# Program Structures and Algorithms

## Spring 2023(SEC-01)

### Assignment 5

Name: Raja Shekar Reddy Siriganagari

NUID: 002762145

**Task:**

Your task is to implement a parallel sorting algorithm such that each partition of the array is sorted in parallel. You will consider two different schemes for deciding whether to sort in parallel.

1. A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.
2. Recursion depth or the number of available threads. Using this determination, you might decide on an ideal number (*t*) of separate threads (stick to powers of 2) and arrange for that number of partitions to be parallelized (by preventing recursion after the depth of *lg t* is reached).
3. An appropriate combination of these.

There is a *Main* class and the *ParSort* class in the *sort.par* package of the INFO6205 repository. The *Main* class can be used as is but the *ParSort* class needs to be implemented where you see "TODO..." [it turns out that these TODOs are already implemented].

Unless you have a good reason not to, you should just go along with the Java8-style future implementations provided for you in the class repository.

You must prepare a report that shows the results of your experiments and draws a conclusion (or more) about the efficacy of this method of parallelizing sort. Your experiments should involve sorting arrays of sufficient size for the parallel sort to make a difference. You should run with many different array sizes (they must be sufficiently large to make parallel sorting worthwhile, obviously) and different cutoff schemes.

**Observations:**

Metrics used:

1. Arrays of size to
2. For each size I have used a cut-off value of the range
3. For each cut-off value the *Parsort.sort* method is called by using different number of threads in the range
4. For each combination of these values the experiment is repeated 10 times and the total time is averaged.

Below are the timings for each array size:

**Table 1:** Parallel sort timings when array size =

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Cut-off value** | **Threads** | | | | | |
| **2** | **4** | **8** | **16** | **32** | **64** |
| 2048 | 241 | 212 | 189 | 162 | 142 | 169 |
| 4096 | 120 | 112 | 111 | 113 | 115 | 115 |
| 8192 | 106 | 111 | 106 | 107 | 110 | 104 |
| 16384 | 110 | 104 | 106 | 108 | 261 | 149 |
| 32768 | 108 | 102 | 106 | 110 | 111 | 108 |
| 65536 | 124 | 117 | 106 | 109 | 100 | 94 |
| 131072 | 116 | 106 | 111 | 108 | 88 | 90 |
| 262144 | 114 | 177 | 135 | 136 | 127 | 120 |
| 524288 | 179 | 257 | 165 | 128 | 101 | 99 |
| 1048576 | 152 | 117 | 90 | 93 | 102 | 90 |
| 2097152 | 112 | 108 | 104 | 101 | 105 | 109 |

**Table 2:** Parallel sort timings when array size =

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Cut-off value** | **Threads** | | | | | |
| **2** | **4** | **8** | **16** | **32** | **64** |
| 4096 | 271 | 269 | 271 | 274 | 298 | 532 |
| 8192 | 403 | 382 | 426 | 609 | 374 | 999 |
| 16384 | 420 | 288 | 329 | 405 | 481 | 434 |
| 32768 | 329 | 275 | 486 | 396 | 391 | 388 |
| 65536 | 282 | 320 | 293 | 246 | 244 | 229 |
| 131072 | 245 | 244 | 262 | 266 | 270 | 220 |
| 262144 | 274 | 396 | 309 | 266 | 195 | 189 |
| 524288 | 321 | 559 | 378 | 201 | 191 | 211 |
| 1048576 | 417 | 431 | 297 | 178 | 168 | 200 |
| 2097152 | 369 | 407 | 445 | 323 | 201 | 286 |
| 4194304 | 327 | 451 | 251 | 221 | 227 | 244 |

**Table 3:** Parallel sort timings when array size =

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Cut-off value** | **Threads** | | | | | |
| **2** | **4** | **8** | **16** | **32** | **64** |
| 8192 | 970 | 940 | 796 | 744 | 723 | 650 |
| 16384 | 614 | 553 | 566 | 632 | 621 | 624 |
| 32768 | 512 | 550 | 592 | 557 | 583 | 558 |
| 65536 | 497 | 562 | 546 | 522 | 529 | 497 |
| 131072 | 462 | 449 | 467 | 469 | 471 | 473 |
| 262144 | 488 | 485 | 473 | 484 | 415 | 417 |
| 524288 | 604 | 473 | 512 | 478 | 406 | 392 |
| 1048576 | 494 | 533 | 503 | 420 | 363 | 406 |
| 2097152 | 551 | 610 | 466 | 348 | 404 | 345 |
| 4194304 | 728 | 505 | 367 | 370 | 367 | 381 |
| 8388608 | 451 | 473 | 452 | 441 | 453 | 450 |

**Table 4:** Parallel sort timings when array size =

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Cut-off value | Threads | | | | | |
| **2** | **4** | **8** | **16** | **32** | **64** |
| 16384 | 2612 | 1197 | 1174 | 1189 | 1336 | 1247 |
| 32768 | 1144 | 1200 | 1110 | 1074 | 1087 | 1063 |
| 65536 | 1019 | 999 | 1019 | 1098 | 1098 | 1184 |
| 131072 | 880 | 891 | 999 | 896 | 872 | 1029 |
| 262144 | 1011 | 896 | 884 | 908 | 891 | 889 |
| 524288 | 1087 | 912 | 882 | 969 | 873 | 780 |
| 1048576 | 991 | 987 | 964 | 886 | 750 | 724 |
| 2097152 | 1026 | 1035 | 1005 | 847 | 722 | 729 |
| 4194304 | 1246 | 1185 | 934 | 694 | 691 | 690 |
| 8388608 | 1309 | 1047 | 736 | 736 | 734 | 737 |
| 16777216 | 966 | 983 | 982 | 969 | 984 | 984 |

**Graphs:**

**Conclusion:**

* On an average the optimal value of threads for different array sizes is between **32-64**.
* The optimal cut-off value varies in the range of  **to** .
* As the number of threads increase the available number of cores to run the threads decrease. As there is more context switching the wait time for the threads to run is more.
* With increase in optimal times for context switching by Intel processors the wait time has decreased even for a greater number of threads. The optimal thread count for my processor is found to be approximately **32.**
* The cut-off value also varies in the same pattern. As the cut-off value increases the number of times we partition the array decreases. That will in-turn cause the number of threads working on sorting to decrease. So, the optimal cap for the cut-off value in my experiments is found at in most cases.
* **Note: The runtimes vary each time the program is run depending on external factors like background tasks and CPU usage. More optimal conditions must be created to test parallel sort.**

**Screenshots:**

**Text

Description automatically generated**

**Text

Description automatically generated**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated