ECE 4730 --Assignment 01
Due Date provided on Canvas
Global Sum

Task: Write an MPI Program to compute the sum of a large quantity of integers and associated helper executables, according to the specification given below. In the parallel version, you will use a block decomposition so that each
processor will find a partial sum of its portion of the file. Then, the final summation of the partial sums will be done using our the global_sum() function based on the technique given in the text and in class.

In order to carry out this objective, you'll need some helper executables:

```
make-list -n 100 -o outfile.dat
```

This serial executable will generate the initial data file that will contain the list of data vales to be added. It takes two arguments "-n" that indicates the number of values to place in the list, and "-o" to indicate the name of the datafile to write the integers to. If one of the arguments is not specified at runtime, there should be defaults with n=100 and the output file called "default-list-file.dat". The first integer in the file should be the number of integers being written to the list. Then directly following this, all the other n numbers should go back to back. (File should be written as binary, NOT ASCII).

```
serial-add-list -i infile.dat
```

This serial executable will open the specified input file, and will determine the sum of all the numbers in it (excluding the first, which indicates how many there are). It will assume the file format is equal to that of above. It should also have a default input file should the -i not be specified. This default should be the same as specified above. This is essentially the non-parallel version of the parallel one described further down in this assignment.

```
print-list -i infile.dat
```

This serial executable will open the specified input file and will print the contents to the screen in one long column. Again, defaults as specified above.

```
parallel-add-list -i infile.dat
```

This is the parallel version of the list adder. Defaults as specified above. As mentioned above, it will use block decomposition. You will likely find useful the author's utility files from Appendix B of the textbook. These two files, MyMPI.c and .h can be found on Canvas. These files may have some places that will show warnings when compiled with modern compilers, as was the case on my laptop. You will need to correct these issues. Be careful of what you change here, you don't want some unintended side-effect.

For this project, you will likely find the following functions from those files useful:

```
void read_block_vector(char *, void **, MPI_Datatype,
int *, MPI_Comm);

void print-block-vector(void *, MPI_Datatype, int,
MPI_Comm);
```

and the macros from the MyMPI.h file, in particular, the:

```
BLOCK_SIZE(rank,size,n)
BLOCK_LOW(rank,size,n)
```

The read_block_vector() function above, will open a file, read the first integer out of the file, and then will have each process malloc() the appropriate amount of space to hold it's part of the file. Then, one process will read the chunks of the file out, and send it to all the other processes. Essentially, this function does some very heavy lifting for you. The same effect can be achieved easily with MPIIO, something that we will discuss later. I expect you to look at what these functions in order to determine what they do, how they do it, and ultimately how to use them to accomplish your goal.

You are to write a function with the following specification:

```
void global_sum(
long int *result, int rank, int size, lonf int *my_value);
```

This function must be executed by all tasks in MPI_COMM_WORLD and computes the sum of all the my_value's and returns the sum of all the tasks individual 'my_value's via the result pointer. (All processors must know the global sum after the call returns) Note, this is similar to MPI_Allreduce with addition does, except I only expect you to deal with scalar local my_values (i.e. not the reductions of arrays of values). Note, for convenience, you may assume that the number of processors you run this program is a 'power of 2', i.e. 2, 4, 8, 16, etc. You must put in some error checking code to see ensure your code only executes if the number of processors matches this assumption and if not, prints a message and immediately exits.

As mentioned above, this is essentially equivalent to an MPI_Allreduce (with the SUM operator), but you are to write it only using MPI_Send and MPI_Recv (or MPI_Sendrecv). You are not to use any collective communication routines. The general discussion of binomial trees that might be useful for this project is located in Section 3.5 of the Quinn textbook.

The project will be broken down into the following files (must contain these):

```
make-list.c
MyMPI.c
parallel-add-list.c    includes global_sum()
print-list.c
serial-add-list.c
functions.h
MyMPI.h
Makefile
```

Develop your programs first on your local machine, however you MUST also execute it on Palmetto on 2, 4, 8, 16, and 32 processors via the PBS queueing system.
Submission into BlackBoard:

Your project should reside in a single directory, that will have all necessary files. That directory should be named first_last_asn3, with the files mentioned above inside. From there, you will tar and gzip that directory, and submit that to BlackBoard. It will contain all your files, as well as the PBS scripts, and the output files from Palmetto. That can be done from the command line as follows with a tar command:

```
$ pwd
/home/wjones/classes/473
$ tar czf will_jones_asn3.tgz ./will_jones_asn3
$ ls
will_jones_asn3 will_jones_asn3.tgz
```

For future reference, you can untar and unzip a tar.gz file like this:

```
$ tar xvfz will_jones_asn3.tgz
./will_jones_asn1/
./will_jones_asn1/make-list.c
./will_jones_asn1/MyMPI.c
./will_jones_asn1/parallel-add-list.c
./will_jones_asn1/print-list.c
./will_jones_asn1/serial-add-list.c
./will_jones_asn1/functions.h
./will_jones_asn1/MyMPI.h
./will_jones_asn1/Makefile
```

It would be a good idea to test to make sure your tarring and zipping worked by copying it to a temp directory, and unzipping it to make sure it is all there.
From there, upload to BlackBoard your .tgz file.

Appendix:

Unix Makefiles are very powerful tools, not just for coding. There are some very advanced features out there; however here is a simple example of my Makefile for this assignment. It might be a good idea to look over a Makefile tutorial as well. There are many out there.

Note that below the indentation is created by using a single TAB, not spaces. This is important.

```
CFLAGS = -g -Wall -Wstrict-prototypes
PROGS = runmain
OBJECTS = HW04_will_jones.o functions.o
LDFLAGS = -lm
CC = gcc
MCC = mpicc

all: $(PROGS)

runmain: $(OBJECTS)
	$(MCC) $(LDFLAGS) -o runmain $(OBJECTS)

HW03_will_jones.o: HW03_will_jones.c
	$(MCC) $(CFLAGS) -c HW03_will_jones.c

functions.o: functions.c functions.h
	$(MCC) $(CFLAGS) -c functions.c

clean:
	rm -f $(PROGS) *.o core*
```

Parsing command arguments in the helper executables should use the standard function "getopt()". Read the man page on this carefully. Here is a skeleton where I used it:

```
int num
char *ofile
while((opt = getopt(argc, argv, "n:o:")) != -1)
{
	switch(opt)
	{
		case 'n':
			num = atoi(optarg);
			...
			break;
		case 'o':
			 ofile = strdup(optarg);
			...
			break;
	}
}
```