	0.0775	0.25310
100000	1.0327975	17.8886	0.0566	0.03279
1000000	0.9951882	16.5772	0.0166	0.00481

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

EXAMPLE: Output for M = 20

The Monte Carlo method will converge "eventually".

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

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

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

Distributed Memory Programming With MPI

- Approximating an Integral
- **MPI and Distributed Computing**
- An MPI Program for Integration
- Coding Time!
- Run Time
- Where Do We Go From Here?

PARALLEL: Is Sequential Execution Necessary?

Now let's ask ourselves a peculiar question:

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

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

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

PARALLEL: Is Sequential Execution Necessary?

Look at the computation of F1, our approximate integral:

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

We have to divide by n at the end.

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

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

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

PARALLEL: Is Sequential Execution Necessary?

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

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

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

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

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

PARALLEL: What Tools Do We Need?

To take advantage of parallelism we need:

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

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

PARALLEL: Is Sequential Execution Necessary?

The FSU High Performance Computing (HPC) facility is a cluster with 5,284 cores, using the high speed Infiniband communication protocol.

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

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

PARALLEL: What Must the User Do?

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

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

- Approximating an Integral
- MPI and Distributed Computing
- **An MPI Program for Integration**
- Coding Time!
- Run Time
- Where Do We Go From Here?

MPI: Logical outline of computation

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

Our program outline is:

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

MPI: The Master Program

```
1   SEND value of n/k to all workers
2   RECEIVE f1_part and f2_part from each worker
3   f1 = 0.0;
4   f2 = 0.0;
5   for ( j = 0; j < k; j++ )
6   {
7     f1 = f1 + f1_part[j];
8     f2 = f2 + f2_part[j];
9   }
10  f1 = f1 / (double) n;
11  f2 = f2 / (double) ( n - 1 );
12  stdev = sqrt ( f2 - f1 * f1 );
13  sterr = stdev / sqrt ( (double) n );
```

MPI: The Worker Program

```
1   RECEIVE value of n/k from master
2   f1_part = 0.0;
3   f2_part = 0.0;
4   for ( i = 0; i < n/k; i++)
5   {
6     for ( j = 0; j < m; j++ )
7     {
8       x[j] = (double) rand () / (double) RAND_MAX;
9     }
10    value  = f ( m, x );
11    f1_part = f1_part + value;
12    f2_part = f2_part + value * value;
13  }
14  SEND f1_part and f2_part to master.
```

MPI: Parallelism

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

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

$$\text{Old time} \approx N * \text{one sample}$$

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

MPI: How Many Programs Do You Write?

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

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

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

MPI: One Program Runs Everything

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

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

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

MPI: One Program Runs Everything

```
1   everyone does this line;
2   if ( id == 0 )
3   {
4     only the master does lines in here.
5   }
6   else
7   {
8     each worker does these lines.
9   }
10  something that everyone does.
11  if ( id == 0 )
12  {
13    only the master does this.
14  }
15  if ( id == 2 )
16  {
17    only worker 2 does this.
18  }
```

MPI: Communication Waits

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

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

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

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

Distributed Memory Programming With MPI

- Approximating an Integral
- MPI and Distributed Computing
- An MPI Program for Integration
- **Coding Time!**
- Run Time
- Where Do We Go From Here?

CODE: Accessing the MPI Include File

```
 1 # include <stdio.h>
 2 # include <stdlib.h>
 3 # include <math.h>
 4
 5 # include "mpi.h"     <-- Necessary MPI definitions
 6
 7 double f ( int m, double x[] );
 8
 9 int main ( int argc, char *argv[] )
10 {
11   ...
12   return 0;
13 }
14 double f ( int m, double x[] )
15 {
16   ...
17   return value;
18 }
```

CODE: Initialize and Terminate MPI

```
1  int main ( int argc, char *argv[] )
2  {
3    MPI_Init ( &argc, &argv );
4    ...
5      <-- Now the program can access MPI,
6          and communicate with other processes.
7    ...
8    MPI_Finalize ( );
9    return 0;
10 }
```

CODE: Get Process ID and Number of Processes

```
1  int main ( int argc, char *argv[] )
2  {
3    int id, p;
4    ...
5    MPI_Init ( &argc, &argv );
6    MPI_Comm_rank ( MPI_COMM_WORLD, &id ); <-- id
7    MPI_Comm_size ( MPI_COMM_WORLD, &p );  <-- count
8    ...
9      <-- Where the MPI action will occur.
10   ...
11   MPI_Finalize ( );
12   return 0;
13 }
```

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

```
1  //  The master sets NP = N/(P-1).
2  //  Each worker will do NP steps.
3  //  N is adjusted so it equals NP * ( P - 1 ).
4  //
5    if ( id == 0 )
6    {
7      n = 1000000;
8      np = n / ( p - 1 );
9      n = ( p - 1 ) * np;
10   }
11 //
12 //  The Broadcast command sends the value NP
13 //  from the master to all workers.
14 //
15   MPI_Bcast ( &np, 1, MPI_INT, 0, MPI_COMM_WORLD );
16   ...(more)...
17   return 0;
18 }
```

CODE: Rules for MPI_Bcast

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

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

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

CODE: Examples of MPI_Bcast

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

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

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

CODE: The Workers Work

```
1   f1_part = 0.0;     <-- Even the master does this!
2   f2_part = 0.0;
3   if ( 0 < id )
4   {
5     seed = 12345 + id;  <-- What's going on here?
6     srand ( seed );
7     for ( i = 0; i < np; i++)
8     {
9       for ( j = 0; j < m; j++ )
10      {
11        x[j] = ( double ) rand ( ) / ( double )
              RAND_MAX;
12      }
13      value  = f ( m, x );
14      f1_part = f1_part + value;
15      f2_part = f2_part + value * value;
16    }
17  }
```

CODE: Almost There!

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

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

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

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

CODE: The MPI_Reduce Command

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

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

CODE: Rules for MPI_Reduce

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

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

CODE: using MPI_Reduce()

```
1   //  The master zeros out F1 and F2.
2   //
3     if ( id == 0 )
4     {
5       f1 = 0.0;
6       f2 = 0.0;
7     }
8   //
9   //  The partial results in F1_PART and F2_PART
10  //  are gathered into F1 and F2 on the master
11  //
12    MPI_Reduce ( &f1_part, &f1, 1, MPI_DOUBLE,
        MPI_SUM, 0, MPI_COMM_WORLD );
13
14    MPI_Reduce ( &f2_part, &f2, 1, MPI_DOUBLE,
        MPI_SUM, 0, MPI_COMM_WORLD );
```

CODE: The Master Finishes up

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

```
1   if ( id == 0 )
2   {
3     f1 = f1 / ( double ) n;
4     f2 = f2 / ( double ) ( n - 1 );
5     stdev = sqrt ( f2 - f1 * f1 );
6     sterr = stdev / sqrt ( ( double ) n );
7     error = fabs ( f1 - 1.0 );
8     printf ( "%7d  %.15g  %6.4f  %6.4f  %8g\n",
9       n, f1, stdev, sterr, error );
10  }
11
12  MPI_Finalize ( );
```

http://people.sc.fsu.edu/~jburkardt/latex/fsu_mpi_2012/myprog_mpi.c

CODE: A "Minor" Improvement

When we run our program, what really happens?

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

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

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

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

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

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

```
np = n / p
```

CODE: Comments

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

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

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

Distributed Memory Programming With MPI

- Approximating an Integral
- MPI and Distributed Computing
- An MPI Program for Integration
- Coding Time!
- **Run Time**
- Where Do We Go From Here?

RUN: Installation of MPI

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

- places "mpi.h" where the compiler can find it;
- places the MPI library where the loader can find it;
- installs mpirun, or some other system that synchronizes the startup of the user programs, and allows them to communicate.

Two free implementations of MPI are OpenMPI and MPICH:

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

RUN: Working on Your Laptop

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

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

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

RUN: Working on Your Laptop

If your desktop actually has multiple cores, MPI can take advantage of them.

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

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

```
mpicc -o myprog myprog.c
mpirun -np 4 myprog     <-- -np 4 means run
                            on 4 processors
```

RUN: MPICC is "Really" GCC, or Something

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

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

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

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

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

RUN: The FSU HPC

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

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

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

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

RUN: Working with a Cluster

There are several headaches to put up with when working on a computer cluster.

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

RUN: Compiling on the FSU HPC

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

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

```
module load gnu-openmpi
```

Compile your program:

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

RUN: Executing in Batch on the FSU HPC

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

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

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

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

RUN: A Batch Script for the FSU HPC

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

Commands to MOAB:

```
#MOAB -N myprog             <-- Name is "myprog"
#MOAB -q classroom          <-- Queue is "classroom"
#MOAB -l nodes=1:ppn=4      <-- Limit to 4 processors
#MOAB -l walltime=00:00:30 <-- Limit to 30 seconds
#MOAB -j oe                 <-- Join output and error
```

Define OpenMPI:

```
module load gnu-openmpi
```

Set up and run the job using ordinary interactive commands:

```
cd $PBS_O_WORKDIR       <-- move to directory
mpirun -np 4 ./myprog   <-- run with 4 processes
```

---

http://people.sc.fsu.edu/~jburkardt/latex/fsu_mpi_2012/myprog.sh

RUN: Submitting the Job

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

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

```
msub myprog.sh
```

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

RUN: The Job Waits For its Chance

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

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

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

```
showq | grep jburkardt
```

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

RUN: All Done!

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

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

```
mpirun -np 4 ./myprog  > myprog_output.txt
```

Distributed Memory Programming With MPI

- Approximating an Integral
- MPI and Distributed Computing
- An MPI Program for Integration
- Coding Time!
- Run Time
- **Where Do We Go From Here?**

CONCLUSION:

Our simple integration problem seems to have become very complicated.

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

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

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

CONCLUSION: Write the Main Program Once

```
1  int main ( int argc, char *argv[] )
2  {
3     int id, p;
4     double t;
5     MPI_Init ( &argc, &argv );
6     MPI_Comm_rank ( MPI_COMM_WORLD, &id ); <-- id
7     MPI_Comm_size ( MPI_COMM_WORLD, &p );  <-- count
8     t = MPI_Wtime ( );
9     ...                              Now write a function
10    do_work ( id, p );    <--        that does work of
11    ...                              process id out of p.
12    t = MPI_Wtime ( ) - t;
13    MPI_Finalize ( );
14    return 0;
15 }
```

CONCLUSION:

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

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

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

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

CONCLUSION:

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

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

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

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

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

CONCLUSION: Web References

- **//www-unix.mcs.anl.gov/mpi/**, Argonne Labs;
- **//www.mpi-forum.org**, the MPI Forum
- **//www.netlib.org/mpi/**, reports, tests, software;
- **//www.open-mpi.org** , an open source version of MPI;
- **//www.nersc.gov/nusers/help/tutorials/mpi/intro**
- **//people.sc.fsu.edu/~jburkardt/pdf/mpi_course.pdf**
- **//people.sc.fsu.edu/~jburkardt/presentations/fsu_mpi_2011.pdf**, a different presentation;
- **//people.sc.fsu.edu/~jburkardt/presentations/fsu_mpi_2011_exercises.pdf**, some MPI exercises;
- **//people.sc.fsu.edu/~jburkardt/presentations/fsu_mpi_2012.pdf**, these slides;

CONCLUSION: Reference Books

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