Brandon Pollack

23754365

HW1

Written Questions

1. My algorithm switched to sequential when the list was divided into the same number of pieces as the PC has cores, this way, there is a physical core for every sort thread (otherwise I am only multithreading through timeshare, which just creates overhead through thread creation and shouldn’t run any faster) So listsize/arg is the max size a thread can be spawned to sort, otherwise the sequential algorithm is run.
2. This is data race free because now two threads ever access the same groupe of data (since each group is broken into halves of their data). The only time a data race would occur is if there was no join of the two children threads that are spawned by each thread, but that join is to ensure the sublists of sublists are merged before merging themselves.
3. I expect that the java sort is selected to be the fastest in place algorithm that can sort anything that implements comparable. There is likely quite a bit of overhead in the java algorithm since it supports any Comparable class, has many checks, try catches, and other safety features. Also, inplace sort may not be quite as fast as merge sort would be (which uses much more memory). Perhaps in place sorts are faster for much smaller lists, since not much extra memory is allocated by seqmergesort, and thus not quite being worth the system call (which is expensive), but pretty quickly it is probably going to overtake java’s sort. Creating a thread is expensive, so another order of magnitude higher is probably when parallelism will kick in and be faster than sequential.
4. The values are not what I expected, I thought Java would be slower, but It looks like Oracle has a few more tricks up their sleeve than I expected.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| size | Java sort | seq merge sort | parallel merge sort | |
| 500 | 79507 | 70693 | 3989024 |  |
| 1000 | 83974 | 143801 | 984337 |  |
| 2000 | 176643 | 627126 | 539830 |  |
| 4000 | 404722 | 826891 | 535181 |  |
| 8000 | 969364 | 816084 | 1547712 |  |
| 16000 | 1806338 | 1696464 | 646203 |  |
| 32000 | 3425468 | 3488676 | 786081 |  |
| 64000 | 6622072 | 6540754 | 2893425 |  |
| 128000 | 6248502 | 12324948 | 2025663 |  |
| 256000 | 13184983 | 29187205 | 3418163 |  |
| 512000 | 27449568 | 55340261 | 7051306 |  |
| 1024000 | 58455185 | 1.17E+08 | 19175222 |  |
| 2048000 | 1.21E+08 | 2.45E+08 | 41323618 |  |
| 4096000 | 2.56E+08 | 5.23E+08 | 78684230 |  |
| 8192000 | 5.36E+08 | 1.08E+09 | 1.64E+08 |  |

1. Parallelism could have been exploited in a number of places, the first place I can think of is to make the random number generator used thread safe, and generate the random list of longs in parallel. You shouldn’t run the tests simultaneously since you are trying to measure their effectiveness, but you could run the checks simultaneously to make sure they were sucessful, and check them with multithreading. A strategy for this would be to break the list up into arg chuncks, check them for being sorted, and then check the boundaries of each chuck of list (on an 8 core processor only 7 edges). I am not sure if this is possible, but all the points where memory is allocated might be done in parallel as well to make allocation quicker (though I don’t know if this would help much)