Multi-Core Architecture and Systems (0368-4183) Lecturer: Adam Morrison

Home Assignment 0

Due date: **November 6th, 23:59** (submit via moodle)

Submission instructions

- **In this assignment, you can submit either in pairs or individually.** Even if working in a pair, each member should submit the assignment through the moodle.
- Submit a **ZIP** file containing your program; see detailed instruction below.
- A solution submitted t seconds late will have its grade multiplied by $1 - \left(\frac{t}{60 * 60 * 24 * 7}\right)^4$.

Integer sorting program

Implement a program that sorts a file containing unsigned 64-bit integers. Try to exploit parallelism in your sorting algorithm so that it runs faster if given more processors.

You can implement the program in any language **except Python**, which is too slow for our purposes. The program must follow the specification provided below and must build and run on the `rack-mad-01` machine in the TAU CS lab. (See instructions on the moodle for how to access this machine remotely from home.)

You can use any sorting algorithm whose sequential time complexity on an array with n random numbers is asymptotically better than $O(n^2)$. (So bubble sort, selection sort, and other such algorithms are not allowed.)

The goal of the assignment is to expose you to multi-core programming and the challenges of exploiting parallelism. As a result, it won't be graded based on performance, but on correctness and on seeing that you made an effort to benefit from parallelism.

1 Specification

1. The executable to run is named `parsort`. It accepts two command line arguments. The first argument specifies the number of processors to use. The second argument specifies the input file.
   Example usage: `parsort 16 input.txt`

2. The file to sort contains a sequence of 64-bit unsigned integers (delimited by new-line, i.e., one per line). The integers are specified in decimal representation and are guaranteed to be unique.

3. The program performs the following flow:

   (a) Open the file and read the integer sequence to memory.

   (b) Sort the integer sequence. Let T be the number (integer, not floating point) of **microseconds** ($\frac{1}{10^6}$ seconds) that the sorting took.

(c) Write the line "`A: T`" to standard output, where A is the sorting algorithm you used. For example, `QuickSort: 1234`.

(d) Write the sorted sequence to standard output, in the same format as the input.

Important: Don't include I/O time (reading/writing the sequence) in the printed time.

2 Useful tips

• You are provided a ZIP that contains an example sequence in `input.txt` and the sorted result in `sorted.txt`. You can use this to check your sorting algorithm. Note that the reference output doesn't include the first line with the time report, only the sorted sequence.
• Make sure you print time in **microseconds**. You can verify this by doing a dummy sort that simply sleeps for one second, and seeing that your program reports the time as ≈ 1000000.
• **Important tip for Java programs:** The Java virtual machine compiles your Java code into machine code using a just-in-time scheme, i.e., while running the code. This means that the first time your code runs, it will be slower, since the execution will be delayed by compilation. It is therefore strongly recommended to run the sorting algorithm twice, and to print the running time of only the second iteration. This way, you will be reporting the time of machine-compiled, optimized code. (Of course, be careful that the second iteration also sorts the original input and not the already-sorted output of the first iteration!)

3 Programming tips

3.1 Java

Java has a built-in fork/join parallel programming framework. You can find tutorials for it online (for example, `https://www.baeldung.com/java-fork-join`).

3.2 C/C++

The GCC version on the server is 7.5.0, which supports the C11 standard but not later standards such as C17.

Be sure to use increase the compiler's optimization level by using the `-O3` flag. The default is to not have any optimizations, which produces very inefficient code.

3.2.1 Parallelism frameworks

It is perfectly fine (and possible) to solve the assignment without using a parallel programming framework. If you do wish to use a framework, below is information about a couple.

OpenMP GCC (and G++) support the OpenMP parallel programming framework. You can find tutorials for it online (for example, `https://hpc-tutorials.llnl.gov/openmp/`). To compile a program with OpenMP support, add the `-fopenmp` flag to the compilation command line.

Cilk: The CilkPlus fork/join framework is installed on the server. Unfortunately, Cilk requires a special compiler, and the one available is based on GCC 4.9, which only supports C11. To compile a C++ program with Cilk keywords, use the command line:

```
/specific/disk1/home/mad/gcc-cilk/bin/g++ prog.cpp -fcilkplus -lcilkrts
```

You can see a couple of examples Cilk programs in the following directory:

```
/specific/disk1/home/mad/cilkutil/examples
```

If you use one of the above frameworks, your submission's scripts must be able to compile and run your program from a clean environment.

4 Submission instructions

Submit a ZIP file containing your source code and a script named `build`. Unpacking your ZIP and then running `build` should build your program. Running `parsort` should then execute the program. Note that `parsort` can also be a script included in the ZIP and does not have to be created by `build`; see the examples below:

- For C programs, `build` should compile the program and name the resulting binary `parsort`.
- For Java programs, `build` should compile the program, with `parsort` being a script you include that sets up the environment variables needed to run the JVM and then runs the program.