CSC 671: Parallel Programing and Extreme-scale Computing

Assignments 1 & 2

IMPORTANT NOTES ON SUBMISSION

- Your assignment should be submitted electronically using Sakai.
- The final submission should be a single zipped file. Within the zipped contents, please supply your source codes as separate files using an appropriate name to identify the program source file. Please include sufficient comments to explain the logic of your programs. Please also include instructions on how to compile and run your code. Finally, please supply a snapshot of the user input (if needed) and output obtained in the assignment solution file along with analysis and graphs (if needed). **If I cannot run you code and reproduce your results, you get no points!**
- Clearly state any assumptions that you make.
- Any submission email received beyond midnight on the due date will be considered late and will be graded with a 25% penalty. Submissions more than 2 days late will not be graded.
- **Assignments are individual efforts. Please submit your own work!**

Introduction

Modern CPU architectures contain multiple processing cores and are capable of carrying out multiple calculations in *parallel*. In fact, even today's smartphones have multiple cores. Thus, it is critical to understand how to program for such architectures in order to fully exploit the benefits of parallelism.

This assignment will cover the following concepts:

a. Thinking Parallel (How do we parallelize problems?)
b. Multi-Core Parallel Execution
c. Performance Analysis of Parallel Programs
d. Programming Shared-Memory Systems using *pthreads* and *OpenMP*

Harnessing the power of parallel computing can provide significant performance increases, increased reliability (e.g., fault tolerance), and support naturally distributed applications. However, as with any complex system, the more components you add, the more challenging it is to ensure these components do not interfere with one another. For example, let's think about modern railway networks. A long time ago, a single section of track was traversed by only one train, so the only thing for railway engineers to look out for were obstructions along that track as they traveled. However, in today's railway systems, there are multiple trains, shared sections of track, switches to change the track a particular train is on, trains traveling at different rates of speed, varying arrival/departure deadlines, etc. Although each train still operates independently

of the others, it is imperative to manage the relationship between it and the other trains, as well as the system around it. It becomes a *concurrency* problem.

In computing, **concurrency** refers to the coordination and management of independent lines of execution. Concurrent programming involves solving a problem by breaking it up into concurrently executing threads or processes. If the problem is purely parallel (viz., the problem can be solved with no coordination between the components), then there are no potential problems. However, it is more than likely that there will be some level of synchronization required and/or the use of a shared resource. It is up to the programmer to evaluate and ensure these issues are properly managed.

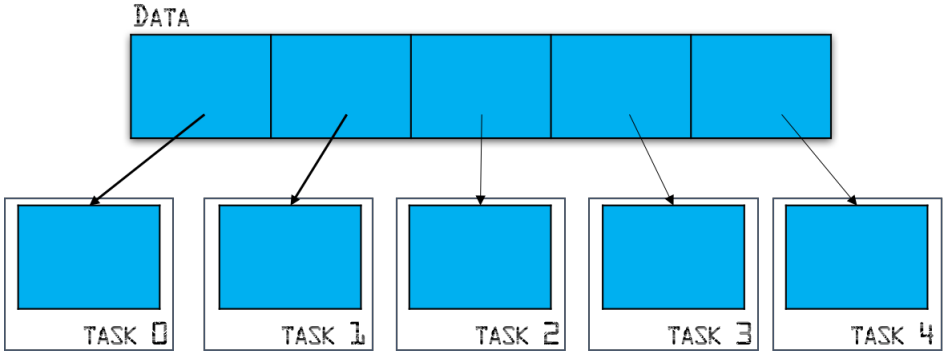
Please be aware that many online resources exist for parallel programming, which can often include code examples. We urge you to hold yourself to high academic standards and focus on the learning aspects of these assignments. It is less important to get the exact right answer than to learn the concepts of parallel programming. That being said, we will be using the MOSS program (Measure Of Software Similarity) on independent code sections that you turn in. It is important that this assignment be done independently.

Background Reading (Refresher)

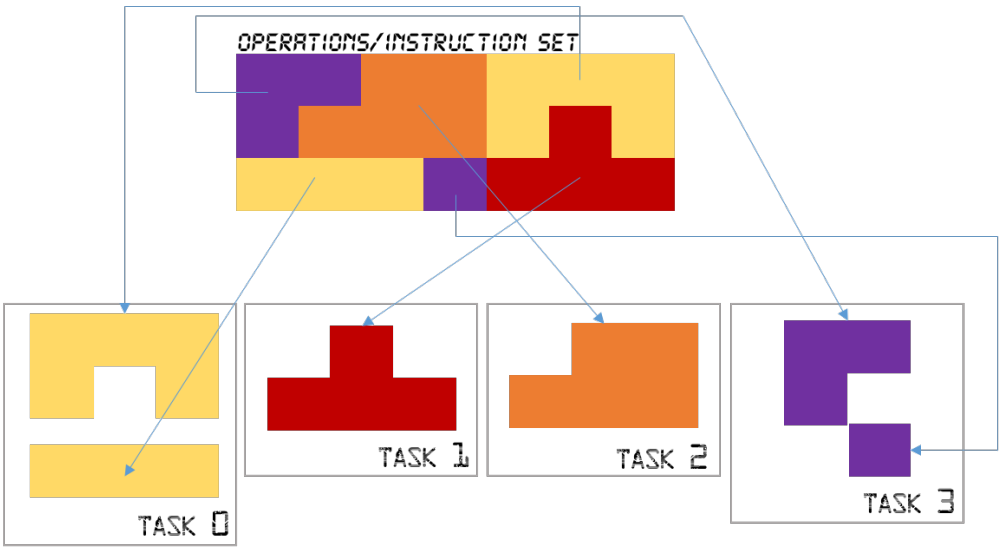
Please read the following section carefully, as it will be helpful for answering the rest of the problems in this assignment.

<u>Reviewing Basic Parallel Concepts</u>

1. ***Decomposing a Problem into Parallelizable Segments***
    1. *Domain Decomposition* focuses on the data (domain) to be operated on by an application. Each parallel task can operate on a section of the data in order to achieve a solution. Keep in mind that there are several ways to perform data decomposition (in multiple dimensions).



    2. *Functional Decomposition* focuses more on the compute tasks to be performed than on the data associated with the problem. This is particularly useful when there are several different computational tasks required to solve a problem, rather than performing the same task on a different section of data.



2. ***Performance*** You are probably already familiar with calculating the performance of your code from your undergraduate courses. We will briefly review these concepts and further extend them to parallel computing. You will then use these concepts to reason about the parallel code you write in this assignment.

Key Elements of Performance Analysis

A. Amdahl's Law

a. Theoretical Speedup is given by Amdahl's Law, where p is the fraction of the problem that is parallelizable.

$$speedup = \frac{1}{1-p}$$

$$p = 0 : None\ is\ parallelizable$$
$$p = 1 : All\ is\ parallelizable$$

b. Speedup is defined as the time, T_S, it takes a program to execute in serial (with one processor) divided by the time, T_P, it takes to execute in parallel (with n processors).

$$speedup = \frac{T_S}{T_P}$$

Considering that solving a given problem, GP, consists of the parallelizable fraction, p, and the sequential fraction, s, then $s + p = 1$. It follows that:

$$speedup_n = \frac{1}{\left(\frac{p}{n}\right)+s} = \frac{1}{\left(\frac{p}{n}\right)+(1-p)}$$

Reminder: n is number of processors/cores

B. Parallel Efficiency

$$Efficiency_n = \frac{speedup_n}{n} = \frac{T_S}{nT_P}$$

C. Gustafson-Barsis' Law: Amdahl's Law is based on the notion of a fixed problem size. However, today's scientific problems are often constrained to a specific size because of the limitations of the underlying hardware. For example, computational chemists were at one time constrained by hardware resources to simulate only 'small' proteins (e.g., hemoglobin, with roughly 10k atoms) when performing molecular dynamics simulations. However, scientists today often run simulations of molecules with millions and billions of atoms (the world record is currently ~4 trillion atoms). The underlying goals of the problem are the same, but enhancements in hardware have enabled scientists to broaden the problem size. Gustafson-Barsis' Law attempts to account for this by providing a measure of potential speedup for a fixed execution time, given improved resources.

Caveat: *For the law to hold, the serial work must grow significantly slower than the parallel work, and the overheads imposed by synchronization, etc., are scalable.*

$$speedup = 1 - p + np$$

Remember:

- Amdahl's Law: fixed workload
  - *Strong Scaling:* How quickly can we compute the same workload by increasing the number of processors?
- Gustafson-Barsis' Law: fixed execution time
  - *Weak Scaling:* How can we analyze a larger data set in roughly the same amount of time? In this case, the problem size per processor stays fixed but the number of processors increases.

3. Shared Memory Model You may remember the concepts of shared memory from your computer architecture or operating systems classes. In this model, processors have access to a pool of shared memory. Most modern high-performance systems contain a mixture of shared and distributed memory, i.e., processors on the same node have access to shared memory, a high-speed network provides global access to memory that is distributed (that is, on other nodes) but further communication is required to exchange data between nodes.

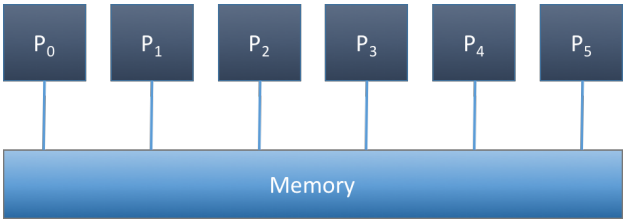
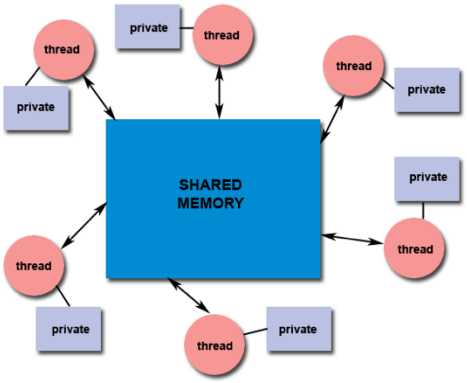


Figure 1, Simplistic View of Processors and Shared Memory

An important advantage of shared memory is that the data within memory is accessible to all of the processors connected to it. Thus, P_0 can access P_4's data without the need for a formal request/exchange, thereby simplifying the code that the programmer has to write. Access is typically controlled through locks/semaphores (also prevents deadlocks and race conditions). However, using shared memory also introduces complexity when it comes to data locality, e.g., multiple processes accessing and potentially updating the same data means that data cannot be as easily cached by the processors and, as a result, more accesses to main memory are often required in order to update the processor caches. In addition, there is increased bus traffic to the memory. We will explore various tools for users that make programming for shared memory environments easier and attempt to mitigate some of the locality issues.

1. Threads Threads are a useful way of programming in shared memory environments. In this model, the main process may spawn 'sub-procedures' (threads) that will run simultaneously and/or independently of the main execution. In this model, all threads have common access to the shared memory and also have private memory which is local to only that thread.



4. Environment Setup

You will be working on the ELF cluster. You can ssh to the elf cluster by using the following command: ssh <netid>@elf.rdi2.rutgers.edu. For more information about the ELF cluster, please see: http://rdi2.github.io/elf1-user-manual/. Homework will be turned in via Sakai. When this assignment asks for you to run code, it is assumed that you will be running this code on the ELF system. You may run it using a formal batch script or in an interactive session.

Important: Try starting a 1-node interactive session with the scheduler (SLURM). Collect some information about your ELF node using basic UNIX commands (for example, hostname, nproc/lscpu, etc.). Please post (as **Problem 0**) in your homework: the commands used and the output collected.

For standardization and library compatibility purposes, we will be using C/C++ for the example code in this problem set. Please submit your answers using either of the two languages.

POSIX threads (pthreads) provide a straightforward way to implement shared memory programs. In order to use pthreads, you must include the library in your program: #include <pthreads.h> On some architectures, you may also need to manually link the pthreads library when compiling (i.e., gcc –lpthreads myThreads.c).

Problem 1: Performance Modeling (Due Friday 10/07/16)

A. Calculate:

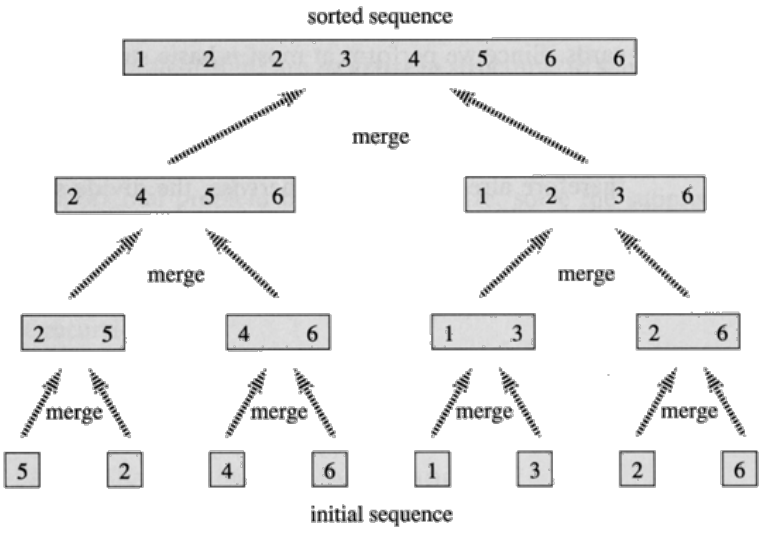
Ocean modeling is an important high-performance computing application that involves a large number of calculations on large, multidimensional datasets. Assume that a sequential application, OceanApp, currently runs in linear time (microseconds). A CS671 student parallelizes OceanApp and achieves a runtime of $\frac{n}{p} + log_2(p)$ microseconds, where p represents the number of processors. Suppose the student wants to run the code on the ELF cluster in order to increase the computing power (number of processors) by a factor of q [assume nodes and network costs are negligible].

i. Write the expression for the speedup achieved. Does this calculation hold if the processors are non-uniform? Why or why not?
ii. In order to achieve constant efficiency as we scale up, how much would we need to increase the problem size by?
iii. If linear speedup is achieved for some program, P, does that imply that P is strongly scalable? Explain your answer.

B. Program:

You have probably used the UNIX command time to capture the runtime of sequential applications you have written. time is also applicable to threaded programs, but the fact that it is limited to millisecond resolution can present a challenge for smaller workloads. Additionally, you may want to collect more fine-grained information from a thread. Fortunately, POSIX realtime extensions can be used to provide a more accurate timing analysis. You will need to include the time library (#include <time.h>) and tell the compiler to link it by passing '–lrt' at compile time.

The following example is a simple sequential implementation of the mergesort algorithm. If you need to refresh your memory on the mergesort algorithm, please see: http://algs4.cs.princeton.edu/22mergesort/ Mergesort is a good candidate for parallelization because it is a divide-and-conquer algorithm. It is also a good example with which to learn the basics of pthreads, because there are no synchronization or load balance issues.



Sequential Mergesort Function[1]

```
void merge (int *a, int n, int m) {
   int i, j, k;
   int *x = malloc(n * sizeof (int));
   for (i = 0, j = m, k = 0; k < n; k++) {
      x[k] = j == n      ? a[i++]
           : i == m      ? a[j++]
           : a[j] < a[i] ? a[j++]
           :               a[i++];
   }
   for (i = 0; i < n; i++) {
      a[i] = x[i];
   }
   free(x);
}

void merge_sort (int *a, int n) {
   if (n < 2)
      return;
   int m = n / 2;
   merge_sort(a, m);
   merge_sort(a + m, n - m);
   merge(a, n, m);
}
```

What you need to do:

i. Write a program, seqMS.c, that calls the above function for a given dataset. How long does your program take to run for varying input dataset sizes?

ii. Provide pseudo-code for a parallelized mergesort.

iii. Implement your parallel mergesort algorithm in C using pthreads. Be sure to include timing code.

iv. Collect timing information from running the serial and parallel codes multiple times. Vary the number of threads, data set size, etc. What is the speedup observed at varying thread counts with respect to the original sequential implementation? Plot the speedup at increasing core counts with respect to a fixed data size. Repeat for two more fixed data sizes of your choosing. Is the parallel code always faster?

v. Does one thread necessarily guarantee one core? What effects, if any, would hyperthreading have on the mergesort algorithm?

[1] Code credit: https://rosettacode.org.

Problem 2 (Due Friday 10/14/16)

The Mandelbrot Set & pthreads

Note: This problem has been reproduced, with permission, from CMU 15-418/618. The code is authored by Professor Kayvan Fatahalian. Answers are not permitted to be posted on any public websites. It will be considered a violation of Academic Integrity if anyone is found posting the answer or soliciting answers from individuals on the Internet.

The Mandelbrot set is generated from a simple function: $f(x) = x^2 + c$, *where c is a constant,* with stunningly beautiful results, as shown below. Build and run the code in the assignment directory (a1/code/) directory of the Assignment 1 code base. This program produces the image file mandelbrot-serial.ppm, which is a visualization of a famous set of complex numbers called the Mandelbrot set. As you can see in the images below, the result is a familiar and beautiful fractal. Each pixel in the image corresponds to a value in the complex plane, and the brightness of each pixel is proportional to the computational cost of determining whether the value is contained in the Mandelbrot set. To get image 2, use the command option --view 2. (See function mandelbrotSerial() defined in mandelbrotSerial.cpp). You can learn more about the definition of the Mandelbrot set at http://en.wikipedia.org/wiki/Mandelbrot_set. Mandelbrot serves as a good example for practicing work decomposition, as the workload is not uniform between threads.

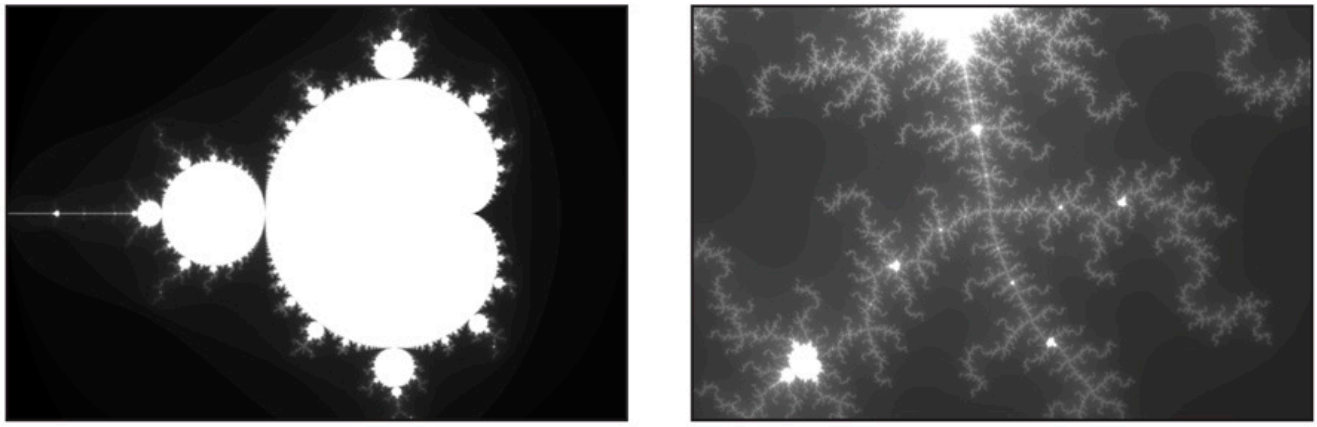


Figure 2, A visualization of the Mandelbrot set. The cost of computing each pixel is proportional to its brightness. When running programs 1 and 3, you can use the command line option --view 2 to set output to be view 2.

Your job is to parallelize the computation of the images using pthreads. Starter code that spawns one additional thread is provided in the function mandelbrotThread(), located in mandelbrotThread.cpp. In this function, the main application thread creates another additional pthread using pthread_create. It waits for this thread to complete using pthread_join. Currently the launched thread does not do any computation and returns immediately. You should add code to workerThreadStart function to accomplish this task. You will not need to make use of any other pthread API calls in this assignment.

What you need to do:

1. Modify the starter code to parallelize the Mandelbrot generation using two processors. Specifically, compute the top half of the image in thread 0, and the bottom half of the image in thread 1. This type of problem decomposition is referred to as *spatial decomposition* since different spatial regions of the image are computed by different processors.
2. Extend your code to utilize 2, 3, and 4 threads, partitioning the image generation work accordingly. In your write-up, produce a graph of **speedup compared to the reference sequential implementation** as a function of the number of cores used FOR VIEW 1. Is speedup linear in the number of cores used? In your write-up hypothesize why this is (or is not) the case? (you may also wish to produce a graph for VIEW 2 to help you come up with an answer.)
3. To confirm (or disprove) your hypothesis, measure the amount of time each thread requires to complete its work by inserting timing code at the beginning and end of workerThreadStart(). How do your measurements explain the speedup graph you previously created?
4. Modify the mapping of work to threads to achieve to improve speedup to at **about 3.5x on both views** of the Mandelbrot set (if you're close to 3.5X that's fine, don't sweat it). You may not use any synchronization between threads. We are expecting you to come up with a single work decomposition policy that will work well for all thread counts---hard coding a solution specific to each configuration is not allowed! (Hint: There is a very simple static assignment that will achieve this goal, and no communication/synchronization among threads is necessary.). In your write-up, describe your approach and report the final 4-thread speedup obtained.

Problem 3 (Due Friday 10/14/16)

The Mandelbrot Set & OpenMP

Pthreads provide low-level integration with system calls and a high degree of freedom for programming virtually every detail of parallel behaviors. However, this freedom comes at a cost – not only is it possible to program all of these details, it is required. This can make it cumbersome to use unless the problem calls for fine-grained control over thread operations (like create/destroy, join, etc) and locking mechanisms (mutex).

That's where OpenMP comes in - OpenMP is another API for shared memory programming. It is more of a user-friendly approach to shared memory but at a much higher level. It is also useful in that it can be used in a wider variety of programming languages and certain thread behavior can be left up to the compiler and runtime systems instead of being explicitly programmed. Another pro to using OpenMP is that when compiling a program (compiler permitting), if OpenMP is not usable for some reason, the code can be compiled and run in serial.

The level at which OpenMP resides is certainly not ideal for every shared memory problem, but its ease of use and programmability have contributed to it having a wider adoption in the programming community. OpenMP employs a fork/join model. Working at a higher level allows us to explore some features in more depth than for pthreads, since it is less work to implement them. OpenMP provides 'API-level' calls for programmers to easily manage critical or atomic sections/thread-safety, locking (mutexes/semaphores), workload distribution (load balancing, scheduling, etc.), variable scope, reduction, etc. Although this homework assignment will not touch upon all of these features, it is beneficial to at least familiarize yourself with them through independent reading.

The previous problem introduced you to the computation of the Mandelbrot set and workload distribution using pthreads. This example will extend the Mandelbrot set programming exercise to OpenMP, as well as get you thinking about aspects of potential optimization. Oftentimes in parallel programming, breaking up the problem into parallelizable chunks is a very manageable task. Trickier though is optimizing your parallel algorithms to reduce overheads and idling, eliminate race conditions, ensure thread-safety, improve scalability, and achieve faster end-to-end runtimes. Although it can be intimidating at first, optimization of parallel algorithms and code is an extremely valuable skill that we will come back to many times in this course.

What you need to do:

1. Write a C program that computes the Mandelbrot set on a given discretized domain of the complex plane in parallel using OpenMP. Repeat the experiments performed in the pthreads example. Plot your speedup results for pthreads, OpenMP, and serial implementations on the same graph. Generate multiple plots for different data sizes (i.e., each plot will contain three datasets [pthreads,OpenMP,serial] for a fixed data size). Which gives the best performance? Explain why you think that is.
    i. Recall (from above) that the Mandelbrot is a set of points in a complex plane with the property that $z(n+1) = z(n)^2 + c$ remains bounded.

2. What different OpenMP parallelization schedules could you employ in your code? Would this have an effect, if any, on your code's runtime? Please provide a justification for your answer. For more information on scheduling, please see: http://cs.umw.edu/~finlayson/class/fall16/cpsc425/notes/12-scheduling.html
3. Perform a scalability analysis of your code, by studying both strong and weak scaling.
4. Provide one example of a real-world parallelizable problem and explain whether you would use domain or task decomposition to separate the workload. Is your problem inherently load balanced? If you wrote code to model your example, what would the shared data / critical sections be? What portions leave room for optimization?