Introduction:

We were invited to participate in the Summer Science Institute, a 10 week full time summer undergraduate research program at the University of Mary Washington. For our research, we studied the performance of various parallel programming libraries. This was set to be an interesting summer because Zach had only a little parallel programming experience, while Jerome did not have any at all. Zach previously wrote and application for the LittleFe that monitors the core utilization on each node in real-time using MPI.

As a standardized test, each piece of hardware solved a travelling salesman problem using an un-optimized brute force algorithm. The traveling salesman problem consists of a certain number of cities, each a known distance away from all the other cities. The solution to the problem is the shortest path which goes through all the cities and returns to whatever city at which the path started. We tried problems varying from 10 to 13 cities on a LittleFe portable computing cluster.

The brute force algorithm has a big-O complexity of x!,

Learned:

The brute force solution to the travelling salesman problem is embarrassingly parallel. When the traveling salesmen wants to visit every city in a designated area and return home using the tour with the cheapest cost, he has N! possibilities where N is the number of cities. It is necessary to simply assign a subset of all possible tours to each available thread, and then compare the best solutions found by each thread. To perform this test on all the systems available to us, we had to learn several programming extensions/libraries, including openMP, CUDA, and MPI.

In our experience, openMP was by far the easiest extension to work with. The #pragma directive was quite intuitive to understand. We found that knowledge of how to use a for-loop was almost all that was needed to utilize a pragma directive. Using openMP required almost no knowledge of the hardware to run, and only basic concepts such as race conditions and critical sections to write a scalable program. It is also easy to implement critical directives when absolutely necessary. Additionally it requires no software setup other than adding a library link directive at compile time.

As a slight variation on our original openMP implementation, we ran openMP natively on Intel’s Xeon Phi coprocessor. The difference in hardware made no difference as far as the code was concerned, but introduced considerable overhead after compilation to execute the program on the coprocessor. Furthermore, when we first got the system containing the coprocessor, we learned that the coprocessor would overheat without proper an-speed configuration. Because of this phenomenon, we also found our performance dropped, and would eventually fail as temperature rose.

CUDA was the most difficult to use. Utilizing the GPU required the most knowledge of hardware memory, as well as the comprehension of how to utilize thread blocks. The method by which you transfer memory from the CPU to the GPU is difficult for programmers new to parallel programming. Of especial confusion was the need to reserve and name device memory on the host system, even though the host system should not be interacting with that memory. Also, the GPU can only run functions written specifically for the GPU, which sometimes forces the programmer to rewrite functions for use on the GPU. The most difficult problem we experienced with CUDA was the “watchdog timer”. This timer silently kills lengthy kernel CUDA functions running if the GPU is displaying the systems video. As a work around, the problem did not exist when Xwindows was terminated. Additionally, we found CUDA notoriously difficult to debug because errors in the kernel function are not handled well within the CUDA library. As an example, when the watchdog driver killed our task, then next CUDA library call would return an “unspecified launch error” regardless of the validity of the call, because the error was outstanding from the Kernel code.

MPI was middle of the road in terms of difficulty. The Bootable-Cluster CD (BCCD) software on the LittleFe greatly simplified the environment setup needed to distribute and run an MPI program across multiple computer nodes. Jerome had some trouble determining intuitively which variables within the program were shared and which variables existed as private copies for each thread. We both found it easy to perform node-specific operations using conditional logic. As a side project, we wanted to compare the efficiencies of pure MPI solutions on the LittleFe to hybrid MPI/openMP solutions.

In our experience, we were able to easily grasp the basic concepts of parallel computing. As we already noted, utilizing openMP was not much different from us than using a for-loop, specifically how you can make each process work on different tasks based on each task’s thread ID. The concept of race conditions and the performance loss from critical sections was also easy to grasp. Some other concepts that we were introduced to, such as cache management and consecutive memory accesses were more difficult to understand, especially since we were not yet using such methods in our sequential programming. Much of the documentation we found dove directly into these more advanced optimizations before giving a satisfactory explanation as to how to implement basic parallel programs. One of the more difficult problems we had to deal with was setting up the proper environment to perform parallel programming rather than the actual programming itself. Debugging parallel programs is less intuitive than there sequential counterparts. This is largely because different threads can be at different places in the program at a given time. It is therefore difficult to tell if a thread crashes or segfaults exactly which process caused the error. Furthermore, errors were sometimes caused by forces outside of the programs themselves. The “watchdog timer” problem we encountered using CUDA was exceptionally difficult to debug because we expected the problem to be in our code. As another example The Xeon Phi Coprocessor threw cryptic error messages. We again thought that the problem was in our code, until the monitoring program for the coprocessor showed us that the coprocessor was overheating.

In our experiments with the LittleFe, we compiled some interesting performance differences from our openMP/MPI version to the pure MPI version. Being that the littleFe system is a six node cluster were each node has a dual core CPU, our hybrid version initialized six MPI processes (one on each node), and let openMP take each core on their respective system into account. With the pure MPI version we just initialized twelve MPI processes (two one each node) letting each core in the system hold their own process. We thought that this would lead to repeating the existing overhead we had in our algorithm across every node, which would cause an initial drop in performance. This was supported in our results the showed a distinct significant performance loss in a lower number of cities with a high number of MPI processes. Additional to the repeated overhead it was found that it took on the order of a second to actually start multiple MPI processes across every node. So the hybrid version started out preforming the pure MPI version one less cities, but as the total permutation rose the MPI version quickly overtook the hybrid version. Originally we thought the hybrid would outperform the pure MPI version. In the hybrid version the two threads would collect their respective data on their own node and then send that data to the host node whereas in the pure MPI version data from each node would send two packets, of data separately from each core to the host node. As a result we thought a network bottleneck would be formed from sending separate packets from a single node, rather than sending one from each. Zach believes that our results could be cause by the MPI an easily optimized method, whereas the openmp interface leaves more room for node specific memory collisions.

For future work in our learning parallel computing, a good next step would be heavily optimizing the code we have written. Although we performed simple optimizations such as reducing communication between processes, we could implement other optimizations such as increasing the rate of cache hits. In our research we shied away from optimizing too heavily for fear that our optimizations might significantly favor one system over the other. Furthermore, we could optimize the algorithm we used to solve the TSP. Due to differences in what can be done with a multi-core system and a GPU, this would have necessarily entailed optimizing for each system as well.

We would like to thank Dave Toth

We would like to thank Ian Finlayson for his assistance in debugging the different versions of our program, especially the CUDA version. We would also like to thank him for reminding us how to cause an intentional segmentation fault within the code for debugging purposes. We would like to thank the University of Mary Washington for providing funding, space, and support for this project. We would like to thank NVidia® for providing a GPU to program.