Chapter 2 How to Design a Parallel Program

Mike Rogers, Sheikh Ghafoor, and Mark Boshart
Tennessee Tech

2.1 Introduction

Most computers sold today have more than one processor. A computer with more than one processor can do more than one operation at a time, and doing more than one operation at a time results in programs running faster. Unfortunately, a program written for a computer having only one processor does not *automatically* run faster. The program must be written to use the processors.

This chapter explains the concepts needed to write parallel programs that can run faster. In particular, this chapter introduces the concept of threads, describes how to use them to write parallel programs, and provides some detailed examples.

2.2 The Concept of Threads

The most common models for parallel programming make use of the concept of *threads*. To understand threads, you must also understand the related concepts of *process*, *race condition*, *deadlock*, *shared resources*, and *parallel speedup*. Following is a simple grocery store illustration to help describe the concepts, followed by a more detailed description of programming using threads.

2.2.1 Parallelism in the Grocery Store

Consider the following example of a grocery store that illustrates the concepts of threads and processes. The grocery store (*process*) has many people inside that are shopping for items that they need. Most of those people are not collaborating. For example, one shopper can be shopping for items that they need for their evening meal, while another shopper can be shopping for items that they need for their daughter's birthday party. In this case, both shoppers are acting independently of each other. However, a husband and wife team could decide to split up their shopping list such that the husband searches for the meat items for the evening meal while the wife searches for the pasta items for their evening meal. In this case the husband and wife team are sharing the load of shopping. Note that they must agree on the workload, i.e. *who* does *what*.

The above illustration describes a number of concepts for threading. First, a *process,* which is represented by the grocery store in the example above, is a set of data and instructions that is given its own memory by the operating system, and is created any time that you execute a program on your computer. Another way to describe a process is that it is an instance of a running program. A process can contain any number of threads, which are illustrated as customers, that act simultaneously. The threads are what execute the process's code, just as the customers walk the grocery store's aisles. The threads can be acting independently or in concert. If threads act in concert, then those threads must be programmed such that they collaborate. In

This effort is partially funded by the National Science Foundation award: OAC 1549812

other words, the programming of the threads must break up the workload such that each thread works on its part.

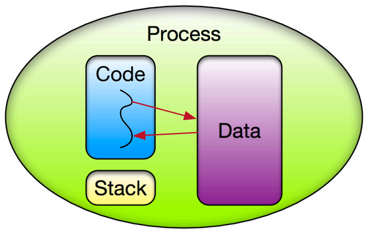
Consider what happens in the grocery store illustration if two customers reach for the same box of spaghetti. This condition is called a *race condition*. If the customers do not have some way to resolve the impasse, then disaster can occur. Perhaps the customers will get in a tug-of-war match until the box of spaghetti bursts and makes a mess on the grocery store floor, or the customers become *deadlocked* because they continue to fight over the single box of spaghetti and never finish their shopping. However, the conflict could be resolved if the customers, upon reaching the shelf that has the boxes of spaghetti, agree to solve their conflict by taking turns such that one customer takes a box, and then the second customer takes a different box.

Similarly, threads accessing *shared resources* can lead to race conditions and deadlock. Shared resources are resources that can possibly be accessed by two or more threads at the same time. To avoid disaster in a program, such as the program crashing or giving wrong answers, the threads must follow a *synchronization* protocol, such as taking turns.

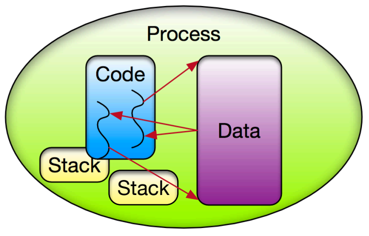
Also note that, in the case of the husband and wife team, they were able to finish their shopping faster. In fact, if they split their shopping list into two equal sized lists of items, then they would be able to finish their shopping in about half the time, given that each item was equally easy to find. If the team brought their children to help them shop, then they could divide the shopping list further and save even more time. Similarly, the term *parallel speedup* describes the performance increase that occurs when multiple threads computing simultaneously results in making a computation take less time. However, note that speedup may not be perfect. For example, if you have 10000 items on your grocery list and send 10000 people to gather the items in the grocery store, the people may have to wait for others to leave a crowded grocery store aisle before they can get to their items. In other words, according to *Amdahl's law*, speedup is limited because the amount of parallelism is limited.

2.2.2 Programming Using Threads

Understanding how threads are implemented by your operating system is helpful for understanding how to program using threads. Therefore, consider Figure 1a. A process is made up of a program's code, memory that stores the program's data, and another type of memory called the *stack* that helps the program keep track of local variables and function calls. The curvy line in Figure 1a represents a single thread of control. The thread can access the process's data and use its stack to create and manipulate local variables and call procedures.



a) A process with one thread.



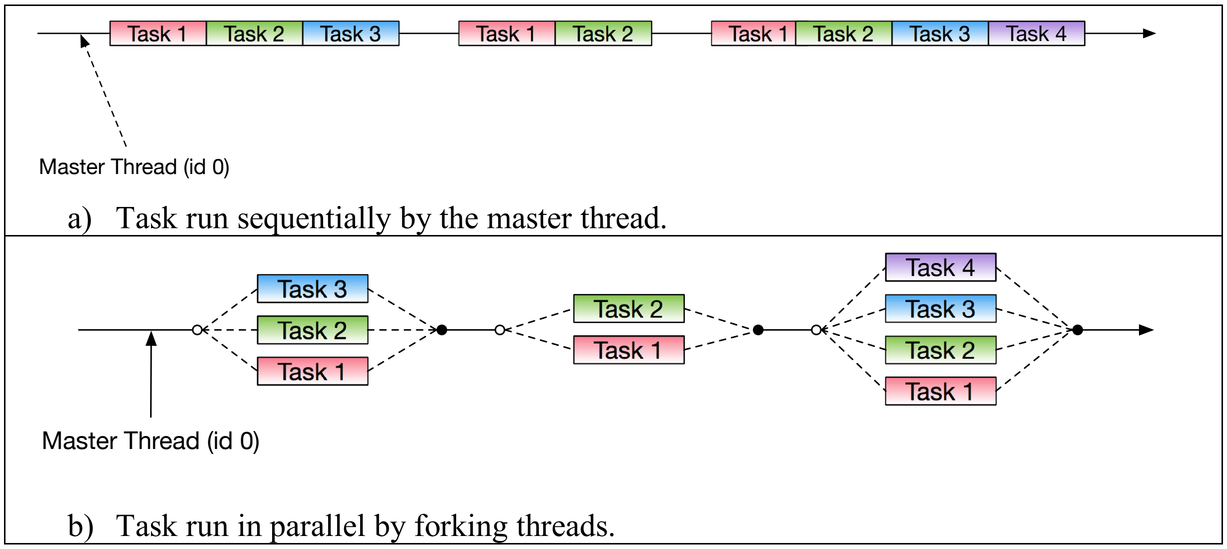
b) A process with two threads

This effort is partially funded by the National Science Foundation award: OAC 1549812

Figure 1: Processes and threads.

Figure 1b shows a process with two threads, represented by two curvy lines. Notice that both of these threads can access the process's data simultaneously. Such memory is called *shared memory* and is a shared resource, as discussed in the grocery store example. Furthermore, each thread has its own stack that is not shared so that it can have its own local variables and keep track of its own function calls. The threads can run the same code or the two threads can run two different functions at the same time. Furthermore, a process can have any number of threads, where each thread executes program code and has its own local variables. So, programs can create multiple threads, and all of those threads can run code and access the process's data simultaneously, which can result in faster computation!

The details of writing programs that use multiple threads to solve problems can vary according to the programming language and threading library used. However, all languages and libraries have similar concepts and follow similar approaches to thread programming. Consider Figure 2 that shows a single threaded process and a multithreaded process. A *task* is defined as a unit of work that can be executed by a thread. A task is usually implemented as a function or code block in a program. Normally, a program starts with a single thread of execution. Therefore, as depicted in Figure 2a, the single master thread, usually given an identifier of 0, must executed each tasks sequentially.



a) Task run sequentially by the master thread.

b) Task run in parallel by forking threads.

Figure 2: Parallel tasks.

However, some tasks can be executed simultaneously if the process had more than a single thread. Multithreading libraries and thread capable programming languages allow the program to do just that. For example, in Figure 2b the process starts with only the master thread. However, tasks 1, 2, and 3 can be executed in parallel. Therefore, the master thread *forks* three threads that each run their tasks in parallel. Note that the three tasks could run the same code or they could run different code. What is important is that the threads are running the tasks simultaneously. Once all threads are finished, the threads *join* at a *synchronization point*, often called a *barrier*. Then the master thread continues. As the master thread continues to run the process code, if it needs to execute more tasks in parallel, then it follows the same fork-join scheme.

This effort is partially funded by the National Science Foundation award: OAC 1549812

A common computational problem is scalar-vector multiplication, and it is a good example to demonstrate the fork-join threading model. Scalar vector multiplication requires multiplying each element of a vector, which can be represented in most programming languages as an array, by a scalar value. Following is a simple sequential algorithm for solving the scalar-vector multiplications problem for a floating point scalar and an array of floats:

```
scalar_vector_multiply
    inputs:
        scalar:  a floating point number
        vector:  an array of floating point numbers
    outputs:
        none - the vector is modified in-place
    begin
       loop variable i from 0 to number of elements in vector
           set $vector_i$ to $vector_i$ * scalar
       end loop
    end
```

Unfortunately, the above algorithm will not run any faster if it is executed on a machine with multiple processors that can run multiple threads. An algorithm that uses multiple threads must divide the work between the threads, fork the threads, and then merge the results, if necessary.

In the scalar-vector multiplication example, dividing the work means determining which elements of the vector will be computed by which threads. Typically, the best way to divide work is to assign each thread an equal portion. For example, if the vector has 100 elements and the process is to fork 4 threads, then the process's code should assign each thread 25 elements. Given that each thread will have a numeric identifier, i.e. the master thread will have id 0, the next thread will have id 1, and so on, then the process's code should assign the elements 0 through 24 to thread 0. The process should assign elements 25 through 49 to thread 1, elements 50 through 74 to thread 2, and elements 75 through 99 to thread 3. Given this scenario, the algorithm for a parallel version of `scalar_vector_multiply` would be as follows.

```
parallel_scalar_vector_multiply()
    inputs:
        scalar:  a floating point number
        vector:  an array of floating point numbers
    outputs:
        none - the vector is modified in-place
    begin
       set chunk_size to 25
       with 4 threads do
          set thread_id to the current thread's id
          set local variable vstart to thread_id * chunk_size
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
        set local variable vend to ((thread_id+1) * chunk_size) - 1
        loop variable i from vstart to vend
            set $vector_i$ to $vector_i$ * scalar
        end loop
     end with
  end
```

In the above algorithm, the `with..do` construct forks the threads, and the threads join after the `end with`. You should trace through the above algorithm and convince yourself that it works.

In the case of the scalar-vector multiply algorithm, once the algorithm is finished, nothing more needs to be done. However, for some algorithms, extra steps must be taken to merge the results from each thread. For example, consider the algorithm below that finds the smallest value in a vector using two threads.

```
1     Parallel_find_smallest()
2        inputs:
3           vector:  an array of floating point numbers
4           vector_size: the number of elements in vector
5        outputs:
6           least: the smallest number in vector
7        begin
8           with 2 threads do
9             set thread_id to the current thread's id
10            if thread_id is 0 then
11               set local variable vstart to 0
12               set local variable vend to vector_size / 2
13               set $smallest_0$ to $vector_{vstart}$
14            else
15               set local variable vstart to (vector_size / 2) + 1
16               set local variable vend to vector_size – 1
17               set $smallest_1$ to $vector_{vstart}$
18            end if
19            loop i from vstart to vend
20               if $vector_i$ is less than $smallest_{thread\_id}$ then
21                  set $smallest_{thread\_id}$ to $vector_i$
22               end if
23            end loop
24          end with
25          set least to the minimum of $smallest_0$ and $smallest_1$
26       end
```

By line 24, each thread has calculated the smallest value in its part and stored that value in the `smallest` array at the index that corresponds the thread's id. However, the smallest value in the

This effort is partially funded by the National Science Foundation award: OAC 1549812

whole array must be the smaller of the two values in smallest$_0$ and smallest$_1$. Therefore, to make the above algorithm complete, line 25 computes the minimum value of the two thread's results. Note that line 25 occurs after the with…do construct, which is after the threads join, and only the master thread is active.

2.3 Thread Libraries

This chapter continues the topic of threads by showing how the concepts of threads are implemented in actual programming code. This chapter describes both C++ with OpenMP and Java's **ForkJoin** framework and gives examples.

2.3.1 C/C++ and OpenMP

Any programming language that supports multithreading must support creating threads, assigning them work, and synchronizing them. Many ways exist for the C and C++ programming to control threads. This section describes one of the easiest to use threading libraries for C/C++ called OpenMP.

2.3.1.1 *OpenMP Basics*

OpenMP was designed so that programmers could put their efforts into designing their programs instead of the complexities of parallel programming. Therefore, OpenMP can parallelize most code by simply annotating the code with compiler directives. In C/C++, a compiler directive is a line in the source code that starts with the pound sign (#) and the word `pragma`. The specific compiler directive for OpenMP is as follows.

```
#pragma omp
```

Adding specific keywords at the end of the pragma determines the pragmas behavior. For example, consider the following C++ hello world example.

```
#include <iostream>
#include <string>
#include <sstream>
#include <omp.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
     std::cerr << "usage: " << argv[0] << " num_threads" << std::endl;
     exit(1);
  }
  int nthreads;
  std::stringstream ss(argv[1]);
  ss >> nthreads;
#pragma omp parallel num_threads(nthreads)
  std::cout << "Hello from thread "
            << omp_get_thread_num() << std::endl;

        return 0;
}
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

So, what does this code do? In short, it spawns two threads and each thread prints out a hello message followed by its thread identifier. In detail, Lines 1 through 4 include some needed headers. In particular, line 4 includes OpenMP definitions. Line 4 is needed to be able to call the omp_get_thread_num() function found on line16. Lines 12 and 13 convert the string value passed on the command line to an integer and store that number into the variable `num_threads`. Then line 14 forks multiple threads, using the value in `num_threads` to determine how many. The use of the pragma keyword `parallel` designates that the next line of code is to be run by each thread in parallel. Lines 15 and 16 prints the string "Hello from thread" followed by the threads identifying number that is returned from the `omp_get_thread_num()` function. This identifying number is often called the thread's *rank* or *id*. Once the parallel code is executed, the threads join at what is called in *implicit barrier*, and only the master thread executes line 18.

What if the thread should execute more than one line of code, i.e. a code block, after the pragma? Surrounding the code with curly braces create a code block that is executed in parallel, as in the following.

```
#pragma omp parallel num_threads(threads)
{
        std::cout << "Hello from thread ";
        std::cout << omp_get_thread_num()
                  << std::end;
}
```

2.3.1.2 *Running and executing the code*

Entering the following command in a terminal (Unix) or a command prompt (Windows) will compile the above program using the Gnu C++ compiler.

```
g++ -Wall -fopenmp -o hello hello.cpp
```

The `-fopenmp` flag must be included to be able to use OpenMP. Then, the program can be run by entering the following command.

```
./hello 2
```

The "./" in front of the program name should be removed to run the program at the Windows command prompt. The following is example output of the executed program.

```
$ ./hello 2
Hello from thread Hello from thread 01
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

Why is the program output garbled? It is garbled because threads in the program are sharing the screen. Thus, a race condition exists because each thread is contending for the shared resource. The pragma `critical` will fix the race condition by requiring the threads to take turns running the code in the critical section, as in the following C++ code snippet.

```
#pragma omp parallel num_threads(nthreads)
  {
    std::cout << "This line is a race condition " << std::endl;
#pragma omp critical
    std::cout << "But this line is not: " << omp_get_thread_num() <<
std::endl;
  }
```

The following is an example of output when executing a program that has the above code in the `main()`.

```
This line is a race condition This line is a race condition

But this line is not: 1
But this line is not: 0
```

2.3.1.3 *A Practical Example*

A more practical, but still simple, example is parallelizing the function to find the smallest value in an array. The program code below is a first attempt.

```
#include <iostream>
#include <omp.h>
float parallel_find_smallest(float vector[], int size);

int main() {
  float vector[17] = {1,9,2.3,4,3,6,5,8,7,9,12.2,65,24,
                      -20.4,9,46,123};
  std::cout << "Smallest is: " << parallel_find_smallest(vector, 17)
          << std::endl;

  return 0;
}

// The function below only works for two threads
float parallel_find_smallest(float vector[], int size) {
  float smallest[2];
  int vstart[2];
  int vend[2];

#pragma omp parallel num_threads(2)
  {
    int id = omp_get_thread_num();
    if (id == 0) {
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
24        vstart[0] = 0;
25        vend[0] = size/2;
26      } else if (id == 1) {
27        vstart[1] = size/2;
28        vend[1] = size-1;
29      }
30      int i = 0;
31      smallest[id] = vector[vstart[id]];
32      for (i = vstart[id] + 1; i <= vend[id]; i++) {
33        if (vector[i] < smallest[id]) {
34        smallest[id] = vector[i];
35        }
36      }
37    }
38    return std::min(smallest[0], smallest[1]);
39  }
```

This code only forks two threads to find the smallest number in the array. Each thread examines each element of its part of the array and keep track of the smallest element in its part. Then, the master thread examines the value from each thread and returns the smallest. In detail, lines 23 through 29 loads an array called vstart with the starting indexes for each thread and an array called vend with the ending indexes of the array for each thread. Lines 32 through 36 executes a loop over the elements for a particular thread's part. Line 38 then returns the smallest value found by the two threads.

Although the above example works and is a good example for how you can partition the array so that the threads can process the array in parallel, OpenMP provides a much simpler way to write the `parallel_find_smallest()` function. The simpler method is shown in the following example.

```
float parallel_find_smallest(float vector[], int size) {
  float least;

  int i = 0;

  least = vector[0];
#pragma omp parallel for reduction(min:least)
  for (i = 0; i < size; i++) {
    if (vector[i] < least) {
      least = vector[i];
    }
  }

  return least;
}
```

OpenMP handles both the partitioning of the array and the combining of the thread's results automatically in this version. Thus, this version is easier to write, smaller, and more readable.

This effort is partially funded by the National Science Foundation award: OAC 1549812

Line 17 uses the OpenMP `parallel for` pragma. When the compiler sees this pragma, it assigns each thread an approximately equal number of iterations of the loop and computes the corresponding start and end values for the loop variable. Additionally, the `reduction` pragma combines the resulting values from the threads stored in the `least` variable (each thread uses a thread local variable), using the operation specified, which is the `min` operation.

2.3.2 Java Threads

The Java programming language, like C++, also support using multiple threads. Fortunately, Java's **ForkJoin** Framework handles most of the complexity for the programmer.

2.3.2.1 *Java Parallel Programming: The ForkJoin Framework*

The ForkJoin Framework is part of Java 8's standard libraries and helps you take advantage of multiple processors to improve the performance of your application. Tasks are distributed by the framework to worker "threads". The ForkJoin Framework uses a *work-stealing* algorithm. Worker threads that complete their assigned tasks can steal tasks from other threads that are still busy.

ForkJoin is a framework that implements an *inversion of control*. In other words, the programmer’s code does not call parallel tasks directly. The framework calls the parallel tasks. Therefore, the programmer must structure the tasks according to the rules of the framework. The ForkJoin framework is designed for work that can be decomposed into smaller tasks recursively. Therefore, code should adhere to the algorithm below[1].

```
if the problem size is “small”
     do the work right here
else
      break the problem into two or more smaller pieces (subtasks)
      start each piece in its own thread (fork)
      wait for the results (join)
```

The framework determines what resources (i.e. processors) are available and uses them as effectively as possible to complete the subtasks in parallel. Starting a task in its own thread is a *fork*, and waiting until the thread to finish its task is a *join*.

The first part of the above conditional is extremely important to get good performance from the ForkJoin framework. Creating and starting threads can be inefficient. A substantial amount of work is performed behind the scenes by the framework. Thus, it is important for the programmer to decide if a problem size is large enough to justify forking a new thread. This important decision may require some experimentation by the programmer. The conditional in the

[1] https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

This effort is partially funded by the National Science Foundation award: OAC 1549812

algorithm above must be carefully constructed to ensure that the program only forks new threads when it is advantageous to do so.

Not every problem can take advantage of multiple processors. In particular, each task must be independent of any of the other tasks results. In addition, how the framework will assign threads to the limited resources and which of the threads will finish first cannot be predicted. Thus, relying on predicting the order that the threads will complete can introduce race conditions that give incorrect results.

ForkJoin Framework Programming

In order to hook into the ForkJoin Framework (so that the framework can call the parallel tasks), the problem must be defined in a class that extends either **RecursiveTask** or **RecursiveAction**. The two are very similar except that for a RecursiveTask, each thread returns a result (object), but, for RecursiveAction, the threads do not. Both classes have a `compute()` method that the framework expects your subclass to override. The `compute()` method returns a value if you are extending RecursiveTask, and is void if you are extending RecursiveAction. The pseudocode in the previous section would be implemented in the `compute()` method for the specific problem.

The `compute()` method takes no parameters. Most likely, the recursive, serial version of the problem needs several parameters. The subclass (extending either RecursiveTask or RecursiveAction) can store the "parameters" that it needs as instance variables. Thus, starting a new thread will involve using `new` and passing the values that the thread needs to the constructor.

To start the new thread, the code should call the `fork()` method on the object just created. The subclass inherits functionality by extending RecursiveTask or RecursiveAction. Therefore, the `fork()` method calls the overridden `compute()` method on the task just started that, typically, starts more threads.

After forking all of the threads, the program must wait for the results. The code should call the `join()` method on the object that was forked earlier. This call to `join()` *blocks* (waits at that point in the code) until that specific thread finishes. The `join()` method will return an object if the problem extended RecursiveTask; otherwise, join() returns nothing (it has a void return value).

Blocking may at first seem inefficient. For example, suppose you fork five threads and your first call to join is on the first one you forked, but it is last to finish (remember you cannot predict the order that the threads will finish). Fortunately, the framework implements *work stealing*. Any threads that finish early will steal work from those that have not yet finished. Thus, the other four threads that finished before the first thread stay busy by taking work from the first thread. Therefore, Java's implementation of work stealing makes parallelizing code much simpler because the programmer does not have to ensure that each thread has approximately the same amount of work.

It is important to fork all threads before joining. Because a join blocks, forking a thread for a task and then immediately joining it before forking any other threads results in a serial solution

This effort is partially funded by the National Science Foundation award: OAC 1549812

to the problem. The code must fork several threads before joining in order for the framework to perform all of its optimizations, including work stealing.

Finally, the program must have a pool of threads available. The ForkJoin Framework makes this easy. The ForkJoin Framework has a singleton object, which the code can get by calling `ForkJoinPool.commonPool()`, that contains a thread pool that can be utilized by any task (RecursiveTask or RecursiveAction). This thread pool reduces the overhead of forking new threads. Passing the RecursiveTask or RecursiveAction that represents the entire problem (i.e. with instance variables corresponding to the entire task to be completed) to the thread pool object's `invoke()` method will start the program up in parallel. The example described next uses the thread pool aproach.

ForkJoin Framework Complete Example: Parallel Sum

As a simple, yet illustrative, example, consider a large array containing integers whose sum is required. The first step is to write a serial, recursive version of the algorithm, shown here.

```java
public class SumRecSerialOne
{
   public static long sumRec(Integer[] array, int first, int last)
  {
       if (last == first)
       {return array[first];}

       int mid = first + (last - first)/2;
       long left_sum = sumRec(array, first, mid);
       long right_sum = sumRec(array, mid+1, last);
       return left_sum + right_sum;
   }
}
```

The user calls `SumRecSerialOne.sumRec()`, providing the array and the range of indices to sum (most likely `first` is zero and `last` is `array.length - 1`). The problem is broken down into a left subtask and a right subtask, and a recursive call is made to solve each subtask. Note that this problem could have been broken into more than two pieces. The results from both subtasks are added together, and this result is returned.

Consider a second version of the same algorithm, below, where the parameters required by the method above become instance variables for a SumRecSerialTwo object.

```java
public class SumRecSerialTwo
{
   private Integer[] array;
   private int first;
   private int last;

   public SumRecSerialTwo(Integer[] array, int first, int last)
   {
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
      this.array = array;
      this.first = first;
      this.last = last;
   }

    public long sumRec()
   {
       if (last == first)
      {return array[first];}

      int mid = first + (last - first)/2;
      SumRecSerialTwo left = new SumRecSerialTwo(array, first, mid);
      long left_sum = left.sumRec();
      SumRecSerialTwo right = new SumRecSerialTwo(array, mid+1, last);
      long right_sum = right.sumRec();
      return left_sum + right_sum;
   }
}
```

The problem with this version is that a lot of new objects are created. Consider a third version, below, that combines the two previous versions.

```
public class SumRecSerialThree
{
   private Integer[] array;
   private int first;
   private int last;

   public SumRecSerialThree(Integer[] array, int first, int last)
   {
      this.array = array;
      this.first = first;
      this.last = last;
   }

   public long compute()
   {
      int num_items =  last - first + 1;

      if (num_items < SOME_NUMBER)
      {return sumRec(array, first, last);}
      else
      {return sumRec();}
   }

    public long sumRec()
   {
      int mid = first + (last - first)/2;

      SumRecSerialThree left = new SumRecSerialThree(array, first, mid);
      long left_sum = left.compute();
      SumRecSerialThree right = new SumRecSerialThree(array, mid+1, last);
      long right_sum = right.compute();
      return left_sum + right_sum;
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
    }

    public static long sumRec(Integer[] array, int first, int last)
   {
       if (last == first)
       {return array[first];}

       int mid = first + (last - first)/2;
       long left_sum = sumRec(array, first, mid);
       long right_sum = sumRec(array, mid+1, last);
       return left_sum + right_sum;
    }
}
```

This third version introduces a `compute()` method which is a *dispatch* method. If the problem size is "small", this dispatcher simply call the method that does not create new objects. Otherwise, `compute()` calls the method that creates new objects.

This serial version of the array summation program will be easy to convert to a parallel version. The following code applies the ForkJoin Framework requirements for our summation program:

```
import java.util.concurrent.RecursiveTask;
public class SumRecParallel extends RecursiveTask<Long>
{
   private Integer[] array;
   private int first;
   private int last;

   public SumRecParallel(Integer[] array, int first, int last)
   {
      this.array = array;
      this.first = first;
      this.last = last;
   }

   public Long compute()
   {
      int num_items =  last - first + 1;

      if (num_items < SOME_NUMBER)
      {return sumRec(array, first, last);}
      else
      {return sumRec();}
   }

    public long sumRec()
   {
      int mid = first + (last - first)/2;

      SumRecParallel left = new SumRecParallel(array, first, mid);
      left.fork();
      SumRecParallel right = new SumRecParallel(array, mid+1, last);
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
       right.fork();

       long left_sum = left.join();
       long right_sum = right.join();

       return left_sum + right_sum;
    }

    public static long sumRec(Integer[] array, int first, int last)
   {
       if (last == first)
       {return array[first];}

       int mid = first + (last - first)/2;
       long left_sum = sumRec(array, first, mid);
       long right_sum = sumRec(array, mid+1, last);
       return left_sum + right_sum;
    }
 }
```

Because our serial recursive methods returned a long, the code extends `RecursiveTask<Long>`. As in the serial version, the `compute()`method simply dispatches based on the current problem size. If the problem size is "small", `compute()` calls the `sumRec()` method that doesn't start any new threads. Finally, the `sumRec()` method without parameters, forks all of the new threads and then joins them.

Lastly, the `main()` method implements the pool of threads as shown in the following code snippet:

```
import java.util.concurrent.ForkJoinPool;
   ...
   SumRecParallel sum_rec_parallel =
          new SumRecParallel(array, 0, array.length - 1);
   long sum_parallel =
          ForkJoinPool.commonPool().invoke(sum_rec_parallel);
```

The finished program has a SumRecParallel object corresponding to the entire problem to be solved. The program passes this object to the invoke method of the ForkJoinPool.commonPool() object, and the parallelized array summation problem is complete!

2.4 Examples

This section contains various examples for both CS1 and CS2. Each example is written in a laboratory format so that the reader can completely implement the example. Therefore, each example lists the skills that the reader will learn once the reader completes the example, a prolog and review of prerequisite material in CS1 or CS2 needed for the example, a description of the

This effort is partially funded by the National Science Foundation award: OAC 1549812

problem that the example is attempting to solve, the methodology for parallelizing the example, and, finally, a description of the parallelized implementation.

2.4.1 Parallel Sum (CS1) - Summing numbers in parallel using C++ and OpenMP

Skills you will learn:

- Summing numbers in parallel
- Parallel Processing using OpenMP

Prolog and Review

For this example, you should review your CS1 course instruction on single dimension arrays and iterating over single dimension arrays. In particular, unlike scalar variables, such as variables of type int, double, and char, array variables can hold more than one value. Consider the following variable declarations.

```
1   int i = 5;
2   int numbers[3] = {10, 20, 30};
```

The variable declaration on line 1 of the above code declares a variable called `i` that holds the number 5, and that variable can only hold one number. However, the `numbers` variable declared at line 2 can hold 3 integers, and is initialized to hold the integers 10, 20, and 30. Accessing the integers is as simple as indicating the correct index. In C++, arrays are zero based, so the first item in the array always starts as index 0. So, in the above example, numbers[0] holds the integer 10, numbers[1] holds the integer 20, and number[2] holds the integer 30.

Once you have item stored in an array, you can iterate over those items using a simple loop structure and indexing into the array using the loop variable. Consider the following code:

```
#include <iostream>

const int NSIZE = 10;

void gen_numbers(float numbers[], int how_many);
float gen_rand(int min, int max);

int main() {
  float numbers[100];

  gen_numbers(numbers, NSIZE);

  std::cout << "Generated numbers are: " << std::endl;
  for (int i = 0; i < NSIZE; i++) {
    std::cout << numbers[i] << " " << std::endl;
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
  }

  return 0;
}

void gen_numbers(float numbers[], int how_many) {
  for (int i = 0; i < how_many; i++) {
    numbers[i] = gen_rand(0, 10);
  }
}

float gen_rand(int min, int max) {
  return (min + static_cast <float> (rand()) /
           ( static_cast <float> (RAND_MAX/(max-min))));
}
```

The above program defines a function called `gen_numbers()`. The `gen_numbers()` function generates an array of floating point numbers by using a `for` loop to place the random float, generated by calling `gen_rand()`, into the array. The `for` loop uses the loop variable `i` to index into the numbers array.

Problem Description

For this example, you will be writing a program to sum numbers stored in an array. Summing numbers is straightforward. Consider Figure 3 below. All that you need to do is use a loop that indexes into the array using the loop variable. You will also need an accumulator variable. The accumulator variable will hold the running sum for each iteration. Once all iterations are finished, the accumulator variable will hold the total sum.

array:										
	5.0	10.3	8.7	5.0	52.9	18.0	13.0	7.3	82.7	68.2

Loop step	Index	array[index]	Accumulator (sum)
initialize	0	----	0
first	0	5.0	0 + 5.0 = (5.0)
second	1	10.3	5.0 + 10.3 = 15.3
third	2	8.7	15.3 + 8.7 = 24.0
fourth	3	5.0	24.0 + 5.0 = 29.0
fifth	4	52.9	29.0 + 52.9 = 81.9
sixth	5	18.0	76.9+ 18.0 = 99.9
seventh	6	13.0	94.9 + 13.0 = 112.9
eighth	7	7.3	107.9 + 7.3 = 120.2
ninth	8	82.7	115.2 + 82.7 = 202.9
Tenth	9	68.2	197.9 + 68.2 = 271.1

Figure 3 Summing integers in an array.

This effort is partially funded by the National Science Foundation award: OAC 1549812

Convince yourself that the table is correct. Can you create an algorithm that does what the table depicts?

Methodology

You will use domain decomposition, also often called data decomposition, to sum the array of numbers in parallel. Domain decomposition requires dividing the array into equal parts and assigning each part to a thread. Consider the simple example in Figure 4. The figure depicts three processors each executing a thread, including a processor executing the master thread. The computation occurs in two phases. First, the work is divided equally among the three processors. In this case, each processor's thread sums four numbers in the array. The master thread on processor 1 sums all elements starting at index 0 and ending at index 3, the second thread on processor 2 sums all elements starting at index 4 and ending at index 7, and the third thread on processor 3 sums all elements starting at index 8 and ending at index 11.

The second phase occurs after all of the threads are finished with the first phase. In this phase, the master adds all of the sums, stored in the variables accum1, accum2, and accum3 in the figure, to compute a final total.

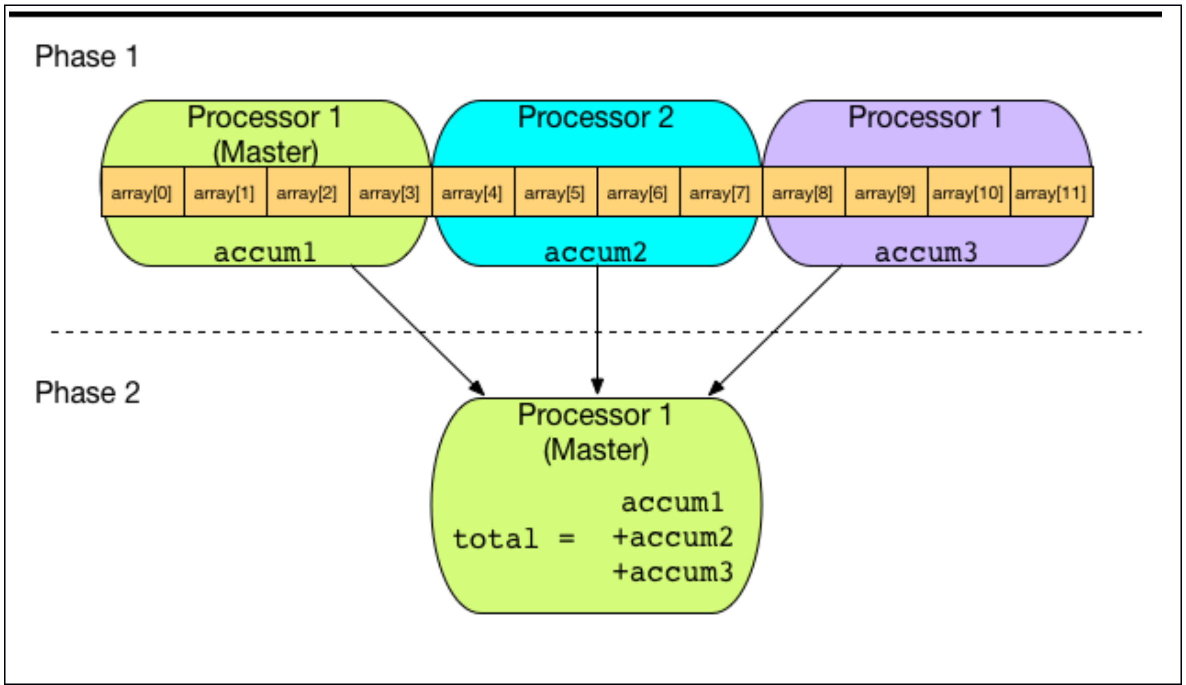


Figure 4: Three processors computing the sum in parallel.

This effort is partially funded by the National Science Foundation award: OAC 1549812

Implementation

For this example, you will be implementing the code to sum numbers in serial and then in parallel.

Tools

You will need to use the following tools to complete your assignment:

- An editor.
- A C++ compiler that is OpenMP capable (such as the gnu C++ compiler).

Generating Lots of Random Floating Point Numbers

You need lots of numbers to store in the array so that you can sort them. Normally, you would have some important data, perhaps stored in a file or database, that you need to sort. However, for this lab, you will generate some data with which to work. The section **Prolog and Review** included some functions that generate floating point numbers. You can use those functions to complete this example (`gen_numbers()` and `gen_rand()`).

Summing the numbers

Following is an algorithm that sums an array of numbers:

```
sum()
    inputs:
        array        - the array of numbers
        num_elements - the number of elements in the array
    returns:
        sum          - the sum of all the numbers in the array

    begin
       set sum to 0
       loop index from 0 to num_elements
          set sum to sum + the number in the array at position index
       end loop
       return sum
    end
```

Implement this algorithm as a C++ function and then write a `main()` that allocates an array of 1000000000 floating point numbers. If you allocate your array statically, you will need to make the array a global variable because an array that large may be bigger that the maximum size of the program's stack for your particular operating system. Call the `gen_numbers()` function to generate numbers and put them into the array. Finally, in `main()`, call the `sum()` function to sum the numbers, and then print the result. Put all of you code into a file called **sum_serial.cpp**. Compile it and run it to make sure it works.

This effort is partially funded by the National Science Foundation award: OAC 1549812

Summing in Parallel

Can you make the serial version faster? You can make it a bit faster by doing both the generating of the floating point numbers and the summing of the floating point numbers in parallel.

Copy the **sum_serial.cpp** file to a file named **sum_parallel.cpp**. Now, modify the new file to make a parallel version. The **Methodology** section above showed how the array should be decomposed for each processor in the system to compute a part of the array, and thus compute the parts in parallel. Fortunately, OpenMP will take care of doing all of the parllelization. All you need to do is annotate your C++ code with the proper compiler pragmas.

First, parallelize the number generation code. In the gen_numbers() function, add a line before the `for` loop. The line should be the following.

```
#pragma omp parallel for
```

Yes, that's it. Now the `for` loop has been properly parallelized.

Now, parallelize your `sum()` function. You can parallelize the for loop in the `sum()` function just as you did the for loop in the `gen_numbers()` function. However, you will not get the correct answer. Review the diagram in Figure 4 of the **Methodology** section. Each processor must have its own accumulator, and then the master processor must sum the individual accumulators together to get a final total. Combining the results of the computations is called a *reduction* is parallel programming. Fortunately, OpenMP will handle this operation for you as well. You just need to tell OpenMP what variable to reduce and what operation needs to be applied. Therefore, add the following line to your sum function directly before the for loop.

```
#pragma omp parallel for reduction(+:accum)
```

Note that the above line of code is valid if you named your accumulator variable `accum`. If you named your variable something else, which you probably did, then change the line to use your variable name instead of `accum`. Compile your code and run it to make sure it works.

So, What's the Difference?

To see how your code with the added OpenMP code makes a difference, you can add the

This effort is partially funded by the National Science Foundation award: OAC 1549812

following code before you call `gen_numbers()` in your `main()` function in both your serial and parallel version.

```
// Note that  the following code works for g++ on Linux, Mac OS, and Windows
(using MinGW)
struct timeval tv1;
struct timeval tv2;
gettimeofday(&tv1, NULL);
```

Next, add the following code immediately after your call the `sum()` function in `main()` for both the serial and parallel version:

```
gettimeofday(&tv2, NULL);
std::cout << "Total time: " << (double) (tv2.tv_usec - tv1.tv_usec) /
1000000 +
  (double) (tv2.tv_sec - tv1.tv_sec) << std::endl;
```

Run your code for both the serial version and the parallel version and compare the times. Try using larger images until you see speedup for the parallel version.

2.4.2 Image Processing (CS1) - Gray-scaling and Flipping an Image Using C++ and OpenMP

Skills you will learn

- Loading images into arrays
- Manipulating images
- Parallel Processing using OpenMP

Prolog and Review

For this example, you should review your course instruction on arrays, structures, and arrays of structures.   Unlike scalar variable, such as variables of type int, double, and char, array variable can hold more than one value. In addition, you are not limited to storing scalar values, such as integers, into arrays. You can also declare arrays that hold structures. Recall that a structure is a data type that can hold a set of specified variables. For example, if you want to create a new data type that holds a person's name, age, and salary, you could declare that data type as follows.

```
struct person_t {
  char name[30];
  int age;
  float salary;
};
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

Then, creating a variable of that type is as simple as the following.

```cpp
person_t joe = {"Joe Schmoe", 22, 20345};
std::cout << joe.name << "'s age is " << joe.age << std::endl;
```

Notice that after the variable joe is assigned a name, age, and salary, the above code accesses joe's name and age using dot notation. Creating and accessing arrays of structures simply combines array syntax with structure syntax. Consider the following.

```cpp
1 person_t people[2] = { {"Joe Schmoe", 22, 20345},
2                        {"Jane Doe",   23, 45323} };
3 std::cout << people[0].name << "'s age is " << people[0].age << std::end;
4 std::cout << people[1].name << "'s age is " << people[1].age << std::endl;
```

As you can see, the above code declares an array that can hold two structures of type `person_t`. The array is initialized with the name, age, and salary of each of the two people. Then, at line 3, the code prints the name and age of the `person_t` at index 0 of the array. Line 4 then prints the name and age of the `person_t` at index 1 of the array.

Problem Description

For this example, you will be writing program to manipulate images. You will write a serial version of the program and time it. Then you will write a parallel version of the program, time it, and then compare the speed of the two programs.

The two image processing techniques that you will implement are grey-scaling and flipping an image. *Figure* 5 shows examples of an images that has been gray-scaled, flipped, and then both gray-scaled and flipped.

Images are represented as pixels. You can think of a pixel and an individual color "dot" on your monitor screen. The color of the pixel is represented as a mixture of intensities of the colors red, green, and blue. Each intensity is represented as an 8-bit number in the ranging from 0 to 255. For example, the values (0,0,0) represents the color black, the values (255,0,0) represent the color red, and the values (255,255,0) represent the color yellow. We call these intensities RGB values (for red, green, and blue).

Gray-scaling an image represented as a series of RGB values is easy. Different methods exist, but an effective method is called the *luminosity* method. Given the *i*th pixel, you gray-scale that pixel with the following formula:

*gray_value[i]= 0.21 * pixel[i].red + 0.72 * pixel[i].green + 0.07 * pixel[i].blue*

This effort is partially funded by the National Science Foundation award: OAC 1549812

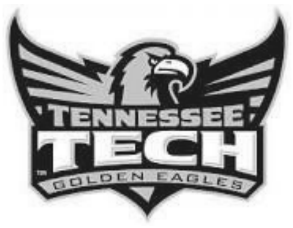
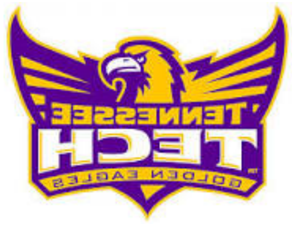
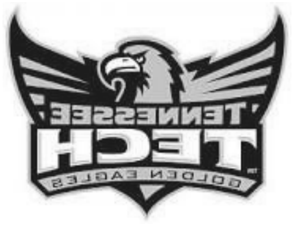
TENNESSEE TECH GOLDEN EAGLES a) Color image	 b) Gray-scaled image
 c) Flipped image	 d) Gray-scaled and flipped image

Figure 5: Flipping and Gray-scaling an image.

Then, for each *i*, set the red, green, and blue component of *pixel[i]* to *gray_value[i]* to gray-scale the image.

Flipping the image is also easy. Flip an image by flipping the 1st pixel with the last pixel, the 2nd pixel with the next-to-last pixel, the 3rd pixel with the second-to-last pixel, and so on.

Methodology

Given the overview of the gray-scale and flipping algorithms above, how do you gray-scale and flip an image in parallel? Consider the representation of an image in Figure 6.

An image has both a height and a width. The array of pixels shown is arranged in rows, where each row is a line of pixels that would appear across the screen. The size of each line of pixels is equal to the image's width, and the number of lines is equal to the images height.

When writing a parallel application, you first must determine how to divide the problem among the available processors. Dividing the problem requires determining 1) how much of the problem each processor should compute, and 2) determining where, in the input data, the processor should begin and end its computations. In general, when dividing the rows among processors, you should divide the work equally. So, if the image consists of *n* rows, and there are *p* processors available, then each processor should get roughly *n*/*p* rows.

This effort is partially funded by the National Science Foundation award: OAC 1549812

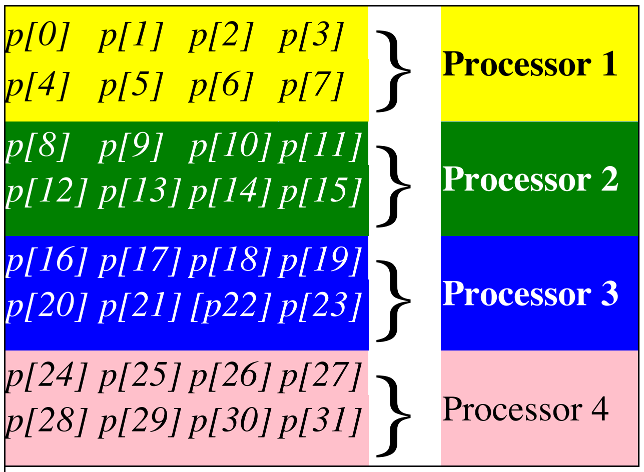


Figure 6: Processing an image in parallel.

A natural division for an image is to divide the image into chunks where each chunk consists of number of sequential rows of pixels. Then you assign each chunk of rows to a processor. So, in the example above, given four processors, each processor is responsible for two rows. Processor 1 would work on the 1st two rows (those rows in yellow) starting at index 0 and finishing with index 7, processor 2 would work on the third and fourth rows (those rows colored green) starting at index 8 and finishing at index 15, and so on.

Implementation

First, you will be implementing functions to read, write and gray-scale image files. You will then modify your code to that it processes the image files in parallel, thus speeding up the grayscale function.

Tools

You will need to use the following tools to complete your assignment:

- An editor.
- A C++ compiler that is OpenMP capable (such as the gnu C++ compiler).
- Some image files with which to work. The following image files are available online: **ttu.ppm**[2] **ttu_tile.ppm**[3]
- A program that can display PPM image files (for example, a browser).

Reading and Writing a PPM File

You will be reading and writing one of the simplest graphic file formats. The format is the Netpbm color image format, also known as the PPM format.

Reading the header

2 http://www.csc.tntech.edu/~mrogers/2100/ttu.ppm
3 http://www.csc.tntech.edu/~mrogers/2100/ttu_tile.ppm

This effort is partially funded by the National Science Foundation award: OAC 1549812

A PPM file consists of a header describing the file type, the image width, the image height, and the maximum color value per RGB component. The header is formatted as in the following table:

Description	Type	Size	Value
Magic Number that identifies the file format	Characters	2	"P6"
Whitespace	Characters	Variable	space, tab, CR, or LF
Width	Characters	Variable	Numbers in character form
Whitespace	Characters	Variable	space, tab, CR, or LF
Height	Characters	Variable	Numbers in character form
Whitespace	Characters	Variable	space, tab, CR, or LF
Max color value	Characters	Variable	Numbers in character form
A single Whitespace	Characters	Variable	space, tab, CR, or LF (usually LF)

Now that you know the header format, you can write a function that can read the header of a PPM image. The prototype for the function should be as follows:

```
void PPM_read_header(std::ifstream &inp, PPM_header &ppm_header);
```

The PPM structure should be as follows.

```
struct PPM_header {
    int width;
    int height;
    int max_color;
};
```

Note that the file should be opened for reading and passed to `PPM_read_header()` as a `ifstream` object. To make reading the header simple, use the input stream operator (operator>>) to read the Magic Number, Width, Height, and Max color value. Make sure that you check the Magic Number and throw a `std::runtime_error` if the signature does not match. You should read the last single whitespace character of the header using `ifstream::read()` instead of the input stream operator, as the stream operator will skip over the whitespace and soak up the next character.

Once you have written the function, write a `main()` driver that calls the function on the **ttu.ppm** image and prints out the images width, height, and max color value. For example, consider the following invocation:

This effort is partially funded by the National Science Foundation award: OAC 1549812

Reading the image

Next, you will write a function that reads the image data. For the purposes of this lab, you will only be responsible for reading files that have a Max color value of 255. In other words, an image has one byte per RGB component. You should create a structure that represent a pixel. The structure should be as follows:

```
struct RGB_8 {
    uint8_t r;
    uint8_t g;
    uint8_t b;
} __attribute__((packed));
```

Note that the attribute added to the end of the structure is to ensure that the compiler does not pad the structure. In other words, the red, green, and blue values must be formatted exactly as they appear in the structure.

The prototype for your image reading function is as follows:

```
void PPM_read_rgb_8(std::ifstream &inp, int width, int height, RGB_8 *img);
```

This function should be called immediately after reading the header. Therefore, the file pointer of the `ifstream` object will be in the correct position to begin reading the image data. Note that the width and height parameters are obtained from the header that was read by the `PPM_read_header()` function.

This function is actually very easy to write. The body of the function should call the `inp.read()` function, passing the `img` array and the number of bytes to read. How do you know how many bytes to read? The number of bytes is the size of an `RGB_8` times the width times the height of the image.

Next, modify your `main()` to read the image. Before you call your `PPM_read_header()` function, you will need to allocate an array to hold the image. To do so, declare a pointer to an RGB_8, and use the new operator to allocate it, like so:

```
RGB_8 *image = new  RGB_8[header.width*header.height];
```

Writing the image

Writing the image is as simple as reading the image. Create a function that has the following prototype:

```
void PPM_write_header_8(std::ofstream &outp, int width, int height);
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

Complete the function so that it writes the image header to a file. Note that the format must match the format described in the above **Reading the Header** section. You will only need the output stream operator (operator>>) to write the header data.

Finally, you can write the function that saves the image data. The prototype for that function is as follows:

```
void PPM_write_rgb_8(std::ofstream &outp, int width, int height, RGB_8 *img);
```

Note that this function is as easy as writing the function that reads the image data. However, it should call `outp.write()`. You should be able to determine what parameters to pass to `outp.write()`.

To test your functions, add code to your main that simply copies an image file, as in the following example run:

[redacted]

Converting to Grayscale and flipping

Finally, you get to do something interesting with the image. You will write a function to convert the image into grayscale. Converting an image into grayscale is simple. You will use the luminosity method, which typically gives good results for most situations. The luminosity method, as mentioned in the **Methodology** section above, re-calculates the red, green, and blue values according to the following formula:

*grayval = 0.21 * red + 0.72 * green + 0.07 * blue*

The color called *grayval* is repeated as the red, green, and blue component for that pixel in the image. Therefore, the algorithm (not c code!) is as follows.

```
void to_grayscale(RGB_8 *img, int width, int height) {
    for each rgb color value in img (denote as img[i])
      set temp to 0.21 * img[i].r + 0.72 * img[i].g
                + 0.07 * img[i].b
      set img[i].r to temp
      set img[i].g to temp
      set img[i].b to temp
    end for
}
```

Show that your program works by modifying your main so that it calls `to_grayscale()` before it calls `PPM_write_header_8()` and `PPM_write_rgb_8()`. Make sure that your code works by

This effort is partially funded by the National Science Foundation award: OAC 1549812

viewing the image (it should be, of course, grayscale).

Next, write a function that will flip the image as if it were being viewed in a mirror. The flip function has the following prototype.

```
void flip(RGB_8 *img, int width, int height);
```

The algorithm is straightforward. You will swap the pixel in the *i*th position of the row with the pixel at the (*width* - *i* - 1) position in the row. So, you will swap the *i*th pixel in the row with the *width*-1 pixel, the *i*th+1 pixel with the *width*-2 pixel, and so on. The algorithm should look like so:

```
void flip( RGB_8 *img, int width, int height) {
    for each row in img
      for each rgb color value in row (denote at row[i])
        swap row[i] with row[width-i-1]
      end for
    end for
}
```

Show that your program works by modifying your main so that it calls `flip()` after it calls `to_grayscale()`, but before it calls `PPM_write_header_8()` and `PPM_write_rgb_8()`. Make sure that your code works by viewing the image (it should be, of course, gray-scale *and* flipped).

Grayscale and Flip in Parallel

Now for the coup de grace. Processing images can be expensive, especially for very large images or processing thousands, or even millions, of images. Fortunately, some image processing can be done in parallel. By using parallel code to modify parts of the image simultaneously, you can greatly speed up the image processing on machines that have more than one processor.

Fortunately, writing code to parallelize simple loops in OpenMP, such as the loops in your `to_grayscale()` and `flip()` functions is trivial. All you have to do is add the following pragma immediately before the `for` loop in `to_grayscale()` and the `for` loop in `flip()`.

```
#pragma omp parallel for
```

So, add the pragma, and compile it, adding the following option to your compiler's command line: `-fopenmp`

So, What's the Difference?

To see how your code with the added OpenMP pragma makes a difference, download the **ttu_tiled.ppm** file at the link provided above (it's a medium sized file). Next add the following code before you call your `grayscale()` function:

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
// Note that  the following code works for g++ on Linux, Mac OS, and Windows
(using MinGW)
struct timeval tv1;
struct timeval tv2;
gettimeofday(&tv1, NULL);
```

Next, add the following code immediately after your grayscale function:

```
gettimeofday(&tv2, NULL);
std::cout << "Total time: " << (double) (tv2.tv_usec - tv1.tv_usec) /
1000000 +
  (double) (tv2.tv_sec - tv1.tv_sec) << std::endl;
```

Now, run the program on the **ttu_tile.ppm** image and record the time.   Next, comment the OpenMP pragma lines so that the code does not run in parallel. Re-run the program. If the parallel version does not run faster, try it with larger image sizes.

2.4.3 Matrix Multiply (CS1) - Parallel Matrix Multiplication Using Java

Skills You Will Learn

- Multiplying matrices
- Partitioning matrices
- Using Java's ForkJoin methodology

Prolog and Review

For this example, you should review your CS2 material on two dimensional matrices. Matrices in Java are represented as two dimensional arrays. A two dimensional array is just an array of arrays. Consider Figure 7. The Matrix is a 2 dimensional, or 2D, array consisting of multiple rows. Each row is itself an array. Mathematically, you would typically represent a particular row and column in an array with notation such as $m_{1,3}$ to indicate the value at row 1 and column 3. However, in Java, you use the bracket operator. In other words, $m_{1,3}$ is denoted as $m[1][3]$.

This effort is partially funded by the National Science Foundation award: OAC 1549812

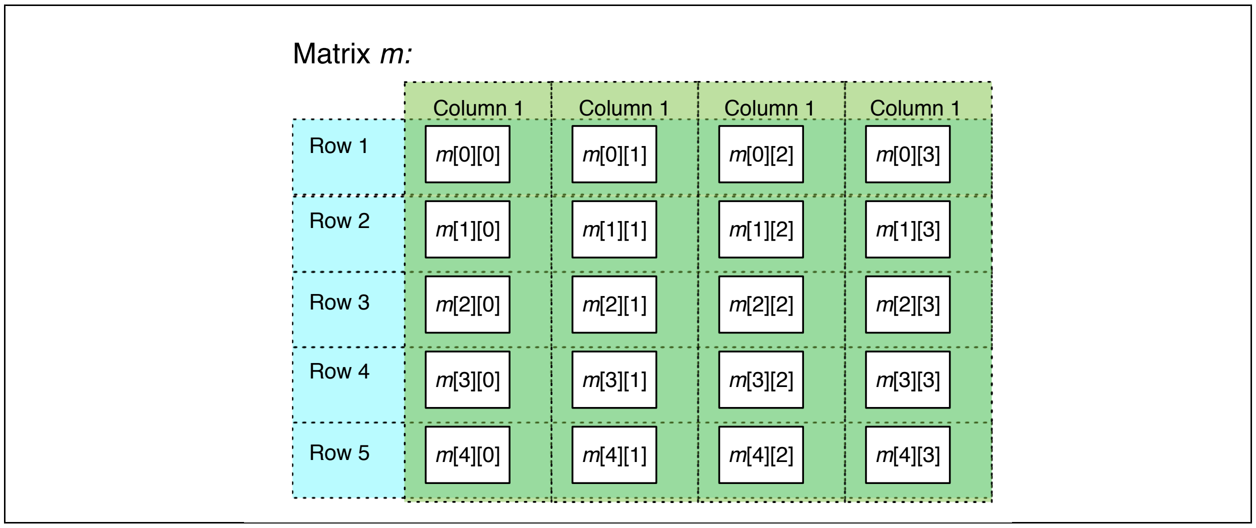


Figure 7: A matrix represented as a 2D array.

Most manipulation of 2D arrays require iterating over the values in the array. For example, consider squaring all the values in a 2D array. The following code shows how the algorithm might be implemented.

```
import java.lang.*;
import java.util.Random;

public class Mat_2d {
    public static final int NUM_ROWS = 5;
    public static final int NUM_COLS = 5;
    public static final double SCALAR = 2.5;

    public static void main(String args[]) {
        double m[][] = new double[NUM_ROWS][NUM_COLS];

        fill_array(m);

        scalar_multiply(m, SCALAR);

        print_array(m);
    }

    public static void fill_array(double m[][]) {
        Random r = new Random();
        for (int i = 0; i < m.length; i++) {
            for (int j = 0; j < m[0].length; j++) {
                m[i][j] = r.nextDouble();
            }
        }
    }

    public static void scalar_multiply(double m[][], double scalar) {
        for (int i = 0; i < m.length; i++) {
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
            for (int j = 0; j < m[0].length; j++) {
                m[i][j] *= SCALAR;
            }
        }
    }

    public static void print_array(double m[][]) {
        for (int i = 0; i < m.length; i++) {
            for (int j = 0; j < m[0].length; j++) {
                System.out.printf("%7.2f", m[i][j]);
            }
            System.out.println();

        }
    }
}
```

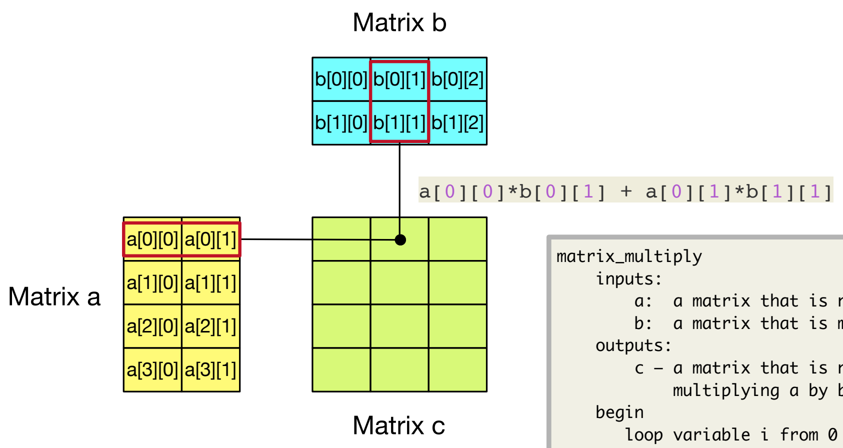
In each of the functions `fill_array()`, `scalar_multiply()`, and `print_array()`, a nested `for` loop iterates across the array. The outer loop iterates across the rows and the inner loop iterates across the columns of that particular row. Note how each function uses `m.length` to determine the number of rows in the matrix, and `m[0].length` to determine the number of columns.

Problem Description

For this example, you will be implementing a special case for multiplying two matrices together. However, before tackling the special case, understanding the generals case is necessary. The general case matrix multiply computation is a bit more difficult that just multiplying a matrix by a scalar. As shown in Figure 8, two matrices can only be multiplied together if the first matrix, denoted as matrix *a*, has the same number of columns as the number of rows in the second matrix *b*. A helpful way to visualize the multiplication is to consider an element of the matrix *c*, that is the result matrix, to be the results of multiplying two vectors together. One of the vectors is a row in matrix *a* and the other vector is a column in matrix *b*. The scalar result of multiplying two vectors together is the sum of the product of the first elements in each of the vectors and the product of the second elements in each of the vector, and so on. For example, given two vectors *x* and *y* each having three elements, the product of *x* and *y* is $x_0*y_0+x_1*y_1+x_2*y_2$.

The general algorithm for multiplying matrices together requires a triply nested loop. The outer loop iterates across the columns in *a*. The first nested loop iterates across the rows in *b*. The inner-most loop that accomplishes the vector multiplication of the row in *a* with the column in *b* by multiplying each element in *a* with the corresponding element in *b* and accumulating the results.

This effort is partially funded by the National Science Foundation award: OAC 1549812



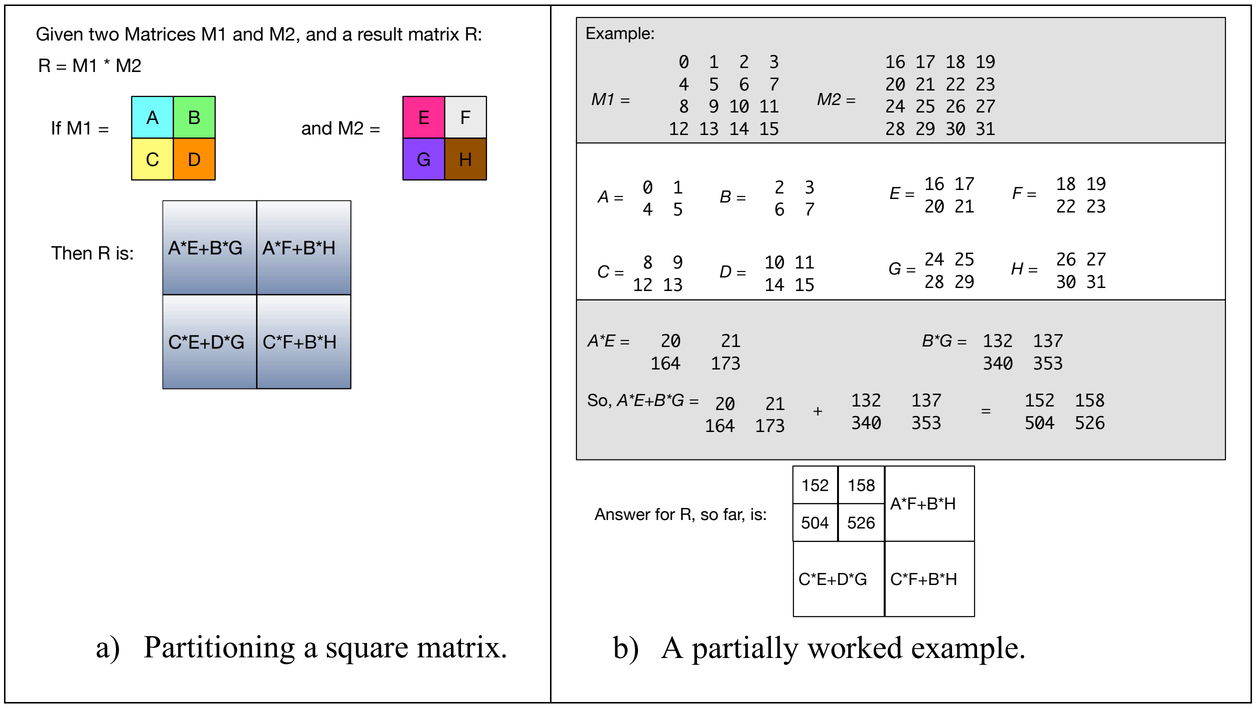
```
matrix_multiply
    inputs:
        a:  a matrix that is nxm
        b:  a matrix that is mxp
    outputs:
        c - a matrix that is nxp and is the result of
            multiplying a by b
    begin
       loop variable i from 0 to n-1
         loop variable j from 0 to p-1
           set $c_{i,j}$ to 0
           loop k from 0 to m-1
             set $c_{i,j}$ to $c_{i,j} + a_{i,k} * b_{k,j}$

           end loop

         end loop
       end loop
    end
```

Figure 8 Multiplying matrix a by martix b.

Methodology

Figure 9 shows a special case for multiplying matrices. In this case, the matrices are two square matrices. In other words, the matrices have identical dimensions and the number of elements in each matrix is a power of two. It is possible to multiply these two matrices using a recursive algorithm that has the same order notation as the general algorithm, O(n^3). Both matrices are subdivided into eight submatrices with dimensions $n/2$. We denote the four submatrices derived from the matrix on the left of the multiplication as A, B, C, and D, and the four submatrices derived by the matrix on the right as E, F, G, and H. The matrix obtained by multiplying the two original matrices together can then also be obtained as A*E + B*G (upper left submatrix), A*F + B*H (upper right), C*E + D*G (lower left), and C*F + B*H (lower right). Thus, we have eight new matrix multiplications to compute, which we obtain recursively. The base case multiplies two 1x1 matrices together.

This effort is partially funded by the National Science Foundation award: OAC 1549812



a) Partitioning a square matrix.

b) A partially worked example.

Figure 9: First step of recursive Matrix Multiply

How can this algorithm be implemented in parallel? Notice that each quadrant in R could be computed concurrently. In a recursive solution, the quadrants will be partitioned further, recursively, and all of those small quadrants can be computed in parallel.

Implementation

Tools

You will need to use the following tools to complete this example:

- An editor.
- The Java 8 SDK
- The following matrix multiply code which you will complete: MatMul.zip[4]

To implement a parallel matrix multiply using Java **ForkJoin** framework, you should first start with a serial recursive version. You will find a serial recursive implementation in the **MatMul.zip** file mentioned in the **Tools** section above.

Recursive Parallel Matrix Multiplication

With Java 8, it is straightforward to convert the serial algorithm into one that automatically uses multiple processors if available. However, we must identify the part of the algorithm that can safely be run in parallel. For parallel matrix multiplication, that the eight submatrix

4 http://www.csc.tntech.edu/pdcincs/resources/CS1/matrix_multiply/Java/complete_me.zip

This effort is partially funded by the National Science Foundation award: OAC 1549812

multiplications are performed does not matter. That is, it does not matter if B*G completes before A*E, as long as they both complete before the two matrices are added together to complete the upper left submatrix. You will now parallelize the above matrix multiplication code by including fork and join at the appropriate spots in the algorithm.

RecursiveTask

First, extend either RecursiveTask<T> or RecursiveAction. This decision is based on whether the recursive procedure returns a value or not. Our recursive procedure returns the result of a matrix multiplication, so we use RecursiveTask<Matrix>, as shown in the code below.

```java
package matrix;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class MatrixMultParallel extends RecursiveTask<Matrix>
{
      private Matrix mat_1;
      private Matrix mat_2;
      private int thread_id;

      public MatrixMultParallel(Matrix mat_1, Matrix mat_2, int t_id)
      {
            this.mat_1 = mat_1;
            this.mat_2 = mat_2;
            thread_id = t_id;
      }

      //...  the rest of the code, see below
}
```

Overriding compute()

Next, override the `compute()` method. This is the method that is automatically called when a new thread for a task is forked. This can be a bit tricky. You don't really want to start new threads if the size of the subtask is small, because forking and joining takes time. If you do too much forking and joining, the serial version can actually run faster! In this case, you only want to spawn eight threads that correspond to the eight matrix multiplications that occur at the first level of recursion. The problem size decreases rapidly, and subsequent recursive levels will not have nearly as much work to do. As shown in the code below, which thread forks is determined by the thread's id. Only the top level thread will call the method that creates new threads.

```java
protected Matrix compute()
{
      if (thread_id == 0)
      {
            return multiplyRec();
      }
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
        else
        {
                Matrix result = MatrixMultSerial.multiplyRec(mat_1, mat_2);
                return result;
        }
}
```

Fork and Join
To fork a thread, simply create a new instance of the class and call fork. The code in the listing below creates new threads for the eight smaller matrix multiplications. Afterwards, the code joins all of these threads, and each join returns the result of the thread because the code is using RecursiveTask<T>. *Forking should be completed before joining*. If you join before all subtasks have been forked, then you are not taking advantage of parallelism. You need to join all of the results in order to obtain the complete matrix multiplication result, but only after all subtasks have been forked.

```
private Matrix multiplyRec()
{

    int mid_row = mat_1.getNumRows()/2;
    int mid_col = mat_1.getNumCols()/2;

    Matrix A = fillHalfMatrix(mat_1, 0, 0);
    Matrix B = fillHalfMatrix(mat_1, 0, mid_col);
    Matrix C = fillHalfMatrix(mat_1, mid_row, 0);
    Matrix D = fillHalfMatrix(mat_1, mid_row, mid_col);

    Matrix E = fillHalfMatrix(mat_2, 0, 0);
    Matrix F = fillHalfMatrix(mat_2, 0, mid_col);
    Matrix G = fillHalfMatrix(mat_2, mid_row, 0);
    Matrix H = fillHalfMatrix(mat_2, mid_row, mid_col);

    //multiplyRec(A, E);
    MatrixMultParallel one = new MatrixMultParallel(A, E, 1);
    one.fork();

    //multiplyRec(A, E);
    MatrixMultParallel two = new MatrixMultParallel(B, G, 2);
    two.fork();

    //multiplyRec(A, E);
    MatrixMultParallel three = new MatrixMultParallel(A, F, 3);
    three.fork();

    //multiplyRec(A, E);
    MatrixMultParallel four = new MatrixMultParallel(B, H, 4);
    four.fork();

    //multiplyRec(A, E);
    MatrixMultParallel five = new MatrixMultParallel(C, E, 5);
    five.fork();
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
    //multiplyRec(A, E);
    MatrixMultParallel six = new MatrixMultParallel(D, G, 6);
    six.fork();

    //multiplyRec(A, E);
    MatrixMultParallel seven = new MatrixMultParallel(C, F, 7);
    seven.fork();

    //multiplyRec(A, E);
    MatrixMultParallel eight = new MatrixMultParallel(D, H, 8);
    eight.fork();

    Matrix AE = one.join();
    Matrix BG = two.join();
    Matrix AF = three.join();
    Matrix BH = four.join();
    Matrix CE = five.join();
    Matrix DG = six.join();
    Matrix CF = seven.join();
    Matrix DH = eight.join();

    Matrix m1 = AE.add(BG);
    Matrix m2 = AF.add(BH);
    Matrix m3 = CE.add(DG);
    Matrix m4 = CF.add(DH);

    Matrix result = fillFullMatrix(m1, m2, m3, m4);

    return result;
}
```

ForkJoinPool

Lastly, you need a ForkJoinPool object so that the forked threads can effectively utilize the available processors. You do not need to create a new object to manage the pool. You can simply use `ForkJoinPool.commonPool()` to obtain the required object. You can do this in MatrixMultDriver class where you will run both the serial and parallel versions, time them, and compare the results. As shown in the code below, simply create a top level instance of the class with the left matrix and the right matrix that are to be multiplied together. Then use the ForkJoinPool object to start up the top level thread with thread id 0. This method call will also return the final summation because you are using RecursiveTask<T>.

```
MatrixMultParallel parallel = new MatrixMultParallel(mat_1, mat_2, 0);
System.out.println("starting parallel: ");

long start_time = System.nanoTime();

Matrix mat_3 = ForkJoinPool.commonPool().invoke(parallel);

long stop_time = System.nanoTime();
long time_diff_millis = (stop_time - start_time)/1.E6;
System.out.println(time_diff_millis);
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

So, What's the Difference?
See if your parallel version is faster than your serial version. If the parallel version does not run faster, try it with larger matrix sizes.

2.4.4 Image Processing (CS2) - Gray-scaling and Flipping an Image Using C++ and OpenMP

Skills you will learn

- Applying multiple image filters
- Functional decomposition via pipelining
- Parallel Processing using OpenMP

Prolog and Review
For this lab, you should review your CS1 material on arrays, structures, and arrays of structures.

You should also review your CS2 course material on queues. The following is a short review. The queue is a container. In other words, it is used to create items of data much like an array. However, it exports an interface that is different from the array for manipulating the contained items. If you want to add items to the queue, you must append the items to the end of the queue. If you want to get the value of an item in the queue, you must get the value of the item at the beginning of the queue.

For example, consider the C++ Standard Template Library, or STL, queue. You must include the following line in your C++ code to use the STL queue.

```
#include <queue>
```

Then you can allocate a queue and used the queue's `push()` method to append an item to the end of the queue, as in the following example.

```
#include <iostream>
#include <queue>

int main() {
  std::cout << "Appending 10 and 20 to the queue..." << std::endl;
  std::queue<int> q;

  q.push(10);
  q.push(20);

  return 0;
}
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

If you want to dequeue an item from the queue to get its value, you must use two methods of the STL queue: `front()` and `pop()`. The STL queue also includes methods for determining if a queue is empty, determining the queue's size, and other helpful methods. Following is a more complete example.

```cpp
#include <iostream>
#include <queue>

int main () {
  std::queue<int> q;
  int number = 0;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  std::cin >> number;
  while (number != 0) {
    q.push (number);
    std::cin >> number;
  }

  std::cout << "The queue is " << (q.empty() ? "Empty" : "Not Empty")
            << std::endl;
  std::cout << "The size of the queue is " << q.size() << std::endl;
  std::cout << "The queue contains: ";
  while (!q.empty()) {
    std::cout << ' ' << q.front();
    q.pop();
  }
  std::cout << std::endl;

  return 0;
}
```

Following is what the code prints when it is executed.

```
$ ./queue
Please enter some integers (enter 0 to end):
1 2 3 4 5 0
The queue is Not Empty
The size of the queue is 5
The queue contains:  1 2 3 4 5
```

Problem Description

For this example, you will be writing programs to manipulate images. The two image processing techniques that you will implement are grey-scaling and flipping an image. *Figure* 5 from Example 2.4.2 shows images that has been gray-scaled, flipped, and then both gray-scaled and flipped. You should review that section's description of how images are represented as RGB

This effort is partially funded by the National Science Foundation award: OAC 1549812

triples, and how images are gray-scaled and flipped.

Methodology

You will be implementing an image processing technique called *Filtering*. Filtering is simply applying a sequence of operations to an image to achieve a desired outcome. For example, if you wanted to show an image as if it were depicted as shown in a mirror on an old black-and-white TV, you would apply the gray-scale and flip filters. Fortunately, for many image filters, such as gray-scale and flipping, you can accomplish the filtering in parallel and thus reduce the computation time. One method is *pipelining*, which is a form of *functional decomposition*. Functional decomposition decomposes a problem according to the functions that are used to accomplish the computations. Functional decomposition is in contrast to *domain decomposition* that decomposes a problem according to the input data.

As shown in Figure 10, Pipelining is a form of functional decomposition in which a series of filters, or functions, manipulates a portion of the input data, and then passes the data onto the next function, which begins processing the data. However, the first function simultaneously begins computing of the next portion of input data, so that both functions are computing in parallel. Note that the technique can be applied to arbitrarily many functions.

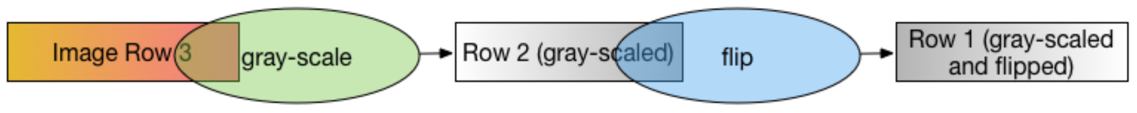


Figure *10*: Using a pipeline to gray-scale and flip and image.

Notice that in Figure 2, the rectangles are overlapping with the ovals to represent that the gray-scale function is simultaneously processing image row 3 while flip is processing row 2, which was previously processed by gray-scale.

So, how do you implement a pipeline? Consider Figure 11. A straightforward way to implement pipelining is to use a queue to pass data between the functions. When the first function, which is gray-scale in this example, finishes a row, it passes that row to flip by putting the row in a shared queue. When the flip function is ready to process the next row, it checks the queue. If the queue has a row in it, then the flip function removes the row and processes it.

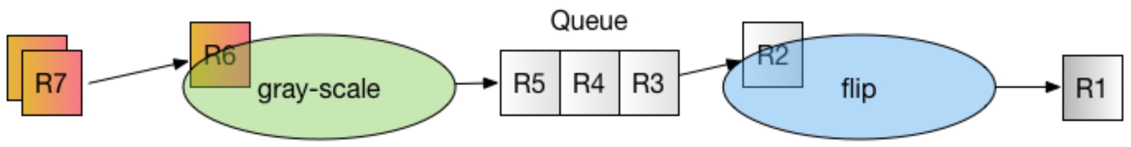


Figure *11*: Implementing a pipeline with a queue.

Implementation

For this example, you will be implementing filters to gray-scale and flip an image. You will then

This effort is partially funded by the National Science Foundation award: OAC 1549812

modify your code so that it processes the filters in parallel via a pipeline, thus speeding up the applications of the functions.

Tools

You will need to use the following tools to complete your assignment:

- An editor.
- A C++ compiler that is OpenMP capable (such as the gnu C++ compiler).
- Some image files with which to work. The following image files are available online: ttu.ppm[5] ttu_tile.ppm[6]
- A program that can display PPM image files (for example, a browser).
- The following library that contains code to read and write PPM images: libppm.cpp[7] and libppm.h[8]

Preliminaries: Reading and Writing a PPM File

To get started, download the **libppm.cpp** and **libppm.h** files above, as well as the **ttu.ppm** and **ttu_tile.ppm** image files. Next, you will make sure that you can compile and run a simple program that uses the functions in libppm.cpp to read and write PPM files. Create a new file in your editor called **ppm_lab.cpp**. Put the following code in ppm_lab.cpp:

5 http://www.csc.tntech.edu/~mrogers/2100/ttu.ppm
6 http://www.csc.tntech.edu/~mrogers/2100/ttu_tile.ppm
7 http://www.csc.tntech.edu/~mrogers/2100/libppm.cpp
8 http://www.csc.tntech.edu/~mrogers/2100/libppm.h

This effort is partially funded by the National Science Foundation award: OAC 1549812

```cpp
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <sstream>
#include <sys/time.h>

#include "libppm.h"

int main(int argc, char *argv[]) {

  if (argc != 3) {
    std::cerr << "Usage: " << argv[0] << " in_ppm_file out_ppm_file"
              << std::endl;
    return 1;
  }

  PPM_header img_header;

  try {

    std::ifstream ifs(argv[1], std::ios::binary);
    if (!ifs) {
      throw std::runtime_error("Cannot open input file");
    }
    PPM_read_header(ifs, img_header);

    std::cout << img_header << std::endl;

    RGB_8 *img = new RGB_8[img_header.height * img_header.width];
    PPM_read_rgb_8(ifs, img_header.width, img_header.height,
        (RGB_8 *) img);

    std::ofstream ofs(argv[2], std::ios::binary);
    if (!ofs) {
      throw std::runtime_error("Cannot open output file");
    }
    PPM_write_header_8(ofs, img_header.width, img_header.height);
    PPM_write_rgb_8(ofs, img_header.width, img_header.height,
                    (RGB_8 *) img);

    ifs.close();
    ofs.close();

  } catch (std::runtime_error &re) {
    std::cout << re.what() << std::endl;
    return 2;
  }
  return 0;
}
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

Note that the above code simply reads the image file given as the first parameter on the command line, and then saves the image to the file given as the second parameter on the command line.

Gray-scaling and Flipping

Review the material **Converting to Grayscale and flipping** on page 27 that describes how to grey-scale and flip images. Implement the serial version for gray-scaling and flipping as described.

Pipelining

Now, you are going to modify your code so that the image filters will be applied in a pipeline. Note that once a row of pixels is finished being gray-scaled, the row can be immediately flipped. Time can be saved by flipping an already gray-scaled row, and, at the same time, gray-scaling the next row in the image. Unfortunately, you have to modify your code to take advantage of this parallelism. Fortunately, OpenMP an help.

You will modify your code so that the `grayscale()` function enqueues the row that it just finished gray-scaling. Then, you will modify your `flip()` function by removing the outer `for` loop that loops through the rows and replacing it with a call that dequeues the next row to flip. Using a queue in this way implements a pipeline between the two functions.

So, now for the details. First, declare your queue at the top of your C++ file. The declaration should look like so:

```
std::queue<RGB_8 *> pipeline;
```

Add the following code to your C++ file. This code accomplishes two goals. First, the queue is a shared resource between the `grayscale()` function and the `flip()` function, so it must be protected from concurrent access (Note the OpenMP `critical` pragma). Second, the `flip()` function can only flip a row when a row is available. Thus the `dequeue()` function must check the availability of a row in a loop.

```
RGB_8 *dequeue(std::queue<RGB_8 *> &q) {
   RGB_8 *image_row;
  bool keep_checking = true;
  while (keep_checking) {
#pragma omp critical (pipeline)
    {
      if (!pipeline.empty()) {
        image_row = pipeline.front();
        pipeline.pop();
        keep_checking = false;
      }
    }
#pragma omp taskyield
  }
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
  return image_row;
}

void enqueue(std::queue<RGB_8 *> &q, RGB_8 *row) {
#pragma omp critical (pipeline)
  {
    pipeline.push(row);
  }
}
```

Next, modify your gray-scale code such that, at the end of processing each row, that row is enqueued. So the algorithm changes to the following:

```
void to_grayscale(RGB_8 *img, int width, int height) {
    for each row in img
      for each rgb color value in row (denote as row[i])
        set temp to 0.21 * row[i].r + 0.72 * row[i].g
                    + 0.07 * row[i].b
        set row[i].r to temp
        set row[i].g to temp
        set row[i].b to temp
      end for
      enqueue(row)
    end for
    enqueue(0)
}
```

Note the `enqueue(0)` at the very end. The sentinel value of 0 is used to signal the `flip()` function that no more rows are available.

Next, modify the `flip()` function. Your flip function should not use an outer `for` loop to determine the next row. Instead, it should get the next row from the queue. Following is the pseudocode:

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
void flip(int width) {
    do
      set row to dequeue(pipleline)
      if row is not 0
        for each rgb color value in row (denote as row[i])
          swap row[i] with row[width-i-1]
        end for
      end if
    while row is not 0
}
```

Finally, modify the main so that OpenMP will spawn the functions in two different threads. So, replace your calls to `to_grayscale()` and `flip()` in the main with the following code:

```
#pragma omp parallel num_threads(2)
  {
    if (omp_get_thread_num() == 0) {
      to_grayscale(image, width, height);         } else {
      flip(width);                   }     }
```

Note that the above code runs `to_grayscale()` in the thread with id 0, and the `flip()` function in the other thread.

Compile your program. Add the `-fopenmp` option to your compiler command (for example: `g++ -fopenmp -o ppm_lab ppm_lab.cpp libppm.cpp`). Test it to make sure that it works.

Now, run both the serial version and the parallel version of your on the ttu_tile.ppm image and compare the timings. Try your code on a larger image if the parallel version is not faster than the parallel version.

2.4.5 Parallel Sort (CS2) - Parallel Number Sorting Using C++ and OpenMP

Skills you will learn

- Sort numbers in parallel
- Parallel Processing using OpenMP

This effort is partially funded by the National Science Foundation award: OAC 1549812

Prolog and Review

You should review your course CS2 instruction on sorting arrays. Most of the common sorting algorithms are implemented with nested loops. For example, selection sort, bubble sort, and insertion sort are all implemented with doubly nested loops. In computer science, such algorithms are termed as, asymptotically, *n*-squared algorithms. An *n*-squared algorithm that has *n* input items takes approximately *n* times *n* steps to sort the items, in the worst case. Such algorithms can take a very long time to sort large data sets. So, how can you reduce the sorting time? You can reduce the sorting time with parallelism, of course!

Problem Description

For this example, you will be writing program to sort numbers stored in an array. You will use the *bubble sort* algorithm. The bubble sort algorithm is a pretty simple sort. Consider the Figure 12. The table shows how the bubble sort algorithm make multiple passes over the array. At each pass, iterates over the array and examines pairs of numbers. The numbers show in white are numbers that did not have to be swapped because the numbers are already in the correct order relative to each other. For example, in the first pass and the first iteration, the algorithm compares the numbers 5.0 and 10.3. The algorithm does not swap the number because 5.0 is less than 10.3. However, in the second iteration, the algorithm swaps 10.3 and 8.7 because those number are out of order. Notice that the algorithm stops when no more swaps are needed to properly order any pair of numbers.

	array:	5.0	10.3	8.7	5.0	68.2
Pass	Iteration					
1	1	5.0	10.3	8.7	5.0	68.2
1	2	5.0	8.7	10.3	5.0	68.2
1	3	5.0	8.7	5.0	10.3	68.2
1	4	5.0	8.7	5.0	10.3	68.2
2	1	5.0	8.7	5.0	10.3	68.2
2	2	5.0	5.0	8.7	10.3	68.2
2	3	5.0	5.0	8.7	10.3	68.2
2	4	5.0	5.0	8.7	10.3	68.2

Figure *12*: Sorting integers in an array.

This effort is partially funded by the National Science Foundation award: OAC 1549812

Methodology
You will use domain decomposition, also sometimes called data decomposition, to bubble sort an array of numbers in parallel. Domain decomposition requires dividing the array into equal parts and assigning each part to a processor. Consider the simple example in Figure 13. The figure depicts threads running on three processors, including a thread designated as the master thread. The computation occurs in two phases. First, the work is divided equally among the three threads. In this case, each thread sorts four numbers in the array. The master thread sorts all elements starting at index 0 and ending at index 3, the second thread sorts all elements starting at index 4 and ending at index 7, and the third thread sorts all elements starting at index 8 and ending at index 11.

The second phase occurs after all of the threads are finished with the first phase. Note that sorting the pieces of the array does not result in a completely sorted array. In the second phase, the master must merge sorted pieces to produce a completely sorted result. However, the second phase must be done serially by a single thread.

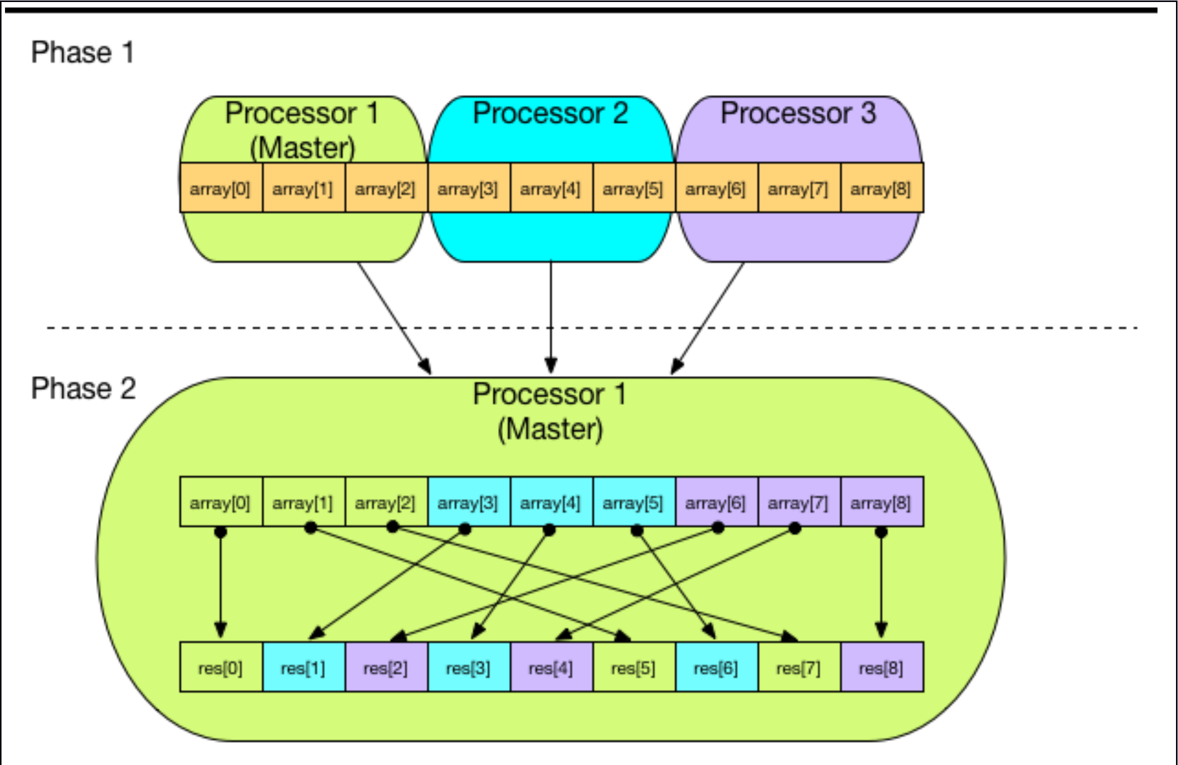


Figure *13*: Three processors sorting in parallel.

Implementation
For this example, you will be implementing the code to sort numbers in serial and then in parallel.

Tools
You will need to use the following tools to complete your assignment:

This effort is partially funded by the National Science Foundation award: OAC 1549812

- An editor.
- A C++ compiler that is OpenMP capable (such as the gnu C++ compiler).

Generating Lots of Random Floating Point Numbers

You need lots of numbers to store in the array so that you can sort them. Normally, you would have some important data, perhaps stored in a file or database, that you need to sort. However, for this lab, you will generate some data with which to work. Luckily, web sites exist that can generate the numbers for us. Click on the following link to generate a million numbers: here[9]. Save these numbers to a file, and then write a main driver that reads the file into a single dimension array of integers. Call your C++ program file **bsort_serial.cpp**.

Sorting the numbers

An example of bubble sorting an array was shown in the **Methodology** section. You should be able to implement the bubble sort algorithm in C++. The simplest algorithm is the following.

```
bsort()
  input:
     array     - an array of integers
     num_items - the number of items in the array
  returns:
     nothing
  begin
    loop i from 0 to num_items - 1
      loop j from num_items-1 down to i + 1
        if the array item at j-1 is greater that the array item at j
          swap the array item at j-1 with the array item at j
        end if
      end for
    end for
  end
```

The above algorithm is a very simple version of the bubble sort algorithm. See if you can implement a more efficient version.

After you write the bubble sort in C++, add code to your `main()` that calls the bubble sort algorithm on the array that after the code has read the numbers from the file. You should test your code to make sure it works, so you should call the bubble sort function on a small array and print the results.

Sorting in Parallel

You can you make the serial version faster in the previous step by sorting parts of the array in

9 https://www.random.org/sequences/?mi0n=1&max=10000000&col=1&format=plain&rnd=new

This effort is partially funded by the National Science Foundation award: OAC 1549812

parallel and then merging the results, as describe in section **Methodology**.

Copy the **bsort_serial.cpp** file to a file named **bsort_parallel.cpp**. Now, you will modify the new file to make a parallel version. The **Methodology** section above showed how the array should be decomposed for each processor in the system to compute a part of the array, and thus compute the parts in parallel. For this lab, you will decompose the array into two approximately equal halves, and then have each half sorted by its own thread. In other words, each thread will call the `bsort()` function, but will pass the beginning of its half along with the number of items in its half. Something similar to the following should replace the call to `bsort()` in your code:

```
1     int size1 = how_many/2;
2     int size2 = how_many-size1;
3     #pragma omp parallel num_threads(2)
4     {
5
6       if (omp_get_thread_num() == 0) {
7         bsort(numbers, size1);
8       } else if (omp_get_thread_num() == 1) {
9         bsort(numbers + size1, size2);
10      }
11    }
```

In the code above, `how_many` is the total size of the `numbers` array (the array to be sorted). The code above stores the size of the first half of the array, i.e. the part that is to be sorted by the first thread, in the `size1` variable. Then, it computes the size of the rest of the array, which is the part that is to be sorted by the second thread, and stores that size into the `size2` variable. Note that `size2` may be one larger than size1. For example, in `how_many` is 11, then `size1` will be 5 and `size2` will be 6.

The pragma in the code above executes a *fork-join* with two threads at line 3. In other words, the program *forks* two threads, and each thread executes the code block beginning at line 4 and ending at line 11. Once both threads finish executing the code block, the threads *join* and only the main thread continues after line 11. What does the block of code do? Each thread, executes the `if` statement at line 6. The thread with the thread id 0 will call the `bsort()` function (at line 7) on the first half of the `numbers` array, and the thread with the thread id 1 will call the `bsort()` function (at line 9) on the second half of the `numbers` array.

Unfortunately, you are not finished. Each part has been sorted, but he entire array is not yet sorted. You must merge the two halve as explained in the **Methodology** section above. Following is a merge algorithm that combines two arrays into a finished result.

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
merge()
  input:
    a1    - first sorted array
    size1 - the size of the first array
    a2    - second sorted array
    size2 - the size of the second array
  output:
    results - the merged result
  begin
    set idx_a1 to 0
    set idx_a2 to 0
    set result_size to 0
    while idx_a1 <> size1 || idx_a2 <> size2
      // there are more items to merge
      while idx_a1 <> size1 // more items yet unmerged in a1 and ...
            and (idx_a2 = size2 // no items to be merged in a2 ...
            or $a1_{idx\_a1}$ <= $a2_{idx\_a2}$)
        set $results_{result\_size}$ equal to $a1_{idx\_a1}$
        add one to result_size
        add one to idx_a1
      end while
      while idx_a2 <> size2  // more items yet unmerged in a2 and ...
            and (idx_a1 = idx1 //no more item to be merged in a1 ...
            or $a2_{idx\_a2}$ <= $a1_{idx\_a1}$)
        set $results_{result\_size}$ to $a2_{idx\_a2}$
        add one to result_size
        add one to idx_a2
      end while
    end while
  end
```

Implement the merge function in your C++ program. Add a call to the `merge()` function after the parallel code block. The call should look like the following:

```
int results[how_many];
merge(numbers, size1, numbers + size1, size2, results);
```

Finally, test your parallel version of bubble sort to make sure it works.

So, What's the Difference?

To see how your code with the added OpenMP code makes a difference, you can add the following code before you call `bsort()` to your `main()` function in both your serial and parallel version (before the parallel block in **bsort_main_parallel.cpp**):

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
// Note that  the following code works for g++ on Linux, Mac OS, and Windows
(using MinGW)
struct timeval tv1;
struct timeval tv2;
gettimeofday(&tv1, NULL);
```

Next, add the following code immediately after your call the `bsort()` function in `main()` for both the serial and parallel version (after the parallel block in **bsort_main_parallel.cpp**):

```
gettimeofday(&tv2, NULL);
std::cout << "Total time: " << (double) (tv2.tv_usec - tv1.tv_usec) /
1000000 +
  (double) (tv2.tv_sec - tv1.tv_sec) << std::endl;
```

Now, run the serial and the parallel version and compare the timings. If the parallel version is not faster than the serial version, increase the size of the array until you see a speedup.

2.4.6 Radix Sort (CS2) - Radix Sort in Parallel (Java)

Skills You Will Learn

- Implementing the Radix Sort algorithm
- Parallelizing the Radix Sort algorithm

Prolog and Review

You are familiar with sorting algorithms that compare elements such as bubble sort, selection sort, and merge sort. The order notation for an efficient sorting algorithm that compares elements is O(*n*log*n*). There are sorting algorithms that do not directly compare elements to each other. One of these is radix sort with an order notation of O(*n*), although it can have a large coefficient, as you will see.

Problem Description

This section describes Radix Sort and then describes and alternative recursive algorithm that can be easily parallelized using Java's **ForkJoin** framework.

Radix Sort 1

Suppose you have numerous items stored in an array that you wish to sort using radix sort. The usual way that radix sort is implemented is to start by examining the last character of the sort key. For simplicity, let's assume that the sort keys contain only lower case letters and that we want to sort in ascending order. Based on the ASCII value of the last character of the sort key, the item with that sort key is placed on a particular queue. Thus, all of the items with the same last character in their sort key will be placed on the same queue. The items are then removed from the queues starting with the queue for 'a', then 'b', and so forth. Each queue is completely emptied before moving on to the next queue. The items are placed back on the array in the order that they were removed from the queues. Of course, the items are not at all sorted at this point.

This effort is partially funded by the National Science Foundation award: OAC 1549812

Next, the entire process outlined above is repeated for the second to last character, then the third to last, and so forth. The items will be sorted after the first character in the sort key has been processed. Thus, even though the algorithm is O(*n*), there is a coefficient equal to the number of characters in the sort key, and **all** of the characters will be examined, even if the first character of the sort keys are all different. Further, sort keys are rarely all the same length. If an attempt is made to access a sort key character that is out of range, one can return a special value that will place that sort key in the very first queue. If two sort keys are identical except for differing lengths, the shorter sort key will be placed earlier in the sorted array.

Radix Sort 2

It is possible to implement radix sort starting with the first character if recursion is employed, as shown in Figure 14. Create a single queue and place all of the items to be sorted on it. Call a procedure that accepts a queue and the index of the current character to be examined as parameters. The initial call to this procedure will set the current character to 1 for the first character in the sort key.

The procedure removes items from the incoming queue and places them on to a new set of initially empty queues by examining the sort key character specified by the current character parameter. For each of these new queues that contains more than one item (or if a user specified maximum character is reached), a recursive call is made that increases the current character by one. In this way, after recursion has completed on one of the new queues, the items in that queue will be sorted. After any necessary recursion has completed, the items are collected from the queues, in order, and placed back on the queue that was passed to the procedure.

This effort is partially funded by the National Science Foundation award: OAC 1549812

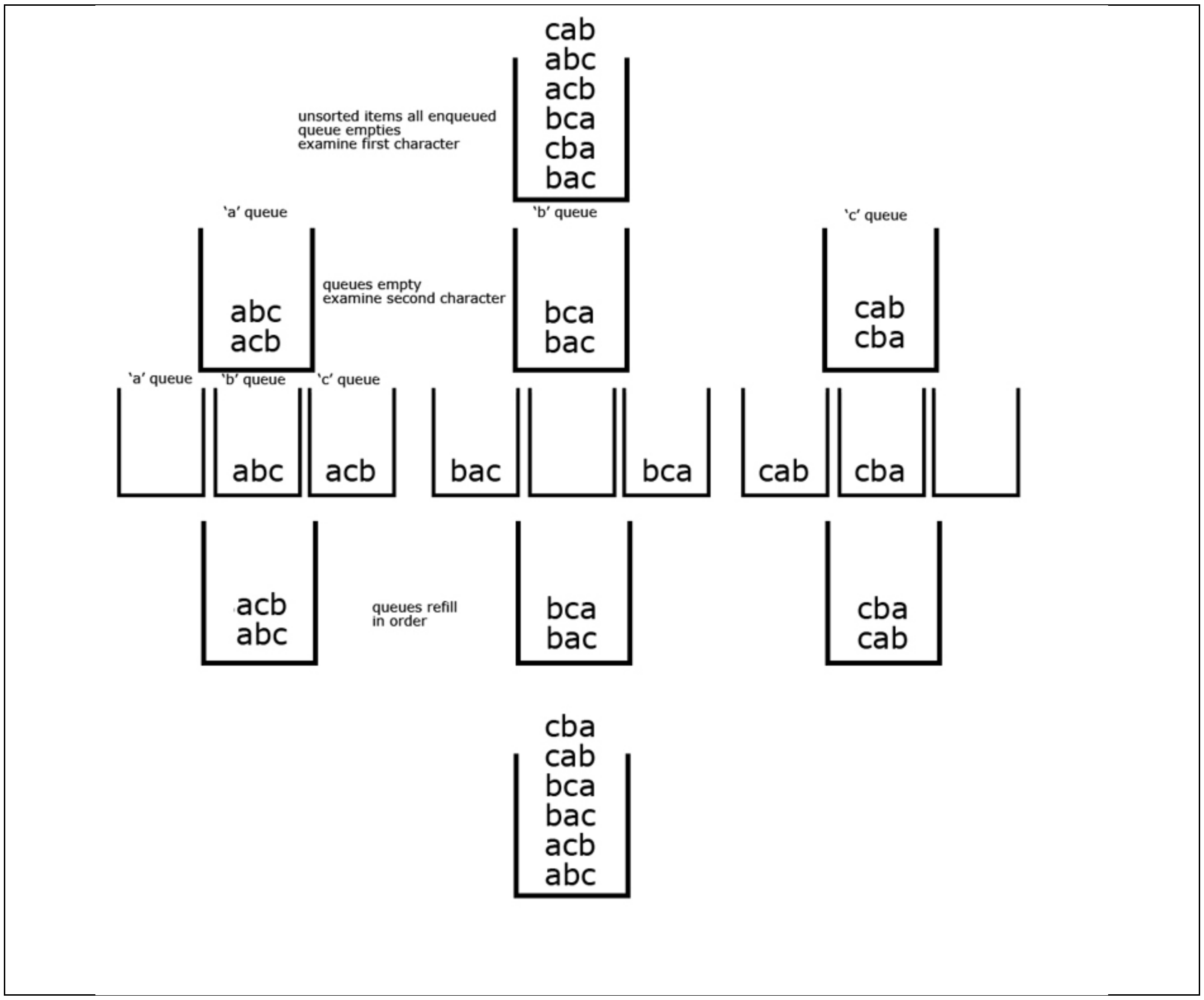


Figure *14*: Radix Sort

This version may appear to be inefficient due to the large number of queues created for each recursion call. However, because the branching factor is so large, very few recursive levels are actually required as most of the queues at the second or third recursive level will be empty or only contain one item (unless there are a lot of nearly duplicate keys). This version is appealing as it starts with the first character, and, in most cases, does not process that many characters in the sort key.

Methodology

Using a recursive fork-join algorithm, it is straightforward to convert the above recursive algorithm into one that automatically uses multiple processors if available. However, we must identify the part of the algorithm that can safely be run in parallel. For parallel radix sort, the order in which the set of queues are sorted does not matter. That is, the queue for 'z' can be sorted before the queue for 'a'. However, emptying the set of queues and placing their contents back on the incoming queue must be done in order. Thus, only after all of the queues have been forked should you start to join them. We can fork when making a recursive call on a queue, and we can join when ready to empty a queue.

This effort is partially funded by the National Science Foundation award: OAC 1549812

Implementation

<u>Tools</u>

- A Text editor
- Java SDK version 8
- The following Radix Sort code that you can complete: RadixSort.zip[10]

<u>Recursive Parallel Radix Sort</u>

Radixator

Radixator is an interface that simply returns the item's requested sort key character. It is implemented by the RadixCDTitles class which uses the title of the CD as the sort key. If another sort key is desired, it is easy to write a new class that implements Radixator in another way.

RecursiveAction

You must extend either RecursiveTask<T> or RecursiveAction. This decision is based on whether the recursive procedure returns a value or not. Our recursive procedure does not return a value so we will extend RecursiveAction.

compute()

You will next override the `compute()` method. This is the method that is automatically called when a new thread is forked. This can be a little tricky. You do not want to start new parallel tasks if the size of the subtask is small, because forking and joining can take quite some time. If you do too much of it, the serial version can actually run faster! In this case, you will only allow the very first set of queues, those that are sorting based on the first character, to run in parallel. It is likely that queues working on subsequent characters will not have many items to sort.

Our top level thread will have thread id 0. If you want to restrict forking and joining to the queues sorting by the first character, then the top level thread is the only thread that will call the version of `binSort()` that involves forking and joining each queue in the set. When other threads call `compute()`, they will call the RadixSortSerial's `binSort()` method which will not start any new threads. Thus, the compute method will "dispatch" to a sorting method that will create threads as available for each of the queues in the set if the thread calling compute has thread id 0.

Fork and Join

The binSort method that will spawn new threads for each of the queues in the set. At most, you need 37 threads for RadixSort (26 queues for lower case letters, 10 queues for digits, and 1 queue for spaces). Therefore, you can create an array of type RadixSortParallel with size 37. If a recursive call is required for a given bin (more than one item in the bin), create a new RadixSortParallel object, place it in the array of threads, and fork it. In a separate loop, join any

[10] http://www.csc.tntech.edu/pdcincs/resources/CS2/radix_sort/Java/code/complete_me.zip

This effort is partially funded by the National Science Foundation award: OAC 1549812

threads that are not null (remember, some queues did not need any recursion). Then proceed to empty the bins in order, placing the results back on the incoming queue.

ForkJoinPool

Lastly, you need a ForkJoinPool object so that our forked subtasks can effectively utilize the available processors. You do not need to create a new object to do this, You can simply call `ForkJoinPool.commonPool()` to obtain the required object.

You can make this call in the RadixSortParallel class (in the `radixSort()` method) where you run and time the parallel version of the algorithm. Simply create a top level instance of the class with parameters appropriate for sorting the entire array (and make sure to set the `thread_id` to 0). Then use the ForkJoinPool object to start up the top level thread. The ForkJoinPool's `invoke()` method will not return anything because we are using RecursiveAction.

The **RadixSort.zip** file that you can download from the supplied link has an incomplete implementation of the parallel Radix Sort. Sections of the code that you can complete to create a working version are in RadixSortParallel and are marked with **DO THIS**.

2.4.7 MiniMax (CS2) - TicTacToe MiniMax Search in Parallel Using Java

Skills You Will Learn

- Implementing MiniMax Search
- Parallelizing MiniMax search using the Java **ForkJoin** framework
- Implementing a TicTacToe game using MiniMax search

Prolog and Review

Review the material in your CS2 course on the tree data structure.

Binary Tree/General Tree

Consider a general binary tree (an unorganized binary tree, not a binary search tree) as shown in Figure 15. You are familiar with the post-order traversal of a general binary tree where both of the children (if the node has both a left and a right child) of a particular node are visited before the node itself is visited. Suppose you wanted to find the largest item stored in the tree. For a given subtree in the binary tree, you make a recursive call on the left child to get the largest item in the left subtree and a recursive call on the right child to get the largest item in the right subtree. The larger of these two items is compared to the item stored in the parent of the subtree to find the overall largest item in the subtree. The largest item from that subtree is then returned to its parent for similar tests, and so forth until the largest item is returned from the root node.

Now consider that each node can have any number of children, not just two or fewer as in a binary tree. This is a general tree, but finding the largest item in this tree is still a postorder traversal as discussed above. The only difference is that a parent will need to store the pointers to

This effort is partially funded by the National Science Foundation award: OAC 1549812

its children in a list or a similar data structure. Recursive calls are made on all the children, identifying the largest item amongst them all.

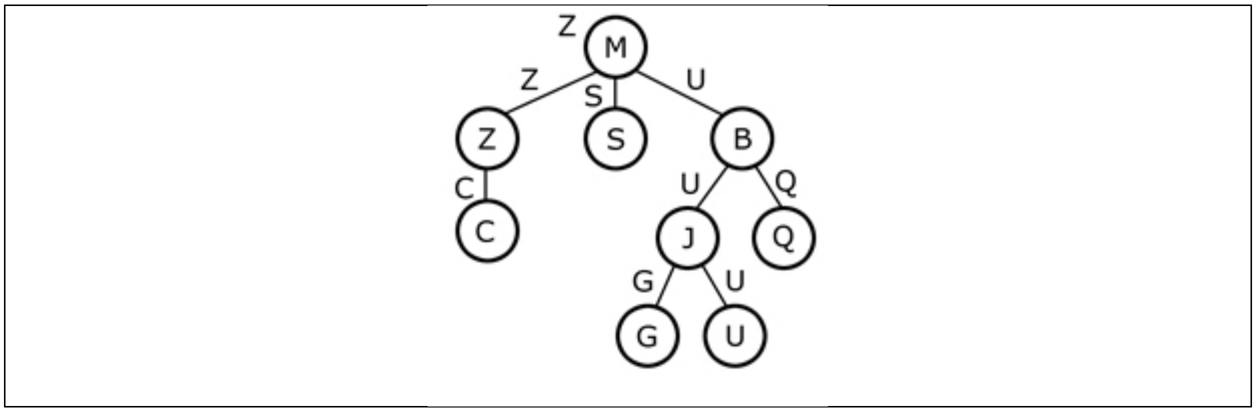


Figure *15*: Search tree.

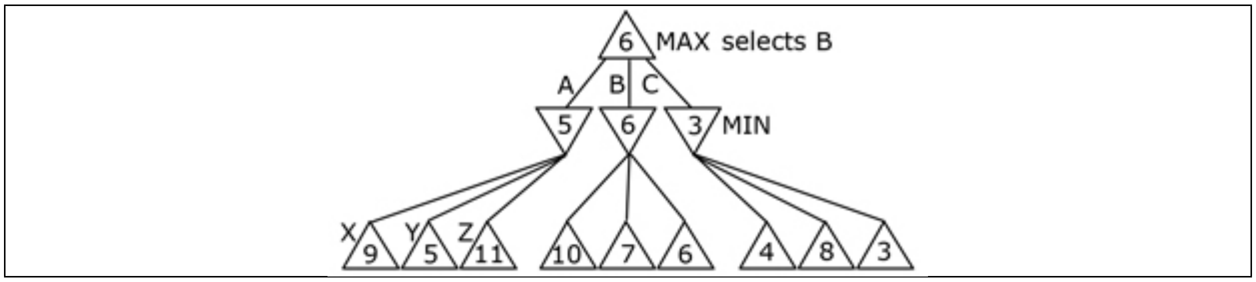
Now suppose the general tree only stores items in the leaves, rather than at each node. This simplifies the post-order traversal for the largest item as now the traversal simply returns the largest item amongst the children rather than comparing this item to the item stored at the parent node as well.

Problem Description

For this example, you will implement a TicTacToe game that utilizes a MiniMax tree to find optimal moves.

<u>MiniMax Tree</u>

Consider a very simple two player game where "Max" can make one of three possible moves (A, B, or C) as shown in Figure 16. If Max selects move A, its opponent, "Min", can respond with three moves of its own (X, Y, Z). After Min takes its turn, the game is over, and the score of the game can be determined. Max is trying to maximize the score while Min is attempting to minimize the score. Suppose the results for Min's three possible moves when Max has selected move A are (9, 5, 11). Thus, if Max selects move A, Min should select move Y to minimize the opponent's possible score. This is the optimal move for Min to take in this situation. For Max to find its optimal move, it needs to figure out how Min will respond to each of its possible moves (A, B, or C) and select the move generating the maximum possible result. This analysis works when it is Min's turn as well, except that Min will be selecting the smallest value from its possible moves.



This effort is partially funded by the National Science Foundation award: OAC 1549812

Figure *16*: MiniMax tree.

Of course, in selecting its move, Max is assuming that Min will play optimally, which may not be the case. Thus, in games that have many more turns for each player, Max must recompute its optimal move every time it is Max's turn to be sure that it is always maximizing its possible score for the current state of the game which can include suboptimal moves by Min.

A MiniMax Tree is a tree designed to find the optimal move in a two player game such as Tic-Tac-Toe, Othello, Checkers, etc. Each node in the tree represents a move by Max or Min, the two players in the game.

Consider a two player game as a general tree where all the possible game scores representing the different choices that Max and Min can make are stored in the leaves of the tree. If it is Max's turn to select a move, the optimal move that Max should make is a postorder traversal of the tree where selecting the maximum of the children or the minimum of the children alternates as you move through the levels of the tree.

Tic-Tac-Toe MiniMax
The goal is to have a computer playing Tic-Tac-Toe optimally as either X or O. Let's use the **MiniMax Tree** scheme to achieve this.

If Max is the Xs player and Min is the Os player, the MiniMax Tree is a general tree where the root node, an empty 3x3 grid, is Max's turn as Xs go first in Tic-Tac-Toe. The root node has 9 children, and Max must eventually examine all 9 to find its best move. First, Max places an "X" in the (1,1) location of the grid. Now Min ("O") has eight locations that it can move. This process of playing the game continues until a leaf is reached and a value is assigned to that leaf, called a terminal state. **Note that the "general tree" is represented by the 3x3 Tic-Tac-Toe grid being filled up as moves are taken by Max and Min**.

With Max as Xs, then the game outcome at the terminal states of the MiniMax Tree is positive if Max has three Xs in a row, negative if Min has three Os in a row, and 0 if neither Xs nor Os has three in a row and all 9 spaces of the board are occupied. Further, if Max determines it can win by traversing the MiniMax Tree, Max should be encouraged to complete its three Xs in a row as quickly as possible. The value (10 - turn) as the game outcome where turn is the depth of the MiniMax Tree (that is, how many turns have been taken by both players) will achieve this. For example, a win by Max in 5 turns is valued as +5 while a win by Max in 9 turns is +1.

As discussed as a postorder traversal, Nonterminal/nonleaf nodes are either selecting the largest values returned by their children or the smallest, depending on whether it was Xs turn or Os turn as that node. The selected value is returned to the parent once all children have been analyzed. Eventually, a result makes it to the root. Now Max ("X") knows that it can win with a move in the (1,1) location on the grid. However, X may be able to win more quickly in some other location, so X must still examine all possible places that it can move. Thus, X next tries the (1,2) location in the grid. This requires that moves be removed from the board once analysis of that move is completed so that the grid is once again empty for Xs next move analysis. Also,

This effort is partially funded by the National Science Foundation award: OAC 1549812

numerous copies of the board/grid are created so that the branches do not overwrite one another's selected moves.

Once X knows the best place for its initial move, it sends the move to the Tic-Tac-Toe game, and that move becomes permanent. Now O takes its turn at the root and finds its best choice, but X already occupies one spot on the board, so O only has to consider 8 moves. Thus, the computer player can find its optimal move whether it is playing Xs or Os (or both).

Methodology

Even a simple game like Tic-Tac-Toe can result in an extremely large MiniMax Tree. It is possible to take advantage of parallelism in a MiniMax Tree traversal to speed up the computation. A postorder traversal does not care the order in which the children are visited, it only matters that all children are visited so all of the results can be compared to one another before the traversal gives the result to the parent.

Thus, when X is deciding on its first move, we can start up several (up to 9) separate threads (fork) to potentially speed up the speed of the traversal by a factor of 9. These threads will each take a different amount of time to finish as the depth of the minimax tree depends on the location of the test move. Therefore, we must wait (**join**) for all of the threads to finish before returning the result.

The order that threads finish is output in the console. For the serial case, these values are in ascending order. For the parallel case, any thread can finish at any time. The sequence will rarely repeat. It does appear that certain threads generally take longer than others. Why?

Implementation

Tools

- A text editor
- The Java SDK version 8
- The following TicTacToe code that you can complete: MiniMax.zip[11]

Recursive Parallel MiniMax

With Java 8, it is straightforward to convert the above algorithm into one that automatically uses multiple processors if available. However, we must identify the part of the algorithm that can safely be run in parallel. For TicTacToe MiniMax, the order that we examine each spot that max (or min) can select on their turn does not matter. However, we must wait for the analysis of all of the possible moves to complete before reporting which one is best. When we allow each possible move to be processed simultaneously (if processors are available), we refer to this as a **fork**. When we wait for the subtasks to complete before reporting the optimal choice, we refer to this as **join**. Thus, only after all of the possible move analyses have begun (fork) should we start to wait for them to complete and compare the results to one another (join).

[11] http://www.csc.tntech.edu/pdcincs/resources/CS2/minimax/Java/code/complete_me.zip

This effort is partially funded by the National Science Foundation award: OAC 1549812

RecursiveTask<T>
We must extend either **RecursiveTask<T>** or **RecursiveAction**. This decision is based on whether the recursive procedure returns a value or not. Our recursive procedure returns the "score" (an integer) of the max or min terminal state encountered in that subtree which will be translated into a tic-tac-toe move (1-9). We will need to extend RecursiveTask<Integer>.

compute()
We next override the compute() method. This is the method that is automatically called when a new thread is forked. This can be a little tricky. We don't really want to start new parallel tasks if the size of the subtask is small. This is because forking and joining can take quite some time. If we do too much of it, the serial version can actually run faster! In our case, we will only fork the moves being analyzed at the top level. For example, if it is Xs first move, we would ideally like to have 9 threads analyzing simultaneously all 9 locations that X can select. The amount of processing required rapidly decreases as we proceed down through the minimax tree, so we should not start new threads at lower levels in the tree. Note that when it is Os turn, ideally we would like 8 threads (as X has already selected a location), and so forth.

Our top level thread will have thread id 0 (see later). If we want to restrict forking and joining to only the top level possible moves for X or O, then compute will only "dispatch" to a method that involves forking and joining if the thread id is 0. Any other thread calling compute will "dispatch" to a method that does not spawn new threads. Both method varieties return the "score" of that location (obtained when a terminal state was reached), and this should be returned.

Fork and Join
The two methods **maxForkJoin** and **minForkJoin** are the methods that spawn new threads. At most, we need 9 threads for TicTacToe (for Xs first move). Therefore, we can create an array of type TicTacToeMiniMaxParallel with size 9. Then, inside a for loop that loops over each possible move location, create a new TicTacToeMiniMaxParallel object with the appropriate parameters, place it in the array of threads, and fork it. In a separate loop, join any threads that are not null (remember, some locations may have already been selected). As join returns the score for that location, keep track of the best "score" obtained and its associated location, and return the optimal location.

ForkJoinPool
Lastly, we need a ForkJoinPool object so that our forked subtasks can effectively utilize the available processors. We do not need to create a new object to do this, we can simply call ForkJoinPool.commonPool() to obtain the required object.

We can do this in the TicTacToeMiniMaxParallel class (in the static search method, which is called by TicTacToe to obtain the optimal move) where we run and time the parallel version of the algorithm. Simply create a top level instance of the class with appropriate parameters (and make sure to set the thread_id to 0). Then use the ForkJoinPool object to start up the top level thread. The ForkJoinPool invoke method will return the optimal move (1-9) since we are using RecursiveTask<T>. This move is then returned to the TicTacToe game.

This effort is partially funded by the National Science Foundation award: OAC 1549812

You can create a working TicTacToe program by completing the sections in the downloadable **MiniMax.zip** file marked **DO THIS** in the starting source code for TicTacToeMiniMaxParallel. You will only need to start threads for the root and first level branches in the minimax tree. Should you extend RecursiveAction or RecursiveTask?

2.5 Review Questions

1) Describe what a thread is. Describe the relationship between a thread and a process. What is shared between threads in a process? What is not shared between threads in a process?
2) Describe what a race condition is and give an example. Do not use the example given in the Chapter.
3) Describe what is meant by fork-join.
4) What OpenMP C++ pragmas discussed in the chapter fork threads? Given an OpenMP program written in C++, discuss when threads join and give an example.
5) What is a reduction? Describe a situation, other than the examples given in the chapter, where a reduction would be useful.
6) Why are there rules that must be followed when designing solutions to problems that will use the ForkJoin Framework?
7) The `compute()` method that you must override takes no parameters. How do you provide "parameters" to your solution?
8) Why is it necessary to complete all forks before any joins?
9) Why is "work stealing" important?
10) What is the difference between a RecursiveTask and a RecursiveAction?
11) Why is it important that all of your parallel tasks be independent of one another?
12) Why is it important to determine a problem size that is too small to justify starting new threads?

2.6 Programming Exercises

1) CS1/CS2 - Suppose there is an array of *n* elements which will be processed by p threads in parallel and *p* is much less *n* ($p << n$). The threads are numbered from 0 through *p*-1 and the elements are indexed 0 through *n*-1**.** To process the n elements by the threads in parallel we need to determine how many elements each thread will process and which elements they will process. A very simple way of doing it is to divide the element as equally in a linear fashion. If n is not evenly divisible by *p* then the last thread will process the remaining elements. For example, if there are 100 elements and 10 threads then each thread will process on 10 elements. Thread 0 process elements 0 through 9, thread 1 will process elements 10 through 19, and so on. Thread 9 process elements 89 to 99. But if there are 99 elements then the thread 9 will process 9 elements from 89 to 98; or if there are 102 elements thread 9 will work on 12 elements from 89 – 101. Write the equations/expression in terms of *p* and *n* the range of elements that thread number *i* will process.
2) CS1 - Suppose an array *x* contains *n* integers. The **serial algorithm** to find the first occurrence of the maximum value and its location in *x* can be implemented as

```
max_value = x[0];
max_location = 0;
for (i= 1; i < n; i++)
```

This effort is partially funded by the National Science Foundation award: OAC 1549812

```
    {
      if (x[i] > l)
          {
            max_value = x[i];
            max_location = i;
          }
    }
```

Write a parallel version of the serial algorithm using OpenMP. [Hint:

i. Create two arrays say `local_max` and `local_ max_location` to store the maximum values and their location found by each thread. Set the size of the array to some value larger than equal to the possible number of thread.
ii. Divide the array among the threads equally. Each thread will find the local maximum value and its location. The thread can then store it local maximum value and its location in the shared array. Once all the thread are done finding and storing their local maximum values and their locations the master thread then can then find the global maximum from the local maximums using the sequential algorithm.
iii. In the parallel region
   1. Determine how many threads are in the team.
   2. Determine the work share of each thread *i.*
   3. Determine the beginning and end iteration number for a thread *i.*
   4. Using a `for` loop find the local maximum and its location for the thread and store the value in the *i*th locations in the arrays created in step 1.
   5. Find the maximum value from the `local_max` array and its corresponding location from the `local_max_location` array.]

3) CS2 - Write the Matrix Multiply using the parallel `for` loop and `reduction` operator in C++ and OpenMP.
4) CS1 - Consider the Parallel Number Summing example in C++ for CS1 given in the chapter.
   a) What is the average time for twenty runs of the serial version of the code?
   b) What is the average time for twenty runs of the parallel version of the code?
   c) Calculate the speedup of the parallel version. Is the parallel code significantly faster?
   d) The Methodology section for the parallel sum example described how you decompose the summation routine to parallelize it. Obviously, OpenMP did all the work for you. How many elements of the array do you think OpenMP assigned to each processor? Hint: have your code print the number of threads in the computation (the function omp_get_thread_num() returns the number of threads).
   e) Write a version of the OpenMP code that does not use the OpenMP reduction pragma. Hint: Before the parallel block, declare a local variable for each thread to hold the sum. Used the threads id, which you can obtain by using omp_get_thread_num(), to use the right local variable for the thread. Then, after the parallel block, have the master thread sum the local variables.
   f) The parallel sum example uses reduction in a very simplistic way. Go online and search for documentation of the OpenMP reduction pragma. Describe its behavior in the general case.
5) CS1 - Consider the Image processing example in C++ for CS1 given in the chapter.

This effort is partially funded by the National Science Foundation award: OAC 1549812

   a) What is the average time for twenty runs of the serial version of the code (i.e. with pragmas commennted)?
   b) What is the average time for twenty runs of the parallel version of the code?
   c) Calculate the speedup of the parallel version. Is the parallel code significantly faster?
   d) The Methodology section for the image processing example described how you decompose the image processing routines to parallelize them. Obviously, OpenMP did all the work for you. How many rows do you think OpenMP assigned to each processor? Hint: have your code print the image's height, and also have your code print out the number of threads in the computation (the function omp_get_thread_num() returns the number of threads).
   e) Modify the flip function so that it flips the image vertically instead of horizontally.Hint: When flipping vertically, you should flip the pixal at offset i with the pixel at offset (h-(i/w)-1)*w+(i%w), given that the image width is w, and the image's height is h.
6) CS2 - Consider the Image processing example in C++ for CS2 given in the chapter.
   a) What is the average time for twenty runs of the serial version of the code (i.e. with pragmas commennted)?
   b) What is the average time for twenty runs of the parallel version of the code?
   c) Calculate the speedup of the parallel version. Is the parallel code significantly faster?
   d) In the Methodology section for the image processing exercise, pipelining, which is a method of functional decomposition, is described. Describe how you would implement the solution using only domain decomposition. Describe how you would implement the solution using both functional and domain decomposition.
7) CS2 - Consider the Parallel Sort example in C++ for CS2 given in the chapter.
   a) What is the average time for twenty runs of the serial version of the code?
   b) What is the average time for twenty runs of the parallel version of the code?
   c) Calculate the speedup of the parallel version. Is the parallel code significantly faster?
   d) In this example, you used only two threads to sort the pieces of the array in parallel. However, you can use more threads if your computer's processor has more cores. If you use more threads, describe how the merge algorithm changes.
   e) Implement the parallel sort example such that three threads sort the array in parallel.
8) CS1 - Instead of breaking the array into two pieces, modify the Java parallel summation example to break the array into three pieces. Do you expect better performance? Why or why not?
9) CS2 - Explain how you could modify the Java Matrix Multiplication example to work on matrices that are not square and with rows and cols that are not powers of two.
10) CS2 - In the Java Radix Sort example, the compute() method only allowed the top level set of queues to be processed in parallel. Relax this restriction. Use System.nanoTime() to compare the new version to the original one. Explain your results.
11) CS2 - An algorithm called **alpha-beta pruning** is frequently used to increase the speed of the standard Minimax algorithm. Look it up and explain why it would be difficult to parallelize.
12) CS1 - Implement a serial, definition based (non-recursive, triply-nested loop) Matrix Multiplication solution. Use System.nanoTime() to compare it to the parallel Matrix Multiplication solution for several different (large and small) matrix sizes. Explain your results. Look up **Strassen's** algorithm. Modify the parallel Matrix Multiplication to use a

This effort is partially funded by the National Science Foundation award: OAC 1549812

parallel Strassen's algorithm. Compare it to the parallel Matrix Multiplication solution using System.nanoTime() for large matrices.

13) CS2 - Look up the Eight Puzzle. Solve the eight puzzle using both a serial and a parallel tree-based search.

This effort is partially funded by the National Science Foundation award: OAC 1549812