AERE 361

Computational Techniques for Aerospace Design

Laboratory Manual

Spring 2021

Prof. Matthew E. Nelson
Teaching Assistants
Hang Li

Contents

1 Lab 10 - High Performance Computing

1.1 Pre-Lab

Reference:
Introduction to Scientific and Technical Computing:
Chapter 12: Introduction to Parallel Programming
Chapter 13: Introduction OpenMP

1.1.1 Background

We have discussed in lecture that we have different computing tools at our disposal. As far as the personal computer has come though, it is engineered and designed to be a general purpose device. For more serious computing or for tackling the tougher engineering problems, we need a computing device that is dedicated to crunching numbers. These computers have stimulated the rapid growth of a new way of doing science and engineering. The two broad classical branches of theoretical science and experimental science have been joined by computational science. Computational scientists simulate on supercomputers phenomena too complex to be reliably predicted by theory and too dangerous or expensive to be reproduced in the laboratory. Successes in computational science have caused demand for supercomputing resources to rise sharply over the past twenty years.

1.1.2 Objectives

- Gain additional experience with using the HPC-Classroom cluster on campus
- Learn how to parallelize tasks to multiple processors
- Learn how processors can communicate between each other using MPI

1.1.3 Methodology

For this lab we will once again log in and use the ISU HPC-Classroom servers. For instructions on connecting to the HPC-Classroom, refer back to Lab 2. Don't forget that you need to be on campus or you need to use a VPN. As outlined in Lab 2, there are several programs that can be used for this and again, those are outlined in Lab 2.

Warning:
This is one lab that you are **not** able to use Visual Code/Studio, Eclipse, Repl.IT or other IDE. The commands and compiler we have here are specific to the ISU HPC-Classroom.

Deliverable	Points
Hello World	5
Breaker, Breaker Processor 9	10
Quad Function	10
FFT Function	10
Report	70
Total Score	**105**

Table 1.1: Rubric for Lab 10

To Do:
In your lab report, under the methodology section, explain what you used to work on your programming. Don't forget the objectives, which you can copy and past from this write-up. That will conclude the Pre-Lab section of the report.

1.2 Grading

There are 4 exercises in this lab. Follow the instructions given for each exercise. For this lab, we will not have an autograder due to the fact that GitHub doesn't support compiling and running code across multiple processors. Likewise, this will not work in Repl.IT. Instead, we will manually check your source files and like Lab 9, most of your points will be in the report. The grading rubric can be found in Table 1.1

1.2.1 Report

In the report folder you will find a starter "main.tex" file. Like the last lab, this template is very barren as you are expected to fill it out. At this point, you are now expected to read the lab write-up and when asked, fill out the appropriate section.

Be Careful:
Make sure your report compiles. It should not only compile in Overleaf but needs to compile using our LaTeX builder script in GitHub. We will always grade what the GitHub action generates. This happens during a pull request and can be triggered manually.

1.3 Lab Work

1.3.1 Getting Started

To start with, go to the link found in Canvas for this lab. Click on it, and then clone the repo. From here you will need to log in to the HPC-Classroom. Refer to Lab 2 on how to connect with SSH. Once you are logged in, you will want to type in `module load git`. This will make sure you are running the latest version of git on the HPC. Then you can clone your repo in the HPC. Again, please refer to Lab 2 if you have forgotten on how to do this.

Be Careful:
Don't forget that your first step is to create a new branch called "develop". You will do all of your programming work in this branch. If you do log out from the HPC, you will need to type in `module load git` or any of the other modules we asked you to load again.

1.4 Basic MPI Concepts

What is MPI? MPI is a library, not a language. It specifies the names, calling sequences, and results of subroutines to be called from the programs. In the message-passing model of parallel computation, the processes executing in parallel have separate address spaces. Communication occurs when a portion of one process's address space is copied into another process's address space.This operation is cooperative and occurs only when the first process executes a send operations and the second process executes a receive operation.

For the sender, the obvious arguments that must be specified are the data to be communicated and the destination process to which the data is to be sent. The minimal way to describe data is to specify a starting address and a length (in bytes). Any sort of data item might be used to identify the destination; typically it has been an integer.

On the receiver's side, the minimum arguments are the address and length of an area in local memory where the received variable is to be placed, together with a variable to be filled in with the identity of the sender, so that the receiving process can know which process sent it the message.

Although an implementation of this minimum interface might be adequate for some applications, more features usually are needed. One key notion is that of matching: a process must be able to control which messages it receives, by screening them by means of another integer, called the type or tag of the message. Since we are soon going to use "type" for something else altogether, we will use the word "tag" for this argument to be used for matching. A message-passing system is expected to supply queuing capabilities so that a receive operation specifying a tag will complete successfully only when a message sent with a matching tag arrives. This consideration adds the tag as an argument for both sender and receiver. It is also convenient if the source can be specified on a receive operation as an additional screening parameter. Moreover, it is useful for the receive to specify a maximum message size (for messages with a given tag) but allow for shorter messages to arrive. In this case the actual length of the message received needs to be returned in some way. Now our minimal message interface has become:

Listing 1.1: Basic send and receive with MPI

```
1 send(address, length, destination, tag)
2
3 receive(address, length, source, tag, aclength)
```

1.4.1 MPI Communicators

An MPI communicator defines a group of processes for use when running a parallel application. MPI provides a predefined communicator: `MPI_COMM_WORLD`.

which consists of a single group of processes: 0, 1, 2, . . . , p-1, where p is the number of MPI processes being used when running the parallel application. Each process in the group is assigned a rank, i.e. rank 0, rank 1, . . . , rank p-1.

1.5 Hello MPI World!

We will begin our study of a practical parallel computing program by using `MPI_Send` and `MPI_Recv` to write a simple program. The first C program in lab 1 was the `hello world` program. We simply print the message "Hello, World!". Here, we make some use of the multiple processes is to have each process print it on the console. A "Hello World" for MPI is shown in Listing 1.2

Listing 1.2: Example of hello world using MPI

```
1 include <stdio.h>
2 #include <string.h>
3 #include <mpi.h>
```

```
4
5  int main(int argc, char* argv[]){
6
7  int rank;         /* rank of processors*/
8  int p;            /* number of processors*/
9
10 /* Start up MPI */
11 MPI_Init(&argc, &argv);
12
13 /* Find out processor rank*/
14 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16 printf("Hello world from processor %d!\n", rank);
17
18 /* Shut down MPI */
19 MPI_Finalize();
20
21 return 0;
22 }
```

The MPI Library provides a number of useful functions that we can use. Below is a list of just some of the functions we can use and what they do. We look at some of these in more detail later.

- **MPI_Init**, to initialize MPI
- **MPI_Comm_rank**, to determine a process's ID number
- **MPI_Comm_size**, to find the number of processes
- **MPI_Reduce** to perform a reduce operation
- **MPI_Finalize**, to shut down MPI
- **MPI_Barrier**, to perform a barrier synchronization
- **MPI_Wtime**, to determine the time

The underlying hardware is assumed to be a collection of processors, each with its own local memory. A processor has direct access only to the instructions and data stored in its local memory. The interconnection network message passing between processors. Processor A may send a message containing some of the local data values to processor B, giving processor B indirect access to those values.

Let's take a look at the code that is found in Listing 1.2. The program begins with processor directives to include the header files for MPI and standard I/O. Next comes the header for function `main`. Note that we include the `argc` and `argv` parameters, which we will need to pass to the function that initializes MPI.

Function `main` has some variables. Variable `rank` is the processor ID number, and `p` is the number of active processes. Note that if there are `p` processes, then the ID number start at 0 and end at `p-1`.

Each active MPI process executes its own copy of this program. That means each MPI process has its own copy of all of the variables declared in the program, whether they be external variables(declared outside of any function) or automatic variables declared inside a function.

1.5.1 Function `MPI_Init`

The first MPI funciton call made by every MPI process is the call to `MPI_Init`, which allows the system to do setup needed to handle further calls to the MPI library.The only requirement of this function is that `MPI_Init` be called before any other MPI function. Note that All MPI identifiers, including function identifiers, begins with the prefix `MPI_`, followed by a capital letter and a series of lowercase letters and underscores. All MPI constants are strings of capital letters and underscore beginning with MPI_.

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

1.5.2 Function `MPI_Comm_rank` and `MPI_Comm_size`

When MPI has been initialized, every active process becomes a member of a communicator called `MPI_COMM_WORLD`. A communicator is an opaque object that provides the environment for message passing among processes. The `MPI_COMM_WORLD` is the default communicator. For most of the programs it is sufficient. However, we could create our own communicator if we need to partition the processes into independent communication groups.

Processes within a communicator are ordered. The rank of the a `process` is its position in the overall order. In a communicator with `p` processes, each `process` has a unique rank (ID number) between 0 and p-1. A processor may use its rank to determine which portion of a computation and/or a data-set it is responsible for.

A process calls function `MPI_Comm_rank` to determine its rank within a communicator. It calls `MPI_Comm_size` to determine the total number of processes in a communicator.

```
MPI_Comm_rank (MPI_COMM_WORLD,&rank);
```

```
MPI_Comm_size (MPI_COMM_WORLD,&p);
```

1.5.3 Function `MPI_Finalize`

After a process has completed all of its MPI Library calls, it calls function `MPI_Finalize`, allowing the system to free up resources(such as memory) that have been allocated to MPI.

1.6 Compiling and Running MPI Programs

Compiling and running programs that take advantage of this extra hardware requires some extra steps. You can see the steps below:

1. Compile the program using `mpiicc`
2. If needed, debug any errors
3. Use `salloc` to configure a work-space
4. Execute our code using `mpirun`
5. Exit our work-space using `exit`

1.6.1 Compiling

The command to compile an MPI program varies from system to system. On the ISU HPC-Cluster, we start by first loading the Intel Compiler. So far, you have been using the GNU C Compiler (GCC), which is a great and free compiler. But GCC does not have support for multi-processors (not by itself). For this reason we need to use a different compiler and on the HPC-Cluster we have access to the Intel C Compiler (ICC). Let's start by loading this compiler.

```
module load intel
```

Now that we have the Intel compiler we can start using it. With the Intel compiler we now have access to a special version of the compiler that supports multiple processors. This is called MPIICC and is called like what is shown below.

```
mpiicc hello.c
```

Many of the same flags that you have used for GCC will work with Intel's compiler. For example, you can use the `-Wall` flag to enable warnings and additional messages. You can also use `-c` to compile source code into an object file. And, you can use `-o` to specify an output file. Like GCC, if we do not specify the output file, then it will save our executable as `a.out`.

Note:
GCC can be setup to compile code for multi-processors when used with openmpi, an open source library similar to Intel's MPI implantation. Also, with the C11 standard or higher, GCC does support multi-threading (but C99 does not).

1.6.2 Allocating a work-space

At this point you are used to simply running your program. But, not so fast. Recall way back at the beginning of the semester (in Lab 2) we had you run COMSOL and schedule that with the HPC to take advantage of all of the processors we have available. We have to do the same with our program as well. Just because we have it compiled to run with multiprocessors doesn't mean it will. We need to tell the OS that we wish to allocate time on the HPC and then we have a special command to run it. Let's start by allocating our work-space.

With COMSOL, we used what is called a slurm job. This submits the job and schedules it to be run. If there is a cluster available, slurm should start running right away. If not, it will wait until there is one free. We can also let the HPC know that we want to allocate some time now.

That slurm job that we setup actually uses `salloc` to establish a work-space to run the program. This is a special setup which allocates the resources from the HPC for us to use. In much larger HPC clusters, we often have multiple nodes and processors. Jobs can be spread across multiple nodes and processors which means we can divide up a large quantity of tasks to be worked on in parallel. On the HPC-Classroom however, we only have one node. But we do have sixteen processors, we will take advantage of.

Note:
We are going to use both "cores" and "processors" interchangeably. In a modern HPC, a core is essentially a standalone processor and can run a job independently of the other cores. They are often all on the same silicon die though (or there might be 2 physical processors with 8 cores each). In any case, for the purpose of this lab, they are the same thing when allocating resources.

The HPC-Classroom has 55 nodes and each node has two Intel E5 2650 processor. Each processor has 8-cores that run at 2.0 GHz. This means that each node has sixteen cores or we often just refer to these as processors. This means that technically, we could request up to 880 processors! For this lab though we are just going to ask for one node, which still gives us sixteen processors to use. But, if you did have a really large job, then you could of course scale this as needed. That is the power of a HPC, we can get access to multiple processors to crunch through the numbers.

To use `salloc`, we simply tell it the number of nodes (for us just one), the number of processors or cores (for us, up to sixteen) and the maximum time we expect. In this example we will use 10 minutes, but other times can be set. However, you should use the smallest amount of time you think you will need. This is because the slurm manager will prioritize tasks with lower time requirements. Below is an example of using `salloc` on the command line:

```
salloc -N 1 -n 16 -t 0:10:00
```

Note:
It is considered bad practice to request and allocate more than you need. This is one reason we talk about complexity, understanding what your program is going to need allows you to better allocate the correct amount of resources. We are going to be good stewards of this resource and only use one node.

From here you can use `mpirun` (which we will talk about next) to run your program. You can run this multiple times if you want, you don't need to setup another `salloc` since you are in that current work-space.

That is why I refer to this as a work-space. Think about this like reserving time to work on a machine like a CNC. You have that machine for the time you specified and you should come to that prepared so you can do your cutting and then once you are done release the machine so someone else can use it. That is why once you are done running your code, type in `exit`. This will have you leave this work-space and return to the normal Linux OS.

> *Note:*
> Don't forget to exit! When you enter `salloc`, you are in a different work-space. You will want to do your compiles OUTSIDE of this, so once your program is done running, type in exit. It is also the courteous thing to do, when you are done, exit so others can use it.

1.6.3 Running your program

Once we have our work-space setup, we can run our program. The typical command for an MPI program is `mpirun`. The `-np` flag indicates the number of processes to use. Based on what we allocated with `salloc`, we have 16 processors available, so we can run up to that many. We could of course specify a smaller number, but why would we do that when we have 16 processors at our disposal. An example of running our program is below.

```
mpirun -np <number of processes> ./a.out
```

A final note on this. While this allows us to technically run any program, only programs that are designed to run on multiple processors will work correctly. If a program is not designed or compiled to run in a multi-processor environment, then it will only use one processor, even if we allocate to use others. Many computationally heavy programs though, like COMSOL, Matlab, and others can be setup to run in this environment, but need to have extra steps setup. This is even true for applications that run on a desktop machine. They may need a setting enabled or a different version to utilize multiple-cores.

> *Note:*
> Again, don't forget to exit out of the work-space once you are done.

1.7 Introducing point to point Communication and Collective Communication

1.7.1 point to point communication

A **point-to-point communication** involves a pair of processes. In contrast, the collective communications operations involve every process in a group. Figure 1.1 illustrates a point-to-point communication. In the picture, process `h` is not in a communication. It contains executing statements manipulating its local variables. Process `i` performs local computations, then sends a message to process `j`. After the message is sent, it continues on with its computation. Process `j` performs local computations, then blocks until it receives a message from process `i`.

In order for the execution of MPI functions calls to be limited to a subset of the processes, these calls are normally inside conditionally executed code.

1.7.2 Function `MPI_Send`

The sending process calls function `MPI_Send`:

Listing 1.3: Example of point to point send

```
1 int MPI_Send(
```

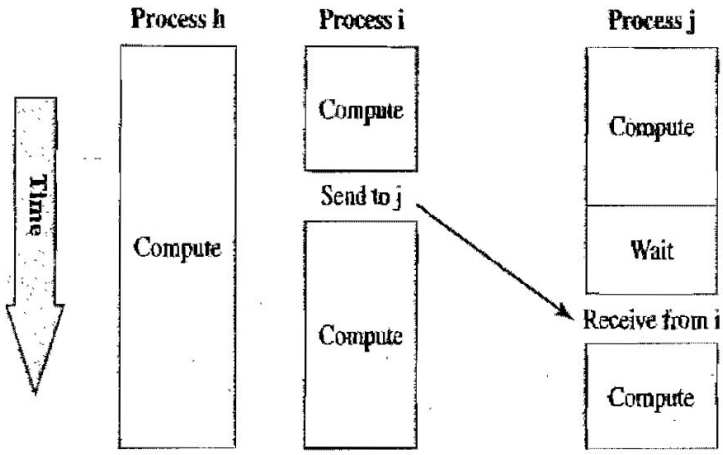

Figure 1.1: Point-to-point communication involve pairs of processes.

```
if (id == i) {
   ...
   /* Send message to j */
   ...
} else if (id == j) {
   ...
   /* Receive message from i */
   ...
}
```

Figure 1.2: Point-to-point communication occur inside conditionally executed code.

```
2 ||    void        *message,
3 ||    int         count,
4 ||    MPI_Datatype    datatype,
5 ||    int         dest,
6 ||    int         tag,
7 ||    MPI_Comm    comm,
8 ||    )
```

In function `MPI_Send`: The first parameter, `message`, is the starting address of the data to be transmitted. The second parameter,`count`, is the number of data items, while the third parameter, `datatype`, is the type of the data items. All of the data items must be of the same type. Parameter 4, `dest`, is the rank of the process to receive the data. The fifth parameter, `tag`, is an integer "label" for the message, allowing messages serving different purposes be identified. Finally, the sixth parameter, `comm`, indicates the communicator in which this message is being sent.

Note that `MPI_Send` blocks until the message buffer is once again available. Typically the run-time system copies the message into a system buffer, enabling `MPI_Send` to return control to the caller. However, it does not have to do this.

Listing 1.4: Example of point to point receive

```
1 || int MPI_Recv(
2 ||    void        *message,
```

C type	MPI type
char	*MPI_CHAR*
unsigned char	*MPI_UNSIGNED_CHAR*
char	*MPI_SIGNED_CHAR*
short	*MPI_SHORT*
unsigned short	*MPI_UNSIGNED_SHORT*
int	*MPI_INT*
unsigned int	*MPI_UNSIGNED*
long int	*MPI_LONG*
unsigned long int	*MPI_UNSIGNED_LONG*
long long int	*MPI_LONG_LONG_INT*
float	*MPI_FLOAT*
double	*MPI_DOUBLE*
long double	*MPI_LONG_DOUBLE*
unsigned char	*MPI_BYTE*

Figure 1.3: MPI data type (not complete)

```
3     int          count,
4     MPI_Datatype     datatype,
5     int          source,
6     int          tag,
7     MPI_Comm     comm,
8     MPI_Status   *status,
9     )
```

In function `MPI_Recv`: The first parameter, `message`, is the starting address where the received data is to be stored. The second parameter,`count`, is the maximum number of data items the receiving process is willing to receive, while the third parameter, `datatype`, is the type of the data items. All of the data items must be of the same type. Parameter 4, `dest`, is the rank of the process to receive the data. The fifth parameter, `tag`, is an integer "label" for the message, allowing messages serving different purposes to be identified. Finally, the sixth parameter, `comm`, indicates the communicator in which this message is being sent.

Note that the seventh parameter, `status`, which appears in `MPI_Recv` but not `MPI_Send`. Before calling `MPI_Recv`, you need to allocate a record of type `MPI_Status`. Parameter `status` is a pointer to this record, which is the only user-accessible MPI data structure.

Function `MPI_Recv` blocks until the message has been received. Note if we want the receiving `process` receive a message from `any` process, we have the option to do that by making the constant `MPI_ANY_SOURCE` the fourth argument to the function, instead of a process number.

1.7.3 Deadlock

A process is in a deadlock if it is blocked waiting for a condition that will never become true. Suppose two processor with rank 0 and rank 1. Each wants to compute the average of a and b. Process 0 has an up-to-date value of `a`; process 1 has an up-to-date value of `b`. Process 0 must read `b` from process 1; while process 1 must read `a` from 0. Consider the following example in Listing 1.5:

Listing 1.5: Example of point to point send deadlock

```
1 float a, b, c;
2 int rank;
```

```
MPI_Status status;
...
if(rank == 0){
    MPI_Recv(&b, 1, MPI_FLOAT,1,0,MPI_COMM_WORLD,&status);
    MPI_Send(&a, 1, MPI_FLOAT,1,0,MPI_COMM_WORLD);
    c = (a + b)/2.0;
}else if (rank == 1){
    MPI_Recv(&a, 1, MPI_FLOAT,0,0,MPI_COMM_WORLD,&status);
    MPI_Send(&b, 1, MPI_FLOAT,0,0,MPI_COMM_WORLD);
    c = (a + b)/2.0;
}
```

Before calling `MPI_Send`, process 0 blocks inside `MPI_Recv`, waiting for the message from process 1 to arrive. Similarly, process 1 blocks inside `MPI_Recv`, waiting for the message from process 0 to arrive. The process are deadlocked. The following code is correct in Listing 1.6:

Listing 1.6: Example of point to point send correct version

```
float a, b, c;
int rank;
MPI_Status status;
...
if(rank == 0){
    MPI_Send(&a, 1, MPI_FLOAT,1,0,MPI_COMM_WORLD);
    MPI_Recv(&b, 1, MPI_FLOAT,1,0,MPI_COMM_WORLD,&status);
    c = (a + b)/2.0;
}else if (rank == 1){
    MPI_Send(&b, 1, MPI_FLOAT,0,0,MPI_COMM_WORLD);
    MPI_Recv(&a, 1, MPI_FLOAT,0,0,MPI_COMM_WORLD,&status);
    c = (a + b)/2.0;
}
```

Now both processes send the data before trying to receive the data. Note that the tag value is 0, we recommend using the same tag value for the program. Due to the reason that if process 0 sends a message with tag 1 and tries to receive a message with tag 1. The process will block inside `MPI_Recv`, because the process can not receive a message with the proper tag.

1.8 Introducing Collective Communication

A **collective communication** is a communication operation in which a group of processes works together to distribute or gather a set of one or more values. Reduction is an example of an operation that requires collective communication in a message passing environment.

Listing 1.7: Example of Collective Communication

```
int maxht, glomx;
...
... (calculations which determine maximum height)
MPI_Reduce(&maxht,&glomx,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
if(rank == 0)
{
...
... (write output)
}
```

1.8.1 Function `MPI_Reduce`

Once a process has completed its share of the work, it is ready to participate in the reduction operation. Function `MPI_Reduce` performs one or more reduction operations on values submitted by all the processes in a communicator The header for function `MPI_Reduce` is

Listing 1.8: Example of setting up the MPI Reduce

```
int MPI_Reduce(
    void         *operand,
    void          *result,
    int          count,
    MPI_Datatype    datatype,
    MPI_Op       operator,
    int          root,
    MPI_Comm     comm,
    )
```

The first parameter is the address of the first reduction element, second parameter is the address of the first reduction result. The third parameter,`count`, indicates how many reductions are being performed. Each process submits `count` values, and each of these values is a list element for a different reduction.

Parameter `operand` is an input parameter. The calling process indicates the location of its element for the first reduction. If `count` is greater than 1, then the list elements for all of the reductions occupy a continuous block of memory. Parameter 4, `type`, is an input parameter designating type of the elements being reduced. Same as the MPI datatype in the `MPI_Send` and `MPI_Send` command. The fifth parameter,`operator`, indicates the kind of process that will have the results of all the reductions. Parameter `result` points to the location of the first reduction result. This parameter only has meaning for process `root`. The last parameter, `comm`, gives the name of the `communicator`, that is the set of processes participating in the reduction.

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Figure 1.4: MPI's built-in reduction operators.

1.9 OpenMP

Parallelism beyond a single node (16 CPUs on hpc-class) requires the use of MPI, however MPI requires major changes to an existing program as you will see in this lab. Two ways exist to get parallelism within a single 16 CPU node: parallelism can either be obtained with automatic parallelism (the -parallel Intel compiler option) or with OpenMP (the -openmp Intel compiler option). One the simplest way to get parallel execution is to add '-parallel' to your compile command. But, there is another way.

Another simple way to obtain parallelism is by using OpenMP, which can be used to express parallelism on a shared memory machine. Since each of the nodes on hpc-class is a shared memory machine with 16 processors, OpenMP can be used to obtain parallelism for 16 processors. It requires changes to the program but not nearly as much as MPI. This of course comes with some tradeoffs. We generally see less gains using using OpenMP then we might with MPI. The gains are generally less than for MPI, but greater than that for automatic parallelism using the compiler option.

In C and C++ we implement OpenMP using `pragma` statements before the part of the code we wish to have run in parallel. For loops are often good candidates to be setup in parallel tasks. Let's take a look at this with a very basic program. Listing 1.9 provides a very simple code you can start with.

For example, let's look at the code below.

Listing 1.9: A very basic For Loop

```
#include <stdio.h>
int main(){
  int i=0;
  for(i=0; i<10; i++){
    printf("%i\n",i);
  }
  return 0;
}
```

If we run the following program, you should get an output like what is below and it is also what we would expect from this program.

```
0
1
2
3
4
5
6
7
8
9
```

Now, let's add some code to make this run in parallel. Listing 1.10 shows how to add the pragma for enabling OpenMP support.

Listing 1.10: Example of using OpenMP

```
#include <stdio.h>
int main(){
  int i=0;
  #pragma omp parallel for
  for(i=0; i<10; i++){
    printf("%i\n",i);
  }
  return 0;
}
```

To compile this code we could use either GCC or the Intel MPIICC. Since we are already using the Intel compiler, go ahead and compile this version but now add the `-fopenmp` flag to it. This will compile the code with that library.

To run this code, we do not need to do anything special. We do not need to setup a `salloc` and we do not need to use `mpirun`. We just run our code as we normally do in Linux. Now, when you run this code you will see something like the following:

```
0
9
7
8
3
4
2
```

```
6
1
5
```

Wait, what happened? Well, we setup the for loop to work in parallel. This means that each processor got a bit of the task and then returned the results. But depending on what the processor was doing, the timing on returning the information is out of order. You saw that with the Hello World as well, where a processor may be delayed in responding if it happened to be doing something else at the time.

As you can see, a major advantage of OpenMP is that it is fairly easy to use. It does not require as many changes to the source code and running our program is a bit more straight forward. But, again there are trade-offs and with OpenMP it can be with the performance of the program using OpenMP.

In general, OpenMP programs run the fastest when most of the operations are on data which is "private" rather than "shared". See the OpenMP Specifications for the meaning of private and shared data with regard to OpenMP.

1.10 Exercises

For the following exercises, make sure you are logged in to the HPC and complete all of them on the HPC. Don't forget to make a develop branch and make sure you push all of your changes back to GitHub when done. You should also capture your outputs so you can use them in your report.

> *Be Careful:*
> Don't forget to load your modules. Also, if you run the `salloc` we gave you, that gives you an hour. Of course you can always put that command in again to get another hour. If you are done, don't forget to exit!

1.10.1 Exercise 1

Exercise 1. Hello MPI World — (5 points): Follow the example in Listing 1.2 and allocate your workspace and run the a.out program that is generated. Capture the output from the screen and put that in your report. Make sure you save your hello world program as `hello_mpi.c`. Run your program with 4, 8 and 16 processors.

For exercise 1, you can use the code found in Listing 1.2. On the HPC, there are some text editors built in. One is Nano and the other is called VIM. Nano can be called with the command `nano` and VIM is, well `vim`. You can also pass in the file name, so for example you can type in `vim hello.c` and it will open the file in VIM. You can also transfer the files back and forth between your computer and the HPC as well. But, be careful with this. Some Windows editors insert extra characters which can cause problems on *Nix machines. Make sure your editor is set to UTF-8 and Unix mode for the CR and LF commands. If you don't know what that means, maybe stick with Nano or VIM.

1.10.2 Exercise 2

Exercise 2. Breaker, Breaker Processor 9 — (10 points): Write a program that allows you to communicate between the different processors on the HPC. Run this code with 4,8 and 16 processors. Note any difference in the time to run this. Finally change the trial variable to 10000 and also compare the time difference in the average time. State on why you think the changes are happening (if any).

Now we will look at a program that allows us to communicate between the different processors. We have given you some starter code in a file called `communicate.c`. Open up this file and take a look at it. Like our Hello MPI World code, we need to include the MPI library to use this. Let's start by Initializing the MPI.

We did this in our Hello MPI World, but this time we need to initialize some communication as well. Look below for the code that needs to be added.

```
/*
    ------------------------------------------------------------------------------------------------

        MPI Initialization
    ------------------------------------------------------------------------------------------------
        */
        MPI_Init(&argc, &argv);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Status stat;
```

Next we need to allocate an array for A, B and for storing our time values. Use Malloc and recall that we can use the size of our int to allocate the size. It should be like the Listing below.

```
    A = malloc(length*sizeof(int));
```

Make sure you do the same for B and then for the time storage you can use the variable `trial` and take it times the sizeof a double.

Now, we need to be able to send and receive information from the processors. We have placed comments on where this needs to go. Let's look at the MPI_Send and MPI_Recv functions we can use.

Listing 1.11: Example of point to point send

```
    MPI_Send(A, n, MPI_INT, j, tag, MPI_COMM_WORLD);
```

Earlier in the writeup, we discussed what each term is.

For our receive, we can look at what we have below.

Listing 1.12: Example of MPI Receive

```
    MPI_Recv(B, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &stat);
```

Again, we discussed what each term is in the write-up. The main thing is that we are sending A in the first example, and then receiving it as B in the next. Your code should look something like what is below.

```
        for (j = 1; j < size; j++) MPI_Send(A, n, MPI_INT, j, tag, MPI_COMM_WORLD);
  // Have the students write the MPI_Recv part
            MPI_Recv(B, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &stat);
                t2 = MPI_Wtime();
                time[iter] = (t2 - t1)*0.5*(1e+6);
    }

    if(rank > 0)
    {
      //Have the students write the Recv and Send
        MPI_Recv(A, n, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
        for (i = 0; i < length; ++i) A[i] += B[i];
        MPI_Send(A, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    }
}
   if(rank == 0) for (i = 0; i < length; ++i) printf("A[%d] = %d\n", i, A[i]);
```

The last thing we need to do is free up our array allocations. Make sure you free up A, B and time.

1.10.3 Exercise 3

Exercise 3. Quad Function — (10 points): Create a program that utilizes multiple processors to solve an estimation of an integration using a Quad function. Again, run this with 4,8 and 16 processors and note the difference in times.

For this exercise we have once again provided starter code for you in a file called `quad.c`. The first thing we want you to do is to add the following equation under the `eval_f` function towards the bottom of the code.

$$value = \frac{50}{\pi * (2500 * x * x + 1)} \tag{1.1}$$

There is one other difference for this program. We are going to use OpenMP to parallize the tasks. In order to do this, we need to add some code like we did in the example in the writeup. We need to also compile this a little differently. We will need to add the `-fopenmp` flag to tell the compiler to use this library. So, to compile this all you need to do is modify your `mpiicc` to include that flag.

Modify your code to have the OpenMP code below added. Place this code in the correct place in your code. This should be above the code you want to have run in parallel. We discussed what types of code is a good candidate, so this should be easy to figure out. The needed is provided in Listing 1.13.

Listing 1.13: Example of setting up pragma for OpenMP

```
1 # pragma omp parallel shared ( a, b, n ) private ( i, x )
2
3 # pragma omp for reduction ( + : total )
```

Run your code with both with the OpenMP statements and without and see if you notice any difference in times. Feel free to adjust some of the other values such as A,B and N as well. Add your results in your report.

1.10.4 Exercise 4

Exercise 4. FFT Function — (10 points): Optimize the FFT program provided to run with OpenMP.

For this last exercise, modify the FFT program called `fft.c`. Add the correct OpenMP code above the correct code and then run your code. Note the times it takes to run with both and the results to your report.

For this one, there is another pragma that we will show you. That is the `nowait`. Normally, OpenMP waits for the other threads to finish before execution continues, but we can also tell it to continue with the nowait option. To add this option, add the pragma as follows.

Listing 1.14: Example of setting the pragma for a nowait in OpenMP

```
1 # pragma omp for nowait
```

Otherwise you will use the same `omp parallel` as used in the quad program. But, please note that you will need to change the variables from the a,b,n to the variables used in the fft program. In our case the n, x, and z variables. The private also needs to be updated to the loop counter, and z0 and z1.

1.11 Report

Your report should have your code outputs from the programs you ran above. It should also include copies of the code you wrote. Finally, make sure you explained how you solved the problems for each Exercise in the **Analysis** section. Remember that your report must be in a file called `main.tex` in a directory called `report` and must build without errors. Also, make sure you address any questions asked in the Exercises.

Warning: As stated in the syllabus, programs and reports that do not compile will be awarded zero points! If you have issues with your lab work, utilize your resources (lecture notes, textbooks, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

1.12 Wrapping Up

Congratulations! You have finished lab 10. This is it, the last lab and you are now done. When you are finished with your lab, check that you have everything done below, and enjoy the rest of your day.

- ☐ **Read the lab write-up and click on the assignment link**
- ☐ **Clone this repo into your account on the HPC**
- ☐ **Complete all 4 exercises**
- ☐ **Ensure the source file compiles with the sample main function without errors**
- ☐ **Push your changes to the develop branch**
- ☐ **Complete your lab write-up**
- ☐ **Open a new Pull Request (and leave it open for the grader to find)**