Drexel University
Electrical and Computer Engineering Department

ECEC 622
Introduction to Parallel Computer Architecture

Programming Assignment 1
April 29th, 2013

Team Members:
Julian Kemmerer, jvk@drexel.edu
John Carto, jjc323@drexel.edu

Design

PThreads

For the PThread implementation of the string search program, it was possible to reuse some of the code from the serial version of the program. The serial version made use of a data structure to queue up files for processing, and this implementation worked in the same way. The key difference was that the queue used here was only used for directories. After the initial directory is added to the queue, a single thread handles going through this directory and searching files as they are encountered. When another directory is found, the thread doesn't enter it, but rather places it in the queue and allows another thread to handle it. When finished with the current directory, a thread retrieves a new directory from the queue, or waits until more work is added to the queue. Each thread keeps a local count of all the string occurrences in its specific context. When no more work is available, these values are combined to produce a total number of occurrences. Pseudocode for the behavior of a single thread can be seen below:

```
while(1){
	wait_on_queue();

	//retrieve directory from queue
	for elem in directory:
		if elem is file:
			count_strings();
		else if elem is directory:
			add_to queue();
}
```

Fig 1. Thread execution structure. Synchronization/ ending sequence not shown.

While the above strategy isn't too complicated in theory, there are a few considerations necessary to handle multiple threads working off and adding to the same queue. In terms of synchronization, the queue is protected by a binary semaphore, which must be held by a thread in order to add or retrieve an item to/from the queue. A counting semaphore was also used for determining if there was work in the queue. Every time work is added to the queue, the semaphore was incremented. Threads wait on the queue until work is available, and then the counting semaphore is decremented as a thread retrieves a directory from the queue to work with. It is important to note that the counting and binary semaphores work together to provide exclusive access to the queue for each of the threads, while also eliminating needless busy waiting. These semaphores are always acquired and released in the correct order to prevent cases of deadlock as well.

The most difficult part of this implementation was determining a way to communicate to all threads that no more work was left to be done. This was done by having every thread increment a shared variable before waiting on the queue for work. If the last active thread finds the rest of the threads waiting, no more work is left, and the cleanup routine is triggered. The last thread flips a special bit, and sends n-1 signals to the counting semaphore before adding its count to the global variable and exiting.. The other threads awake, add their local counts to the global string count, and exit. The global variable is protected here with another semaphore as well. This process is illustrated in the pseudocode below:

```
if doneCount == numThreads-1:
       done = 1
       add_count_to_global();
       pthread_exit();

//protected by semaphore, other threads awake here
wait_on_queue();
       if done==1:
              add_count_to_global();
              pthread_exit();
```

Fig 2. Thread cleanup sequence shown.

For more specifics on this technique and implementation, please see the commented source files.

OpenMP

The OpenMP implementation was designed to dynamically allocate thread execution towards either 1) gathering a file list or 2) actually search the files themselves. At the start of the program (the OpenMP section, that is) all threads are allocated to finding new files and directories. This populates two queues: directories (to be searched) and files (to be opened and read). A pseudo code snippet can be seen below:

```
while( directories or files exist to be search )
{
       check_and_process_directories();
       check_and_process_files();
       sched_yield();
}
```

Fig. 3. Structure of each thread's execution.

A relatively simple algorithm where each thread checks first if more directories need to be searched, followed by searching through files that have been found. Pseudo code describing the two ‘check’ functions are show below:

```
check_and_process_*()
{
        if(queue is empty)
                return;
        list l = get_items(directory queue, number of items);
        if(list.size <= 0)
                return
        process_items(l);
}
```

Fig. 4. Structure of the ‘checking’ functions.

The code snippet above has an important feature. The ‘number of items’ argument allows for a single thread to request multiple files/directories to become its ‘responsibility’. This helps avoid threads quickly reading a file/directory and immediately going to acquire the shared global queue resource again. In other words, this allows for threads to have an ‘appropriate’ amount of work. Experimentally, the optimal number of files per thread was determined to be 10. Where the optimal number of directories per thread was determined to be 1.

The ‘process_items’ function simply loops through each item in the list for this thread. If they are directories, their sub directories and files are added to the appropriate shared global queues. If this function happens to be ‘checking’ files then the ‘process_items’ functions opens each file in the list and looks for the desired text.

This may seem to be a feasible and well performing algorithm, however, it suffers from a sort of live-locking. That is, the code is running but nothing productive (or nothing *very productive*) is happening. Threads are jumping from directory-processing to file-processing as highlighted in Fig. 3 and the problem exists when a single thread reads directories off of the directory queue while other threads have finished their work and are waiting (idle while looping) for more. Consider the initial starting case explained below:

Table. 1. Execution of original algorithm over a handful of context switches.

Time	**Threads**			
0	Thread 0: Remove /home/DREXEL/nk78 from dir queue			
1		Thread 1: No dirs, no files	Thread 2: No dirs, no files	Thread 3: No dirs, no files
2	Thread 0: Place a few files/dirs into the files/dirs queue			
3		Thread 1: Some dirs / files to removed from queues	Thread 2: Some dirs / files to removed from queues	Thread 3: Some dirs / files to removed from queues
4	Thread 0: Place a few files/dirs into the files/dirs queue			

The starting directory is taken off of the directory queue by a single thread. There are no files yet so the other threads loop continuously waiting for a directory or file to appear. When a file or directory does appear it is likely that they too are immediately taken off the queue by other running threads. This repeating pattern causes the queues to rarely become full beyond a few items. The threads reading the files/directories and the threads searching the files are either 1) forced to run in sort 'lockstep' with each other or idle in a while loop.

This was fixed in the final implementation. It was required that the file queue have a significant number or 'buffer' entries. That is, there needs to be a sufficient amount or files to search through before assigning multiple threads to the task. This is completed by forcing the program to collect at least 1,000 file names. This points out an immediate downfall in this implementation. It must be known that over 1,000 files even exist! Assuming this can be implemented in a different way (possibly via looking at how often threads are iterating and discovering queues with no files and no directories, then adjusting for a larger buffer) the program can still be considered a valid implementation.

For more specifics on this technique and implementation, please see the commented source files.

Performance

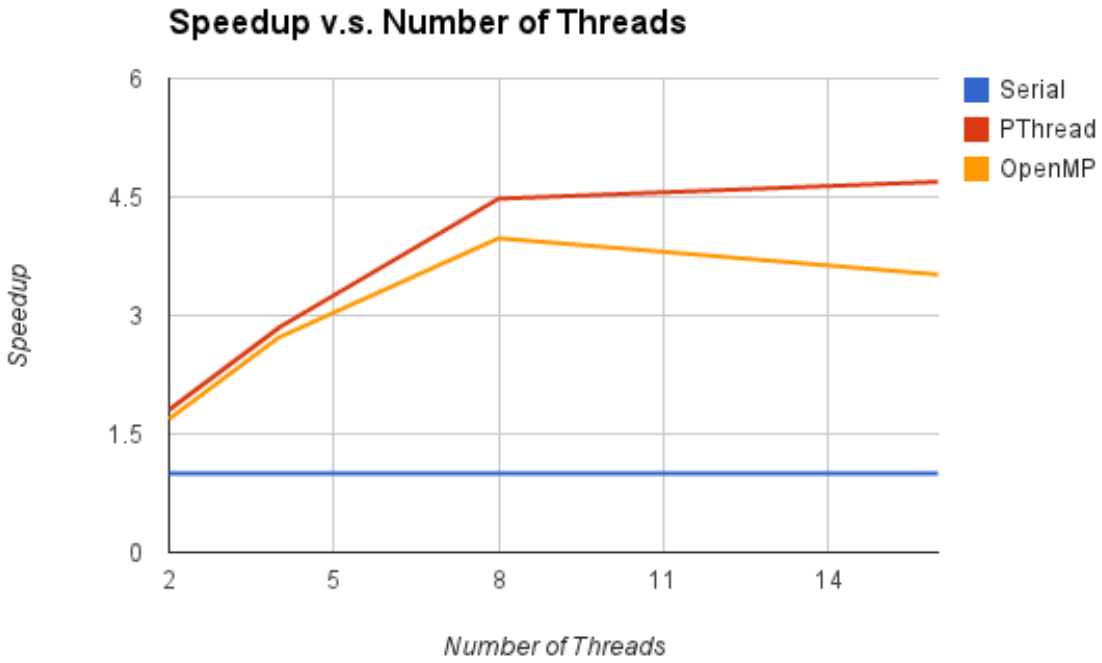

Fig. 1. Speedup achieved over the serial version for 2, 4, 8, and 16 threads

Figure 1. shows that both PThread and OpenMP implementations are indeed faster than their serial single threaded counterpart. The speedup achieved through multithreading maxes out at approximately 4.5x faster. This is achieved with approximately ~8 threads of execution for both PThread and OpenMP. It appears that the OpenMP implementation begins to suffer from an excess of threads earlier than the PThreads implementation; further experimentation is needed.