Project 1: Multithreaded Programming

Preston Maness
Ingram School of Engineering
Texas State University
San Marcos, Texas
Email:aggroskater@gmail.com

***Abstract*—This project introduced us to threaded programming, predominantly with pthreads and OpenMP. A small exercise with Win32 threads was also included. Part 1 required us to crunch the fibonacci sequence in a separate thread from the main calling program. This entailed ensuring that the main thread would wait for the fibonacci thread to complete. Part 2 introduced a task-parallelizing implementation of matrix multiplication by treating each product matrix element as a task to execute in parallel. Part 3 required the searching of a large array for specific, matching elements using serial code, and code parallelized with OpenMP and pthreads. Speedup characteristics with 2, 4, and 8 cores are investigated. Non-ideal speedup is explained.**

CONTENTS

I. PART 2B: MULTITHREADED QUESTIONS

1) The advantages of multithreading are speedup. The disadvantages are the added complexity to ensure thread-safeness.
2) The code is not applicable for 100-by-100 matrices. The code as it stands creates MxN threads, and then waits for all of them to finish prior to destroying them. As soon as the number of threads exceeds the number of physical cores, context switching will occur to balance the various threads. It would be a better idea to implement a thread pool with the number of physical cores. This pool remains active and is constantly fed new tasks to complete once the current task is finished.
3) This code is not efficient for 1M-by-1M matrices. Creating a thread pool only addresses the thread creation overhead. However, the algorithm still requires grabbing memory from the global text section of the program. For 1M-by-1M, the matrices are likely to be allocated on the heap. As well, the algorithm has to grab all of matrix A's columns for a given row, and all of matrix B's rows for a given column, when crunching any given element of the resultant matrix C. Even if the threads are modified to reference a single global array in memory and read from that array –this action is presumably still safe, since no two threaded events are ever reading/writing the same memory– that array is very large. More importantly, that array is in external memory, and grabbing from external memory is time-costly, since it is not local (L1 or L2 cache). A possible way to mitigate this (to an extent), would be to store each matrix's rows and columns discretely. This can permit more efficient cache manipulation.

II. PART 3: SPEEDUP ANALYSIS

A. Results

technique	1 core	2 cores	4 cores	8 cores
serial	1	1	1	1
OpenMP speedup	0.9127	1.7319	2.8533	2.7517
pthreads speedup	0.8323	1.4216	1.5709	1.9950

These speedups are calculated in the proj3.c program. Each run of the program runs the task in serial, OpenMP, and pthreads, and then prints out the speedup compared to the serial case when using OpenMP and pthreads. The above table simply lists "1" as the reference speed for serial runs. In each run of the program, the run time for serial execution differed slightly of course. However, we are concerned with the average speedup OVER this time.

This average speedup was calculated by running the program five times and averaging the speedups reported for OpenMP and pthreads implementations.

B. Conclusions

This program was executed on a machine with 4 physical cores, 3 GB of RAM, and a GNU/Linux Debian install running Linux kernel 3.10. The constant NUM_THREADS was adjusted in source, and the program recompiled, for each of N=1, N=2, N=4, and N=8.

In both the OpenMP and pthreads implementations, we can see the performance penalty of thread creation and destruction when using only one thread. The threaded implementation is slightly slower than the serialized code in this case.

We can see that OpenMP is more efficient than my likely inferior pthreads implementation. The speedup is approximately the expected (N-threads - 1) up to N=4. Remember that the main() function still exists in its own thread and so only 3 threads are really available to be utilized. However, once 8 threads are requested, the effects of context switching by the kernel are apparent and the OpenMP implementation actually suffers a slight degredation.

However, the pthreads implementation, while not as effective as the OpenMP implementation, still exhibits improvement over the serial case. In the phtreads implementation, it appears that N=8 also experienced some speedup over N=4, presumably since the speedup was still not reaching 3.0.

The likely causes of the slowdown in my inferior pthreads implementation are caching issues I have been unable to remedy. I suspect that the reduction operating that the OpenMP implementation is transparently utilizing is the source of its speedup over my pthreads implementation.