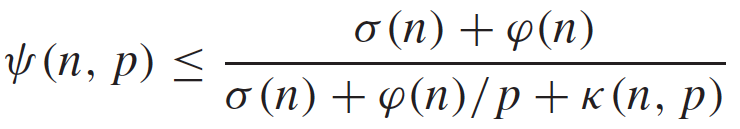
**Instructions:**

Now that the class has been moved online due to COVID-19, I expect this work to be done individually, and not in conjunction with any past or present students, faculty, or other people. However, you may use your class notes and textbooks to respond to these.

Respond completely, INSIDE THIS file itself, as I will be grading this exam inside the word document. Type up responses, draw diagrams, respond fully. Take screenshots, make illustrations, take as much space as you need. Answer completely. State assumptions. Do what you need to in order to demonstrate understanding and mastery of this material. Note, the last page is a coding part of the exam, which also needs to be done and submitted. Look at instructions. Pay attention to directions.

**Performance Evaluation of Parallelization**

In class, we derived and discussed this inequality for speedup. (this is the notation used in class and also by the Quinn textbook).



Define each term here, and does it represent and where does it come from?



**- Speedup**



**- Serial portion of execution time**



**- Parallel portion of execution time**



**- Parallel portion of execution time on p processes**



**- Parallel overhead**

What does the numerator above represent? What does the denominator represent?

**The numerator represents the total execution time on one process (T1) while the denominator represents the total execution time on p processes (Tp). The complete fraction is .**

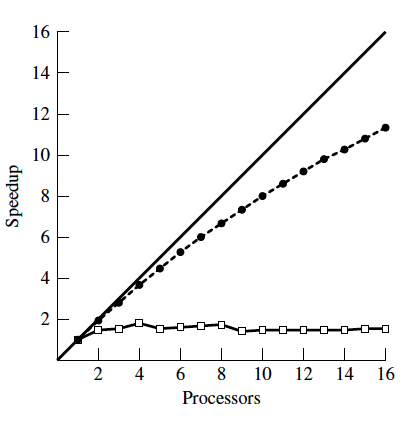
Define efficiency.

**Efficiency is the measure of processor utilization. The inequality for this is**

What does  represent, and is it better for this to be a close to 0 or close to 1, and why?

**This represents the serial fraction of a program. It is better for it to be close to 0 as to minimize serial computation**

Consider these predictions of speedup:





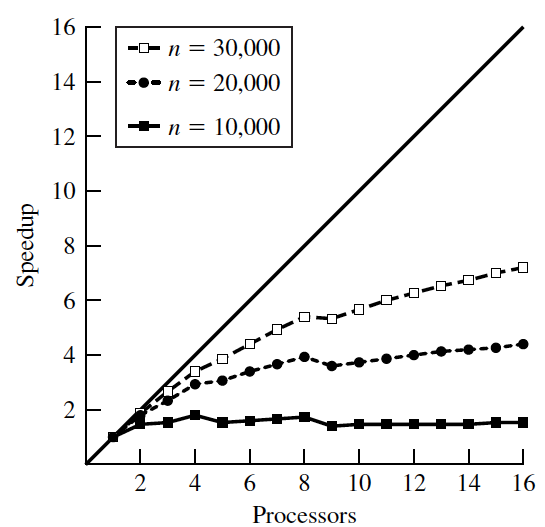
What does the ‘a’ curve represent?

**‘a’ represents the ideal speedup, i.e. when speedup is equal to the number of processors.**

Of curves b and c, one of these curves includes a non-zero k(n,p) term. Which term assumes k(n,p) is 0, and which does not, and why?

**Curve ‘b’ assumes that κ(n,p) is zero and curve ‘c’ includes a non-zero κ(n,p). The reason for this is that including parallel overhead in the speedup inequality will make the fraction smaller, thus the speedup will be smaller and not including parallel overhead will make the fraction larger, thus increasing the speedup.**

Consider these observed measures of obtained speedup:



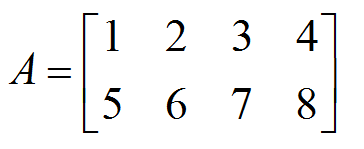
We have seen that as the problem size increases, our measured speedups tend to improve for a fixed number of processors. Why is this? What are some reasons this could be?

**The reason for this is that a larger problem size results in better utilization of multiple processors, as in the parallel time divided by number of processors in the speedup inequality gets smaller as more processors are used, leading to a larger speedup.**

**Pointers / 2-D arrays / Memory Hierarchy**

Assuming that a 2D matrix is stored in memory using the allocation technique we made use of in class (from the Robey textbook).

Assume the matrix is



Assume A was declared and malloced from main() like this:



Sketch the entire 2D array structure, including the pointers, for how A is stored in memory, for the method we discussed in class, and have been using for our assignments so far. Show the variables on the stack and on the heap. Show individual pointers. Make up memory addresses, be consistent and also draw arrows for the pointers.

A

0x100

0x102

0x101

0x101

0x103

0x102

0x107

0x103

1

0x104

2

0x105

3

0x106

4

0x107

5

0x108

6

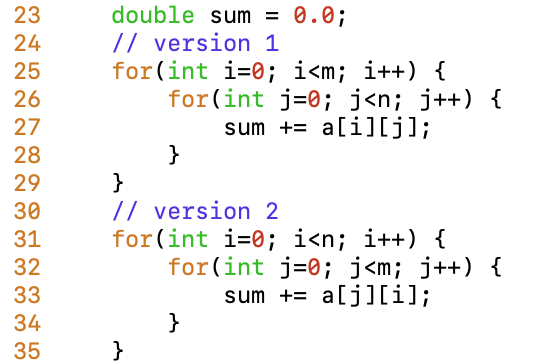
0x109

7

0x110

8

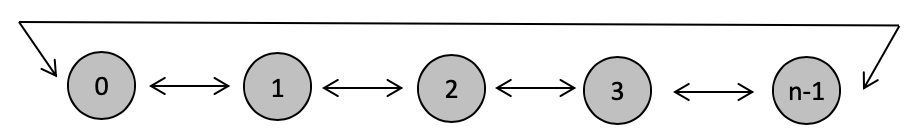
Given what we discussed in class, which of the following two versions of code would likely run faster for a large matrix A[m][n] (m rows, n cols), why, explain fully, sketch out the issue, etc. Explain the overarching issues, from a computer system point of view.



**The C language uses row major ordering for its arrays, that is the rows of an array are contiguous in memory. This means that data in a row will be accessed faster than data in a column. In the case above, version one of the loop would run faster for large matrices, as it traverses the rows of the array first, adhering to the row major order of C.**

**MPI Programming**

Imagine that all processes are logically connected in a ring topology as follows:



Therefore, processes (*not processors*) can send and receive data **only** from their adjacent neighbors. Note that the ring wraps around, i.e. process 0's left neighbor is process (n-1) and process (n-1)'s right neighbor is process 0.

**Each process has a value “*myNum*” that is a randomly generated integer number between 0 and 100.** Be sure each process seeds its random number generator with its rank. *(use random() and srandom() from stdlib.h … look at the “man” page for this for more information about how to do this)*

Each process will first print its own ID (its rank), it's *myNum* value, as well as the ranks of its left and right neighbor, all on a single line. Then each process will exchange its *myNum* value with its neighbors. Each process will then print its own ID, and the values it received from its neighbors, all in a single line. Should work for both an even and off number of processes, any number of processes > 1.

A sample execution of this program on 9 processes should look like this, (*with the possible exception of the random number values and the order the prints occur in*).

(your formatting MUST look like this)

Process [ 1] has myNum = 83, R\_rank = 2, L\_rank = 0

Process [ 2] has myNum = 90, R\_rank = 3, L\_rank = 1

Process [ 4] has myNum = 1, R\_rank = 5, L\_rank = 3

Process [ 6] has myNum = 41, R\_rank = 7, L\_rank = 5

Process [ 7] has myNum = 77, R\_rank = 8, L\_rank = 6

Process [ 8] has myNum = 96, R\_rank = 0, L\_rank = 7

Process [ 3] has myNum = 46, R\_rank = 4, L\_rank = 2

Process [ 0] has myNum = 1, R\_rank = 1, L\_rank = 8

Process [ 5] has myNum = 75, R\_rank = 6, L\_rank = 4

Process [ 3]: recv'd 1 from R\_rank = 4, recv'd 90 from L\_rank = 2

Process [ 4]: recv'd 75 from R\_rank = 5, recv'd 46 from L\_rank = 3

Process [ 1]: recv'd 90 from R\_rank = 2, recv'd 1 from L\_rank = 0

Process [ 2]: recv'd 46 from R\_rank = 3, recv'd 83 from L\_rank = 1

Process [ 5]: recv'd 41 from R\_rank = 6, recv'd 1 from L\_rank = 4

Process [ 6]: recv'd 77 from R\_rank = 7, recv'd 75 from L\_rank = 5

Process [ 7]: recv'd 96 from R\_rank = 8, recv'd 41 from L\_rank = 6

Process [ 8]: recv'd 1 from R\_rank = 0, recv'd 77 from L\_rank = 7

Process [ 0]: recv'd 83 from R\_rank = 1, recv'd 96 from L\_rank = 8

In terms of MPI calls, you may only make use of MPI\_Sendrecv(); no other point-to-point or collective communication constructs may be used. **Your solution must be guaranteed deadlock free, regardless of any internal MPI buffering.** Communication must also be done in such a way that multiple processes are communicating at the same time. *If you think you need a loop to carry out the communication,* ***you're doing it the wrong way****.*

Chapter 3 (Point to Point Communication) of the MPI Specification Text is a valuable repository of knowledge and examples.

Your project will have the following files:

Makefile

first\_last\_midterm\_ring.c

These files will reside in a directory called “first\_last\_midterm”, where first and last are your first and last names, respectively.

The Makefile will be a properly formatted Makefile with both an 'all' and a 'clean' section.

Test your code on 2, 3, 4, 5, 8, 9, 10, and 16 processes on your own machine (if you’re using you own machine) and ALSO on ci.coastal.edu. Make sure all files reside in your directory.

**Report in Midterm:**

**Discuss your code, provide screenshots, show example output demonstrating that it works.**

To submit your project, you must tar gzip your project directory, and its contents by:

tar cf - ./first\_last\_midterm | gzip > first\_last\_midterm.tar.gz

INCLUDE THIS FILE with your responses in it, inside this TAR GZ.

You will perform this operation from the parent directory of “ first\_last\_midterm”.

Submit your tar.gz files on Moodle by the time and date indicated. No late work will be accepted. (will be submitted along with your solution to the exam)

**Midterm Exam – Report**

**I. Implementation**

**The program joshua\_catoe\_midterm\_ring (**Figure 1**) generates a random number within the range 0-100 and uses MPI to have each process send their own random number to their left and right neighboring processes (best envisioned as a circle or ring topology). At the beginning of the program, the MPI environment was initialized followed by the declaration of all necessary variables. The total number of processes (*size*) and the individual process’s rank were then determined using the *MPI\_Comm\_size()* and *MPI\_Comm\_rank()* functions. The random number generator was seeded using the current time plus the process’s rank to ensure different random numbers for each process at every program execution. The process’s own random number (*myNum*) was then generated using the *random()* function modulo 100. Modulo 100 was used in order to shrink the range of random numbers down to 0-100. Following this, each process’s left and right “neighbors” were determined. Processes zero and *size*-1 were special cases, as this “ring” of processes “wrapped around”, meaning that zero’s left neighbor was the process of highest rank (*size*-1) and the process of highest rank’s right neighbor was process zero. All other processes determined their left neighbor by subtracting one from their rank and their right neighbor by adding one to their rank; process zero and *size*-1 also used this technique to determine their left and right neighbor, respectively. Each process then printed out their random number, right neighbor rank and left neighbor rank. The *MPI\_Sendrecv()* function was used to send a process’s random number to a neighboring process and receive a neighboring process’s random number. To avoid deadlock and ensure that each process was always active, processes first sent their data to their left neighbor and received from their right, and then they sent to their right neighbor and received from their left. Finally, each process printed out the random number received from their right and left neighbors, along with their right and left neighbor ranks, and the MPI environment was shut down.**

**A screenshot of a computer

Description automatically generated**

**Figure 1: joshua\_catoe\_midterm\_ring**

**Figure 1 (2)**

A screenshot of a computer

Description automatically generated

**II. Execution**

**The program executed successfully on both a local machine and the COMET supercomputer and produced the expected output (**Figure 2**). It works for either even or odd numbers and powers of two and non-powers of two. The order of the printed output was very close to what was desired, but due to the nature of printing to *stdout*, it did not match completely.**

**Figure 2: Program output**

**A screenshot of a computer

Description automatically generated**