Dalton Scharff

Prof. Bigrigg

CS 1645

30 January 2017

**Homework 1 Findings**

|  |  |  |  |
| --- | --- | --- | --- |
| **NUM\_OF\_RANDOMS: 10** | | **REPEATER: 100** | |
|  | Sort Whole (Part2\_combined.c) | Sort Halves and Merge (Part2\_split.c) | Parallel Sorting (Part5\_server.c) |
| Average: | 2μs | 1μs | 76921μs |
| Fastest: | 1μs | 1μs | 188020μs |
| Slowest: | 3μs | 4μs | 16494μs |

|  |  |  |  |
| --- | --- | --- | --- |
| **NUM\_OF\_RANDOMS: 1000** | | **REPEATER: 100** | |
|  | Sort Whole (Part2\_combined.c) | Sort Halves and Merge (Part2\_split.c) | Parallel Sorting (Part5\_server.c) |
| Average: | 18093μs | 13499μs | 75870μs |
| Fastest: | 17760μs | 13313μs | 151953μs |
| Slowest: | 18494μs | 13650μs | 5145μs |

|  |  |  |  |
| --- | --- | --- | --- |
| **NUM\_OF\_RANDOMS: 10000** | | **REPEATER: 100** | |
|  | Sort Whole (Part2\_combined.c) | Sort Halves and Merge (Part2\_split.c) | Parallel Sorting (Part5\_server.c) |
| Average: | 2819090μs | 1408677μs | 759247μs |
| Fastest: | 2800604μs | 1399519μs | 1080382μs |
| Slowest: | 2834849μs | 1416778μs | 520034μs |

|  |  |  |  |
| --- | --- | --- | --- |
| **NUM\_OF\_RANDOMS: 30000** | | **REPEATER: 25** | |
|  | Sort Whole (Part2\_combined.c) | Sort Halves and Merge (Part2\_split.c) | Parallel Sorting (Part5\_server.c) |
| Average: | 25805519μs | 12475601μs | 8512555μs |
| Fastest: | 25711946μs | 12439129μs | 12592765μs |
| Slowest: | 25884853μs | 12531974μs | 4775964μs |

Not until the number of random integers increased to 10000 was the network lag overtaken by the processor speed as the main bottleneck of the program. This means that at 10000 integers, the parallel sorting algorithm was the fastest (on average), but was trumped by split algorithm in all tests with a lesser number of integers.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2 CPUs | 3 CPUs | 4 CPUs |
| AVG | 4892743 | 8455987 | 9785730 |
| MAX | 5170885 | 9980401 | 10461546 |
| MIN | 4691124 | 7893504 | 9247051 |

The graph above shows that increasing the number of CPUs utilized to sort integers does not scale linearly and in fact increases in time as the amount of CPUs increase. However, just like when I compared the original 2 CPU runtime to that of the split and merged single CPU, there is a threshold under which increasing the number of CPUs utilized will actually give a undesired increase in time. Only when processing vast amounts of data would 3 cores process more quickly than 2. It now makes sense to me as to why some games and programs only utilize a maximum of 2 cores. If there is not a lot of data to be processed, or most of that data is being handled by a GPU’s cuda cores, it is not necessairily beneficial to utilize more cores.

**Explanation of Server/Client Interaction**

The key is how the server identifies a shared portion of memory. shmget identifies a shared memory space and gives it the key previously specified. shmat returns a pointer to that shared memory so we can modify it.

Then, the server splits up the data into segments depending on how many clients are going to call. Also depending on how many clients will call, the server sets up the first few bytes to be flags for other clients to relay that they are finished with their work.

My server program is also the program that ends up merging all the data into a sorted final array. I chose to do this because clients could then act just like forks and quit as soon as they finish sorting their portion of the array instead of potentially having to read off of the network twice (once to gather the portion of the array to sort, and the second to grab the rest of the data and merge).