*Brindan Adhikari and Ankit Hriday*

*ASEN 4057*

*Final Project*

*May 4, 2018*

**Final Project Report**

**I. Description of C Program to Improve**

For the final project, we chose to optimize the sorting algorithm code. This program is a selection sort algorithm which sort the arrays. The selection sort algorithm scans all items and finds the smallest, and then swaps it into the first position. It repeats the process on the N-1 numbers remaining and continues until all the numbers have been checked one by one. This method is inherently slow; the best and worst case scenario will be of order O(N^2) operations. To sort the largest array “I” (N=1,000,000) the program took several hours. Thus, it is evident that a better algorithm should be used to improve performance.

**II. Part 1: Serial Code Optimization**

**a. Original C Code Profile Report and Discussion**

GPROF was used to evaluate the performance of the non-optimized code for sorting array “H” (N=100,000) and is shown in *Table 1*. It is labeled selection\_sort\_analysis.txt and is located in the Profiling directory in the submission folder.

**Each sample counts as 0.01 seconds.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **% Time** | **Cumulative**  **Seconds** | **Self Total**  **Seconds** | **Calls** | **Self**  **ms/call** | **Total**  **ms/call** | **Name** |
| 100.61 | 63.94 | 63.94 |  |  |  | main |

***Table 1: Flat profile for selection sort of a N=100,000 array.***

From Table 1 we see that the time taken to sort the N=100,000 array was 63.94 seconds. The time to sort array “I” was too long to get results.

**b. Proposed Serial Code Improvements**

To improve the serial performance of the code, we employed a merge sort algorithm. Merge sort works by dividing the input array in half and then using recursion, it calls itself to further divide each half in half again until they can't be divided in half any further. It then reconstructs the sorted array by calling merge on the two arrays. Merge assumes the two arrays are sorted and merges them into one. It then works its way back up until the entire array is merged.

This is a significant improvement to the selection sort algorithm utilized above. It greatly reduces the number of operations as not every number has to be compared with every other number. It can be shown that merge sort performs the operations on the order of O(NLogN) for all scenarios since it always divides the array into halves and takes a linear time to merge the two halves.

**c. Timing Comparison**

Comparing the selection sort code time from *Table 1* to the time from the merge sort code in *Table 2,* we see that there is a drastic speed up. The selection sort code took **~64 seconds** while the merge sort code took **~.03 seconds.** The merge sort performs nearly **~85,000** times better than the selection sort code. Using our estimates described above for O(N^2) (selection sort) vs O(NLogN) (merge sort) for the N=100,000 array we get 500,000 and 10,000,000,000 operations respectively. So we would expect the merge sort to do ~**20,000** times better.

**d. Optimized C Code Profile Report and Discussion**

*Table 2* shows the GPROF flat profile and *Table 3* shows the call graph results of the merge sort C code using matrix “H” (N=100,00). The text file is labeled *merge\_I\_analysis.txt* and is in the profiling directory in the submission folder.

From the flat profile we see that most of the time was spent in the merge function ~100.41%, the percent the time spent in the other functions was minor and therefore reported as 0%. We also see that merge was called 99,999 times, this makes sense as the array was of size 100,000, thus it had to merge 99,999 times. Furthermore, we see that the total time taken (.03 seconds) is insignificant compared to the selection sort code (64 seconds). From the call graph we see that mergeSort was called recursively indicated by the + sign.

**Each sample counts as 0.01 seconds.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **% Time** | **Cumulative**  **Seconds** | **Self Total**  **Seconds** | **Calls** | **Self**  **ms/call** | **Total**  **ms/call** | **Name** |
| 100.41 | 0.03 | 0.03 | 99999 | 0.00 | 0.00 | merge |
| 0.00 | 0.03 | 0.00 | 1 | 0.00 | 0.00 | BuildMatrix |
| 0.00 | 0.03 | 0.00 | 1 | 0.00 | 30.12 | mergeSort |

***Table 2: Flat profile for merge sort results sorting “I” N=100,000 array.***

**granularity: each sample hit covers 2 byte(s) for 33.20% of 0.03 seconds**

**index %time self children called name \_\_\_\_\_\_\_\_\_**

0.03 0.00 99999/99999 mergeSort [2]

[1] 100.0 0.03 0.00 99999 merge [1]

----------------------------------------------- -----------------------------------------------------

199998 mergeSort [2]

0.00 0.03 1/1 main [3]

[2] 100.0 0.00 0.03 1+199998 mergeSort [2]

0.03 0.00 99999/99999 merge [1]

199998 mergeSort [2]

----------------------------------------------- -----------------------------------------------------

<spontaneous>

[3] 100.0 0.00 0.03 main [3]

0.00 0.03 1/1 mergeSort [2]

0.00 0.00 1/1 BuildMatrix [4]

----------------------------------------------------------------------------------------------------

0.00 0.00 1/1 main [3]

[4] 0.0 0.00 0.00 1 BuildMatrix [4]

----------------------------------------------- -----------------------------------------------------

***Table 3: Call graph for merge sort results sorting “I” N=100,000 array.***

**Index by function name**

*[4] BuildMatrix [1] merge [2] mergeSort [3] main*

**Makefile**

The makefile for the serial code using merge sort is called serialMake and is in the Submission folder.

**II. Part 2: Parallelize C Code**

**a. Proposed Strategy for Code Parallelization**

To parallelize the merge sort function, we used MPI. The strategy used was to break up the initial array into several pieces equal to the number of processors. This was challenging because if the array was not divisible by the number of processors there were remainders that had to be dealt with. Thus, some arrays had one more value than others. After the array was broken into pieces, the pieces were sent to each processor. The processors then received and sorted their prospective arrays using merge sort. The most difficult part was designing the algorithm to build the pieces back up. Originally, we simply sent each piece to the server rank and merged them as they came. However, this was not the optimal solution. Thus, we made the algorithm send the arrays in an alternating fashion to the processors. At each step one half of the remaining processors can't be used as the other half merges two arrays. Finally, the server rank does the last merge and the array is sorted.

The file containing the finished source code is called **final\_mpi\_merge\_sort.c.** All the functions are housed in this file as we had trouble getting the header file to work with mpirun. We also left **mpi\_merge\_sort.c** in the Submission folder as it better shows the process we took to get there.

We want to clarify that a much more elegant implementation is labeled **merge.c** (which is copy and pasted from the internet just for demonstration) and it works perfectly.

**b. Scalability Study**

Comparing times using 1,2,4,8,16 processes, we see that the codes is able to scale with the number of processors. As the number of processors increase the time decreases until you get past 4 processors. We think this is due to the fact that the computer we were running the code on could only utilize 4 cores. We can see that the time goes down by about a factor of two when the number of processors is doubled up to 4 processors. This is roughly what we would expect as the work is being split in half each time the number of processors is doubled. There are still unknowns like communication that will cause it to not be directly proportional. As the processors increase past 4 to 8 and 16 we see a slow down due to the fact of increased communication between the cores without any extra processing taking place.

|  |  |  |
| --- | --- | --- |
| **Number of Processors** | **Time(s)**  **N=100,000 “H”** | **Time(s)**  **N=1,000,000 “I”** |
| 1 | .01 | .18 |
| 2 | .006 | .08 |
| 4 | .004 | .04 |
| 8 | .01 | .04 |
| 16 | .05 | .19 |

***Table 4: Scalability study using 1,2,4,8,16 processors.***

**c. Profile Report and Discussion**

*Table 5* shows the flat profile chart and *Table 6* shows the call graph for the code run on 4 processors. The file is in the Submission/profiling folder labeled **mpi\_merge\_4\_I\_analysis.txt.**

**Each sample counts as 0.01 seconds.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **% Time** | **Cumulative**  **Seconds** | **Self**  **Seconds** | **Calls** | **Self Calls**  **ms/call** | **Total**  **ms/call** | **Name** |
| 100.55 | 0.01 | 0.01 |  |  |  | libc\_csu\_init |
| 0.00 | 0.01 | 0.00 | 749996 | 0.00 | 0.00 | merger |
| 0.00 | 0.01 | 0.00 | 249998 | 0.00 | 0.00 | mergeSort |

***Table 5: Flat profile for merge sort run on 4 processors for “I” N=100,000***

**granularity: each sample hit covers 2 byte(s) for 99.46% of 0.01 seconds**

*index % time self children called name*

<spontaneous>

[1] 100.0 0.01 0.00 \_\_libc\_csu\_init [1]

0.00 0.00 749996/749996 merger [2]

0.00 0.00 249998/249998 mergeSort [3]

----------------------------------------------- ------------------------------------------------

0.00 0.00 749996/749996 \_\_libc\_csu\_init [1]

[2] 0.0 0.00 0.00 749996 merger [2]

----------------------------------------------- ------------------------------------------------

0.00 0.00 249998/249998 \_\_libc\_csu\_init [1]

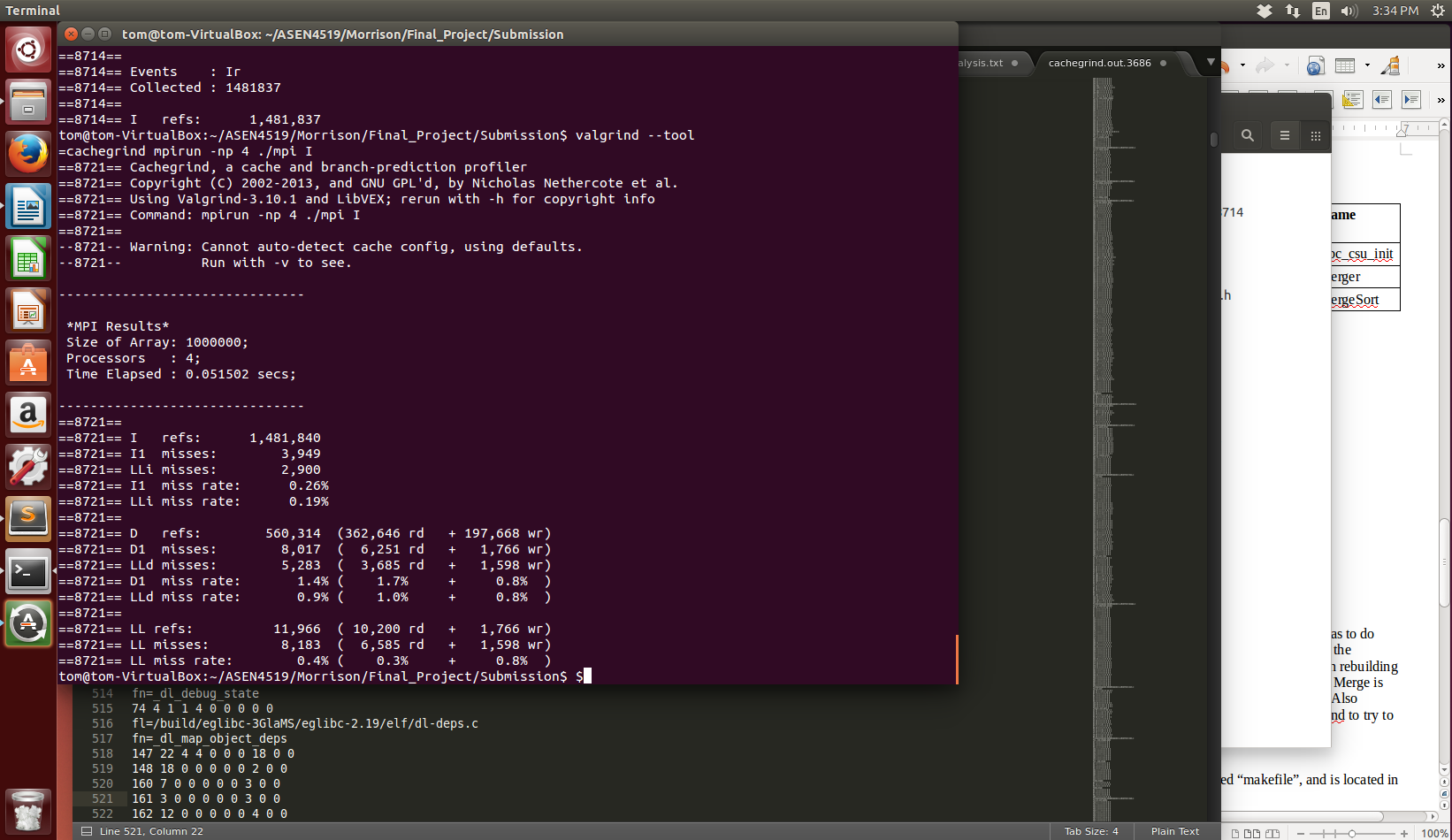
[3] 0.0 0.00 0.00 249998 mergeSort [3]

-----------------------------------------------------------------------------------------------

***Table 6: Call graph for merge sort results run on 4 processors sorting “I” N=100,000 array.***

**Index by function name**

*[1]\_\_libc\_csu\_init [3] mergeSort [2] merger*

From *Table 5* we see that 100% of the time was spent in libc\_csu\_init, we think this has to do with MPI and the different cores. This profile report is a bit confusing as it only lists two of the functions in the program, merger and mergeSort. Merger should have only been called when rebuilding the array and should have only been called half as many times as the number of processors. Merge is the other function that is missing from the profile and should have been called many times. Also, mergeSort should have been called recursively similar to table 3. Additionally, we used valgrind to analyze the cache misses. When using 4 processors as shown in the picture below the LLi miss rate was .19% the Lld miss rate was .9%. This was almost identical to the cache misses for one processor.

**Makefile**

The makefile for building the parallelized merge sort source code is calsubl led “makefile”, and is in the Submission folder.

**Make:** make

**Compile (without make):** mpicc -o mpi\_merge final\_mpi\_merge\_sort.c

**Run:** mpirun -np 4 ./mpi\_merge

**IV. Summary of Findings and Potential Future Improvements**

In summary, there were drastic improvements to the run time for sorting arrays. The biggest improvement resulted from changing the algorithm from selection sort to merge sort and was relatively easy. This emphasizes the importance of algorithm design. There was also significant improvement by using MPI; it is even more drastic as the array size increases and the number of processors increase. All though comparatively the improvement from MPI vs. updating the algorithm was minuscule and much harder to implement. This suggests that algorithm design should be a high priority.

Future improvements should include, cleaning up the code, (we took many paths and struggles trying to learn MPI. Trial and error lead to some extra steps that probably aren't needed). It could very much be simplified and reduced like the code we saw on the Internet, **merge.c. merge.c** utilized MPI\_Broadcast and MPI\_Scatter for the initial array break up and sorting. We were unfamiliar with those commands so we didn't utilize them. Other improvements would be to implement a better algorithm than merge sort. There seems to better algorithms out there, but it would be interesting to see if they could be parallelized as effectively.