**CSCI 473 --Assignment 2 Spring 2020**

**Due Date provided on Moodle**

***Implementing your own MPI\_Reduce() for the cpi.c file***

Take a look at the prior assignment. Look specifically at the MPI call:

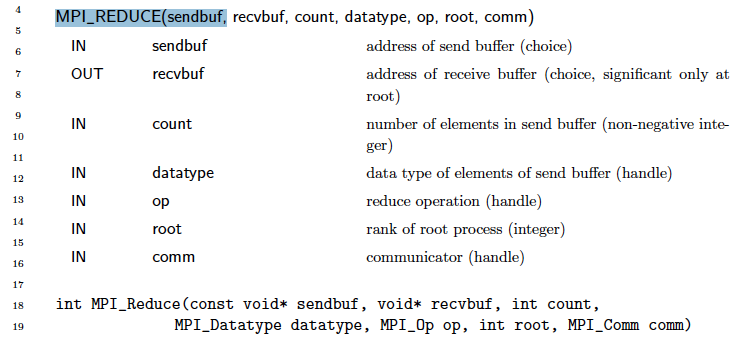


If you look at the MPI Specification, which can be found at:

<https://www.mpi-forum.org/docs/>

Specifically, at version 3.1: <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>

You’ll find a great description of the MPI\_Reduce call on page 174:



Read about this in the MPI specification, and also look at it in the Quinn textbook.

As such, you can see that in the cpi.c program, the authors used MPI\_Reduce to sum up all the partial sums across all processes to get the final estimate for PI.

In this assignment, I want you to write your own function call, with the following prototype:

**void** global\_sum(

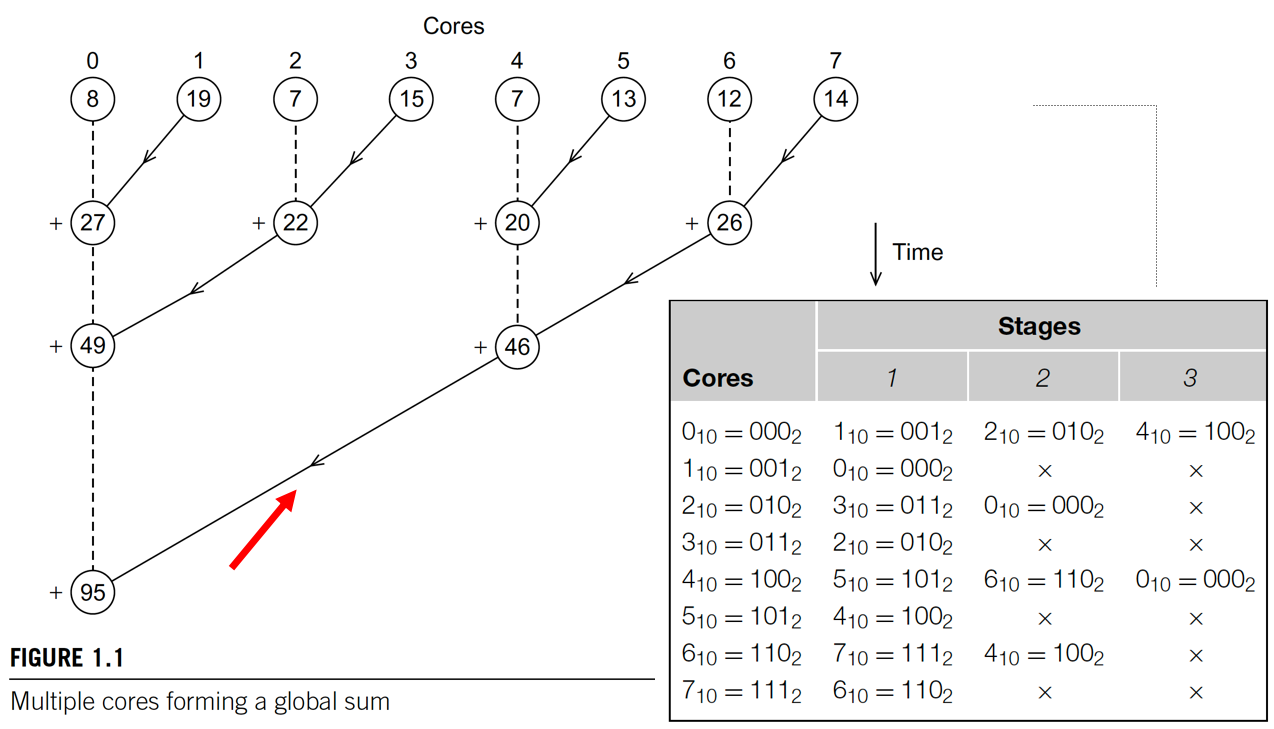
**double\*** result, **int** rank, **int** size, **double** my\_value);

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)

This does not than just an MPI\_Reduce, in this case, all processes have the sum.

Note, for convenience, you may assume that the number of processors you run this on 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, immediately exits. *(you should be able to find a piece of code that checks to see if an integer is a power of two)*

You are to write it only using MPI point to point communication primitives: MPI\_Send and MPI\_Recv (or MPI\_Sendrecv). **You are not to use any collective communication routines**. You will find that the trick to doing this well (in log(p) number of communication phases, versus p phases) will utilize a variation of this tree-based strategy:



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

**cpi.c** // this is where main() is and also where each process calculates his partial integration for PI, and then all processes invoke global\_sum() and then report the global to stdout. This is your “driver” (executable) program. The executable should be called ***driver***. (i.e. the C file is named cpi.c, but the -o executable is called driver)

**functions.c** // this is where you have global\_sum() defined, among others

**functions.h** // this is where you have your global\_sum() prototype, among others

**Makefile** // this is a typical Unix makefile that will compile your main program as well as functions.c file to objects, and will then link them together into a final executable. Use the Makefile I give below (in Appendix)

Therefore, you should be able to use my own Makefile, and your cpi.c file, and compile in your code and still work. (in other words, you must have your functions called exactly as above).

**Example Execution:**

CCU018260% make clean

rm -f driver \*.o core\*

CCU018260% make

mpicc -g -Wall -Wstrict-prototypes -c cpi.c

mpicc -g -Wall -Wstrict-prototypes -c functions.c

mpicc -lm -o driver cpi.o functions.o

CCU018260%

CCU018260% mpirun -oversubscribe -np 8 ./driver

Phase 0 - P 2 (010) sendrecv with P 3 (011), val 2.0

Phase 0 - P 3 (011) sendrecv with P 2 (010), val 3.0

Phase 0 - P 5 (101) sendrecv with P 4 (100), val 5.0

Phase 0 - P 6 (110) sendrecv with P 7 (111), val 6.0

Phase 0 - P 1 (001) sendrecv with P 0 (000), val 1.0

Phase 0 - P 4 (100) sendrecv with P 5 (101), val 4.0

Phase 0 - P 7 (111) sendrecv with P 6 (110), val 7.0

Phase 0 - P 0 (000) sendrecv with P 1 (001), val 0.0

Phase 1 - P 3 (011) sendrecv with P 1 (001), val 5.0

Phase 1 - P 2 (010) sendrecv with P 0 (000), val 5.0

Phase 1 - P 0 (000) sendrecv with P 2 (010), val 1.0

Phase 1 - P 7 (111) sendrecv with P 5 (101), val 13.0

Phase 1 - P 4 (100) sendrecv with P 6 (110), val 9.0

Phase 1 - P 6 (110) sendrecv with P 4 (100), val 13.0

Phase 1 - P 1 (001) sendrecv with P 3 (011), val 1.0

Phase 2 - P 4 (100) sendrecv with P 0 (000), val 22.0

FINAL IN MAIN: Process: 4 has Sum = 28.000000

Phase 1 - P 5 (101) sendrecv with P 7 (111), val 9.0

Phase 2 - P 5 (101) sendrecv with P 1 (001), val 22.0

Phase 2 - P 6 (110) sendrecv with P 2 (010), val 22.0

FINAL IN MAIN: Process: 6 has Sum = 28.000000

Phase 2 - P 7 (111) sendrecv with P 3 (011), val 22.0

Phase 2 - P 0 (000) sendrecv with P 4 (100), val 6.0

FINAL IN MAIN: Process: 0 has Sum = 28.000000

Phase 2 - P 2 (010) sendrecv with P 6 (110), val 6.0

FINAL IN MAIN: Process: 2 has Sum = 28.000000

Phase 2 - P 1 (001) sendrecv with P 5 (101), val 6.0

FINAL IN MAIN: Process: 1 has Sum = 28.000000

Phase 2 - P 3 (011) sendrecv with P 7 (111), val 6.0

FINAL IN MAIN: Process: 5 has Sum = 28.000000

FINAL IN MAIN: Process: 3 has Sum = 28.000000

FINAL IN MAIN: Process: 7 has Sum = 28.000000

CCU018260% (that is the order it came out on my machine). Now, looking at it.

Now, I’ll sort the output (post-run) so that we can see the patterns:

CCU018260% mpirun -oversubscribe -np 8 ./driver | sort

Phase 0 - P 0 (000) sendrecv with P 1 (001), val 0.0

Phase 0 - P 1 (001) sendrecv with P 0 (000), val 1.0

Phase 0 - P 2 (010) sendrecv with P 3 (011), val 2.0

Phase 0 - P 3 (011) sendrecv with P 2 (010), val 3.0

Phase 0 - P 4 (100) sendrecv with P 5 (101), val 4.0

Phase 0 - P 5 (101) sendrecv with P 4 (100), val 5.0

Phase 0 - P 6 (110) sendrecv with P 7 (111), val 6.0

Phase 0 - P 7 (111) sendrecv with P 6 (110), val 7.0

Phase 1 - P 0 (000) sendrecv with P 2 (010), val 1.0

Phase 1 - P 1 (001) sendrecv with P 3 (011), val 1.0

Phase 1 - P 2 (010) sendrecv with P 0 (000), val 5.0

Phase 1 - P 3 (011) sendrecv with P 1 (001), val 5.0

Phase 1 - P 4 (100) sendrecv with P 6 (110), val 9.0

Phase 1 - P 5 (101) sendrecv with P 7 (111), val 9.0

Phase 1 - P 6 (110) sendrecv with P 4 (100), val 13.0

Phase 1 - P 7 (111) sendrecv with P 5 (101), val 13.0

Phase 2 - P 0 (000) sendrecv with P 4 (100), val 6.0

Phase 2 - P 1 (001) sendrecv with P 5 (101), val 6.0

Phase 2 - P 2 (010) sendrecv with P 6 (110), val 6.0

Phase 2 - P 3 (011) sendrecv with P 7 (111), val 6.0

Phase 2 - P 4 (100) sendrecv with P 0 (000), val 22.0

Phase 2 - P 5 (101) sendrecv with P 1 (001), val 22.0

Phase 2 - P 6 (110) sendrecv with P 2 (010), val 22.0

Phase 2 - P 7 (111) sendrecv with P 3 (011), val 22.0

FINAL IN MAIN: Process: 0 has Sum = 28.000000

FINAL IN MAIN: Process: 1 has Sum = 28.000000

FINAL IN MAIN: Process: 2 has Sum = 28.000000

FINAL IN MAIN: Process: 3 has Sum = 28.000000

FINAL IN MAIN: Process: 4 has Sum = 28.000000

FINAL IN MAIN: Process: 5 has Sum = 28.000000

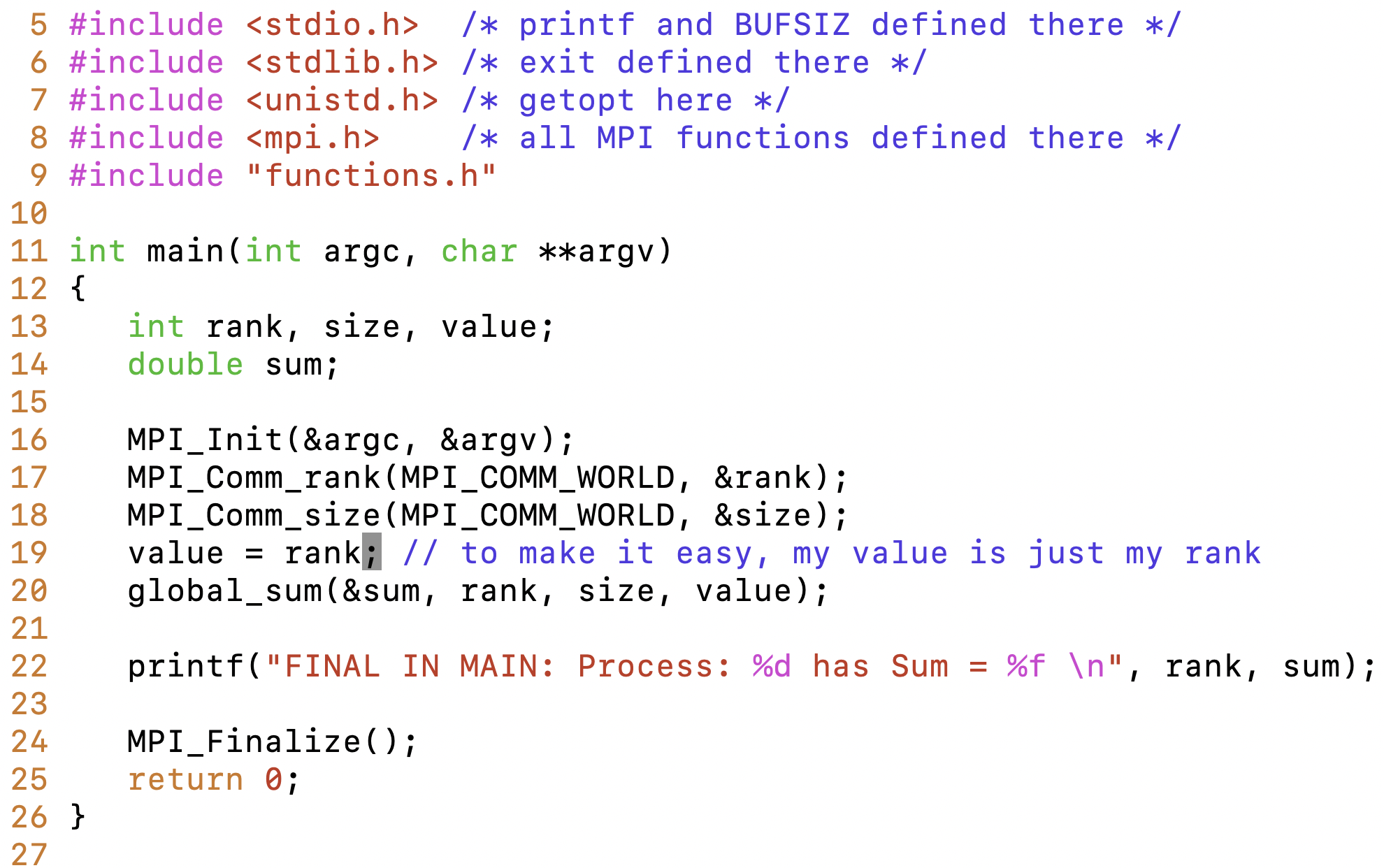
FINAL IN MAIN: Process: 6 has Sum = 28.000000

FINAL IN MAIN: Process: 7 has Sum = 28.000000

CCU018260%

**Your program MUST produce the same format of output as above. (you do not have to implement the binary representation string (the ones highlighted above))**

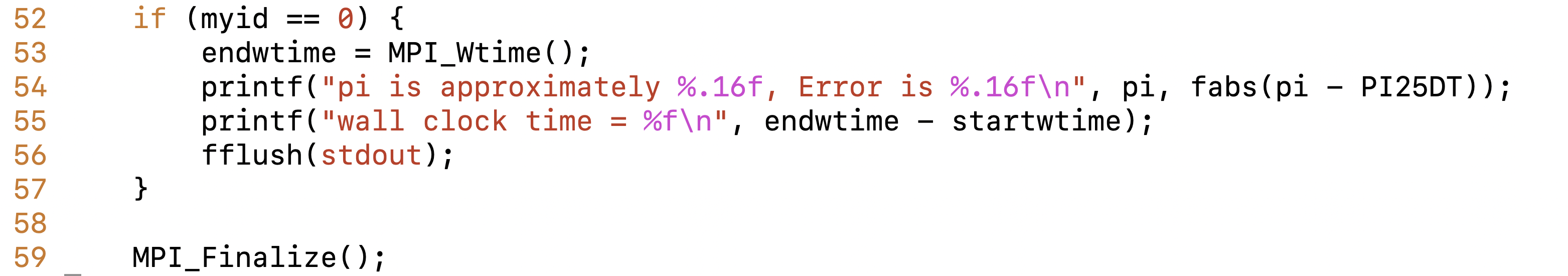
My main program looks like this. This would be a good place to start:



Here in this example, I just use each processes’ rank as the partial value of PI, but in your program, when finished, it should be the partial integration of PI on each process. But you may find that the above works well to help you debug your code.

Note, this is an example main, but you’ll want to call your new function from the cpi.c file from the last assignment, and then have all processes (from main) print out their calculated values for PI. I want a Word doc / report that shows the program being compiled, executed, and discussed. For np = 2, 4, 8, 16 processes. Take plenty of screenshots showing in detail everything.

For example, in the cpi.c file:





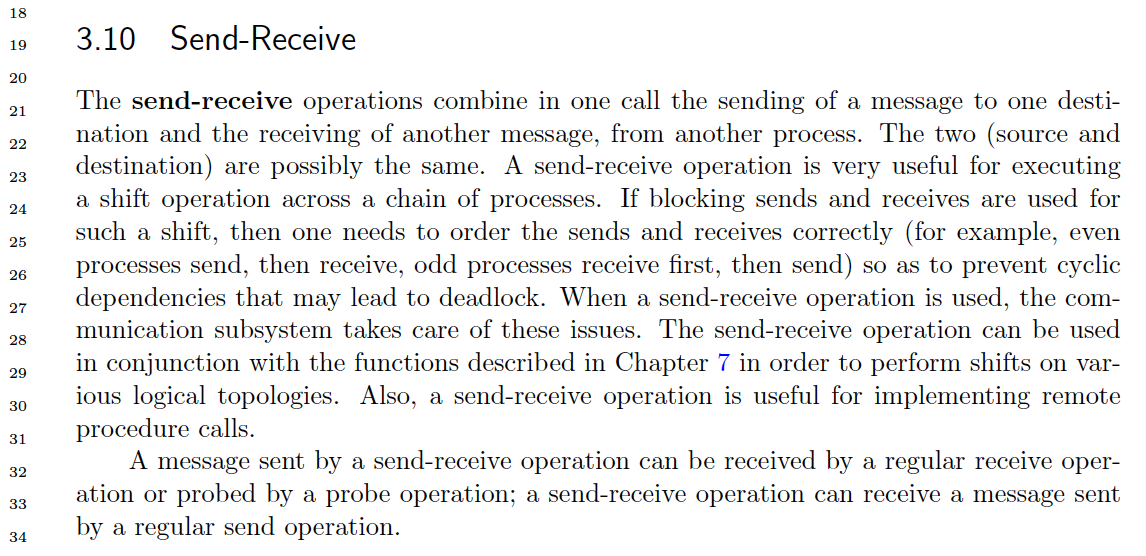
Only process 0 printed out the results (because it did a reduce to rank 0, but in your program, they all will print out their results. So basically, the cpi.c file should be just as it was before, but instead of calling MPI\_Reduce() it would be calling your global\_sum() function, and instead of just rank 0 printing results, all ranks print their results.

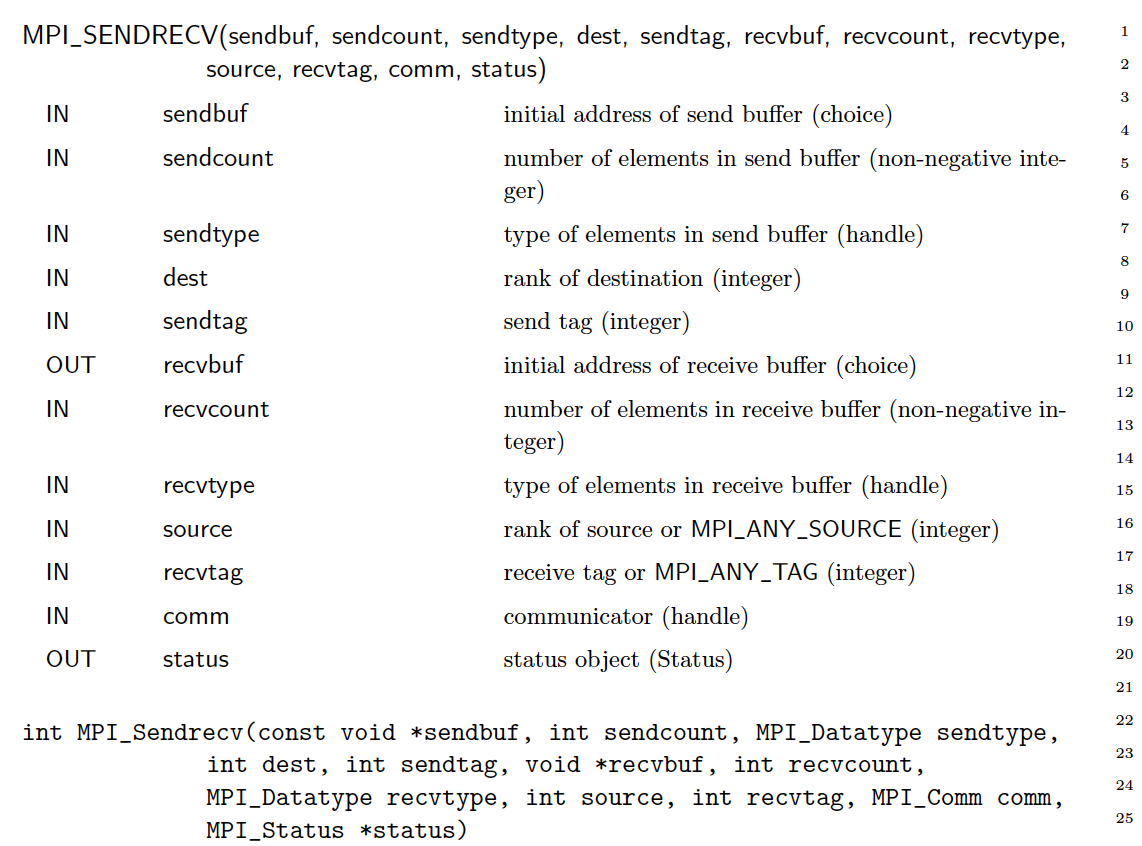
**Some Hints:**

Here’s a place to start to determine if a number (size) is NOT a power of 2:

if((size & (size-1)) != 0) then not a power of two

Use MPI\_Sendrecv() instead of dealing with Send() and Recv() separately:





**Grading Rubric:**

1) you will get minimum points for a completely working solution where the number of communication phases is O(2log n) “C”

2) you will get additional points if your solution uses a bitmask in controlling the

stages of data exchange. (using bitmask and shifting, not dividing) “B”

2) you will get additional points for a solution that does NOT do a reduce followed

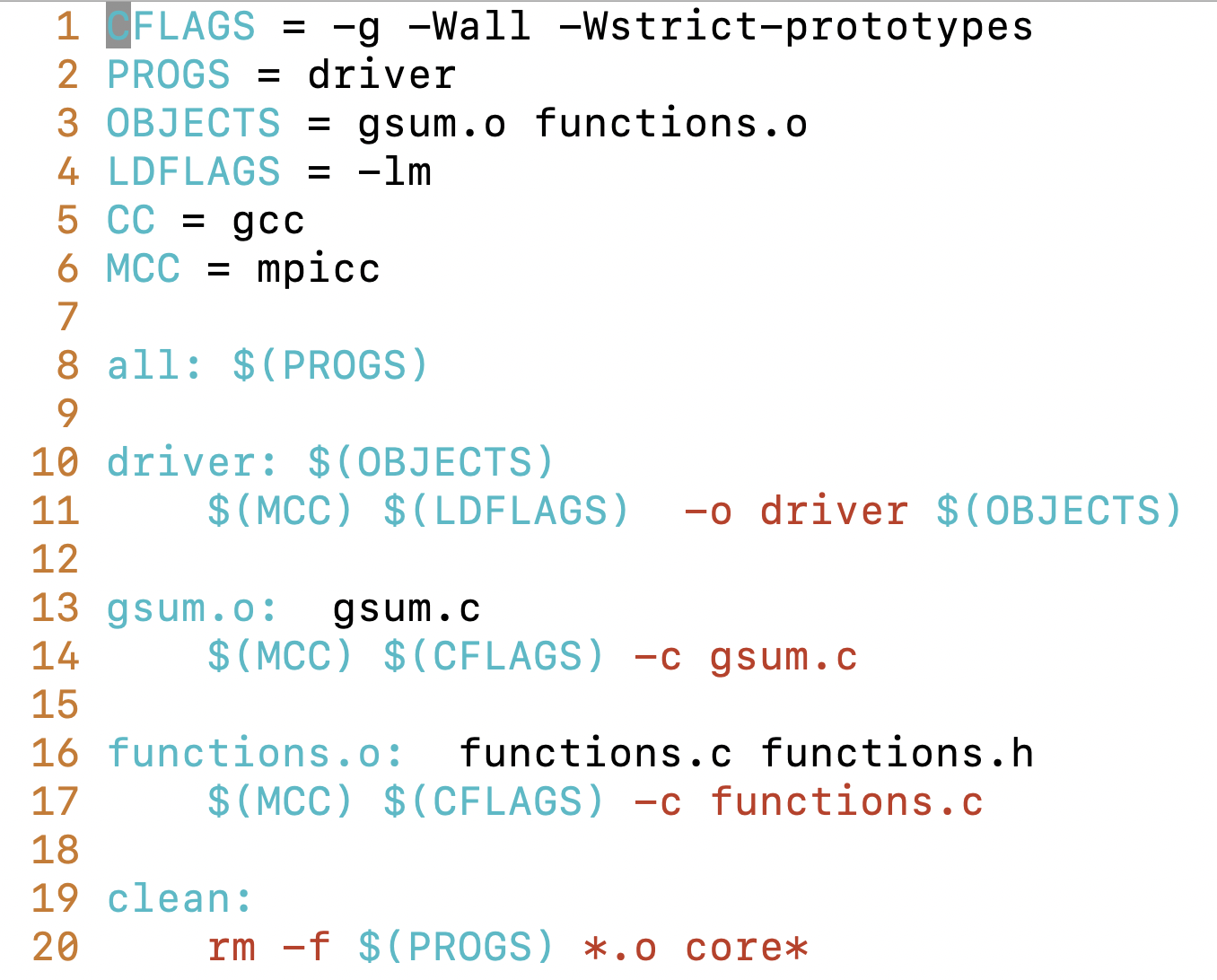
by a broadcast. (i.e. O(log n) instead of O(2log n)) (in other words, does not do an inverted tree, followed by a tree, but rather does all the exchanging in the same first phase of communications. “A”

Look at my output above. Each “phase” is one set of communication that is taking place essentially concurrently. So since np = 8 in my example, there were log(8) = 3 phases. Your code should execute in these phases too.

**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. You already have a example of a Makefile from the git repo, but here is another one, slightly different than what you have seen in class so far.

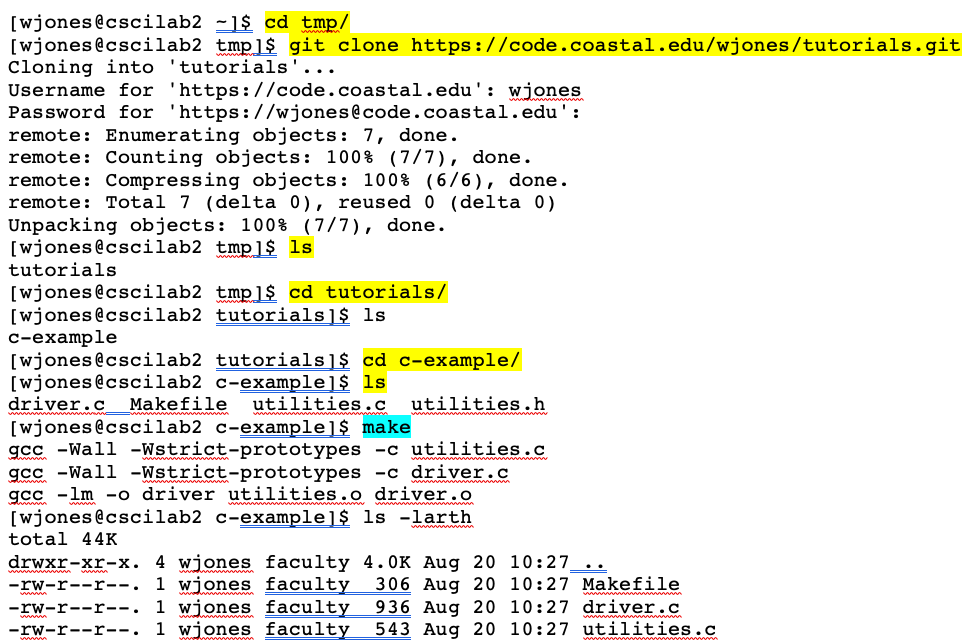
Note that below the indentation is created by using a single TAB, not spaces. This is important.





*Note, the screenshot above the file gsum.c is the same as your cpi.c*

Note, you can see another example of a Makefile in the c-example folder inside of the git repository that you cloned in the “Accessing the CI project” that I asked you to do, specifically:



**Submission into Moodle:**

I expect that your project will reside in a single directory, and that will have all necessary files in that directory. Including the report mentioned above. That directory should be named first\_last\_mpi02, with the files mentioned above inside. From there, you will tar and gzip that directory, and submit that to Moodle. That can be done from the commandline as follows with a tar command piped to gzip.

(remember make clean, before zipping)

$ pwd

/home/wjones/classes/473

$ **tar cf - ./will\_jones\_ass2/ | gzip > will\_jones\_ass2.tar.gz**

$ ls

ass1 ass2 ass3 will\_jones\_ass2 **will\_jones\_ass2.tar.gz**

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

$ **tar xvfz will\_jones\_ass2.tar.gz**

./will\_jones\_ass4/

./will\_jones\_ass4/functions.c

./will\_jones\_ass4/cpi.c

./will\_jones\_ass4/Makefile

./will\_jones\_ass4/functions.h

*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 Moodle you .tar.gz file.