Goal

Write a multi-threaded program. A thread is an independent locus of control within a process (for the time being a process is a program in execution). Thus if you have several things to do and several CPUs, using multiple threads you can do them at the same time and thus speed things up!

Problem Statement

Many different sorting algorithms have been devised over the years. In addition to their complexity (e.g., \(O(N^2)\) versus \(O(N\log N)\)), they differ in other ways. One of mergesort’s advantages is that it is very easy to parallelize. One approach is to have different threads sort different chunks of the data. The sorted chunks can then be merged together.

Your task is to write a multi-threaded mergesort that takes as command line arguments an element count and a number of threads to use. Your program should create an array of element-count random integers and then sort it. The element count will be given as a power of two and you can assume that element count is evenly divisible by the number of threads. For example, the command `mergesort 10 4` would create an array of \(2^{10}\) elements and then sort it using 4 threads. Each thread would mergesort a quarter of the data and then the four sorted chunks would be merged together by the main thread. The only “output” of interest is the time taken (as printed by the code found in the repo). Finally, run your code twice in one of two different modes: checking-mode and performance-mode. In checking-mode, you should include memwatch to check for common memory allocation errors and also use the built in function `qsort` (quick-sort) to create a test oracle. For the latter `qsort` a copy of the original array and then compare it with the output of your mergesort. In performance-mode memwatch and `qsort` are not used so that the runtime of just merge sorting can be gathered.

What to hand in (Please no .docx files.)

1. As part of your design include a picture for `mergesort 3 4`. Be sure to identify which elements of the array get assigned to each thread? What does the array look like when all four threads are done. Repeat this for `mergesort 5 4` to help you derive the general equations for determining each thread’s range of the data.

2. A well-formatted 2-up printout of your source code. You must use `a2ps` after removing all the tabs from your code. Indent code 2 or 4 spaces at most.

3. A GitHub repo that includes (you must use these names as the grading script will assume their use!)
   - `README.md` (this should include your analysis, design, and test-plan),
   - your source code (at least `main.c`, `mergesort.c`, and `mergesort.h`), and
   - `Makefile` (where make will build your code in performance mode, and make checking will build your code using memwatch (reporting zero anomalies) and `qsort`).

Assignment Notes

• (on 4/6/18) I’ll pull your repo at 6am and look at your analysis answering any analysis questions you have included in README.md.
• (on 4/9/18) I’ll pull your repo at 6am and look at your design and plateau schedule, answering any questions you have included in README.md.

• I expect to pull your code, run make, and then run my test script.

• The output of git log will again factor into your grade.

• Here is the git classroom invitation https://classroom.github.com/a/fSGqMqsK

• The size of a statically allocated (temporary) array is effectively limited by the size of the stack. If you run into this problem, consider allocating large (temporary) arrays off the heap.

• While optional, the following is a more sophisticated approach, which you might implement and then empirically compare with the one used in the assignment to see if it is faster. For mergesort 10 8.
  1. Divide data into eight chunks ($c_1 \cdots c_8$).
  2. Create eight threads ($t_1 \cdots t_8$) to sort the eight chunks.
  3. After sorting $t_i$, $i = 2, 4, 6, 8$ can terminate.
  4. Each $t_i$, $i = 1, 3, 5, 7$ should wait for $t_{i+1}$ and then merge $c_i$ and $c_{i+1}$.
  5. When done, $t_3$ and $t_7$ can terminate.
  6. $t_1$ and $t_3$ should wait for $t_{i+2}$ and then merge the two corresponding chunks.
  7. When done $t_5$ can terminate.
  8. Finally, $t_1$ does the final merge!

• Initial grading rubric includes, but is not limited to
  - quality of the plateau schedule
  - quality of the test plan
  - quality of the source code, which includes but is not limited to
    - function headers (you should write these first!)
    - organization (e.g., lack of functions, bonus block opens, ....)
    - indent 2 or 4 spaces
    - no magic numbers
    - cloning rather than reuse
    - one function one thought!
    - lack of code proof reading (check out gv[1])
      - e.g., no function should span a page break
    - use of globals
    - code complies without generating any warnings
    - code passes your test cases
    - code passes my test cases
    - git was used correctly
    - memwatch is happy