TDDD56 Multicore and GPU computing
Lab 3: Parallel sorting

Nicolas Melot
nicolas.melot@liu.se

November 23, 2015

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOGN)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

Figure 1: Creative Commons Attribution-NonCommercial 2.5 License. http://xkcd.com/1185/

1 Introduction

Sorting in one of in the most important routines, used in many programs and run frequently on a computer to perform everyday tasks. Because of the very intensive use of sorting, its performance has a great chance to influence the performance of other programs using it and the overall performance of a complete system. Consequently it is a good idea to take profit of multicore computation capabilities to accelerate sorting operations through parallel sorting algorithms. Compared to many other calculations, sorting requires much fewer calculations, almost comparisons only, but a lot of memory operations to swap values. Because of this property, one can expect sorting algorithms and their parallel equivalent to be very sensitive to data locality issues.

This laboratory consists in the design, implementation and assessment of one parallel sorting algorithm of your choice, using the technique of your choice (pthreads, openmp, etc.) and using the Drake framework introduced in Lesson 1. A brief report should describe the programming experience.

2 Getting started

2.1 Installation

Fetch the lab 3 skeleton source files from the CPU lab page[1] and extract them to your personal storage space.
Students in group A (Southfork): install the R packages ggplot2 and plyr by downloading and running this script[2].
Load modules to set up your environment.

In Southfork (group A): type the following in a terminal after each login

```
:~\$ setenv MODULEPATH /edu/nicme26/tddd56:$MODULEPATH; module add
    tddd56
```

In Konrad Zuse (groups B and C): type the following in a terminal

```
:~\$ module add ~TDDD56/module
:~\$ module initadd ~TDDD56/module
```

3 More on lab skeleton

The lab work consists in the implementation of two versions of parallel sort. One approach should use one of the classic parallelization technique such as pthread or openM (we call it *classic* parallel sort) uses implementation of function sort(**int**$*$, size_t) in src/sort.cpp, compiled into *src/parallel-sort*) with flag NB_THREADS. The second approach should use Drake to exploit pipelined parallelism ("*drake*" sort). This second approach implements parallel mergesort as a streaming application where all tasks merge two sorted input data streams into one bigger data stream. A merging task implementation stub is available in *src/merge.c*. Figure 2 represents the initial task graph provided in the lab skeleton.

Streaming tasks have a life cycle in 5 phases, spanning from the creation to the destruction of the associated pipeline, including when the pipeline runs:

[1]`http://www.ida.liu.se/~nicme26/tddd56.en.shtml`
[2]`http://www.ida.liu.se/~nicme26/tddd56/southfork_setup`

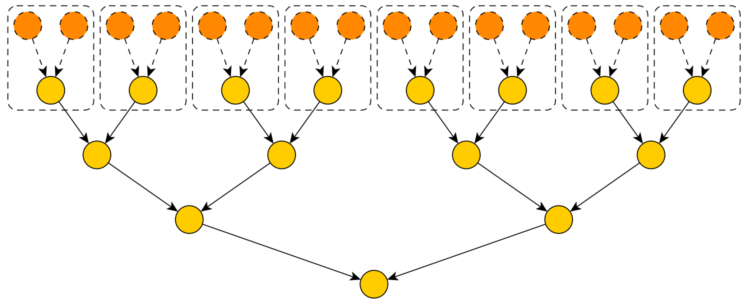


Figure 2: Graphic representation of a 4-levels pipelined merging tree as given in the lab skeleton. Leaf tasks create presorts tasks (in orange), that are tied to their corresponding leaf task.

1. *Init* phase: a task is initialized when the pipeline is created. This is an occasion to do "heavy" initial work such as loading input data or allocating memory. This step is implemented in drake_init(task_t $*$task, **void**$*$ args) available in *src/merge.c*. In the skeleton, leaf tasks create virtual presort tasks (presort tasks are shown in orange in Fig. 2) and make them to load a portion of input data. In the same phase, the root task allocates a sufficiently big output fifo buffer.

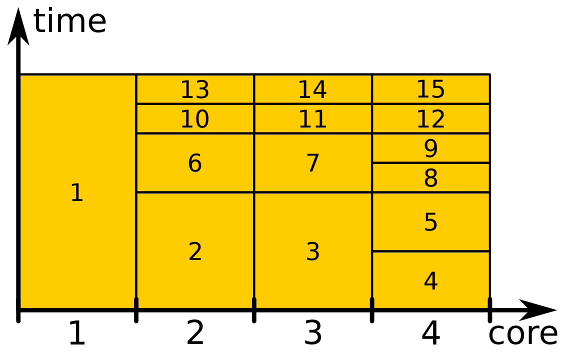
2. *Start* phase: when the pipeline starts, all tasks are in *start* phase and Drake runs drake_start(task_t $*$task) the first time they are scheduled. This phase is ideal to do work that is necessary only once but whose work is relevant to the global work to be achieved in the streaming application. In our skeleton, a leaf task also presorts the data of both its associated *presort* virtual tasks. If drake_start(task_t $*$task) returns 0, then drake runs it again for the corresponding task next time it is scheduled. Otherwise, the task enters the *run* phase.

3. *Run* phase: When a task in *run* phase is scheduled, Drake runs drake_run(task_t $*$task) for it. The run phase performs the main work associated to the task, i.e., merging two input data streams. When drake_run(task_t $*$task) returns 0, then Drake considers the tasks is done working and the task enters the *kill* phase.

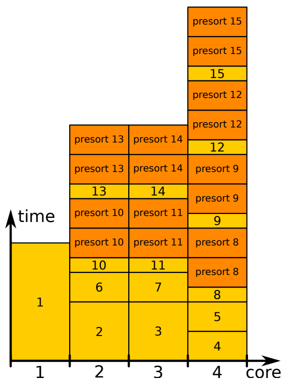
4. *Kill* phase: Drake runs drake_kill(task_t $*$task) for each task in phase *kill* that is scheduled. When all tasks entered the *kill* phase, then all tasks switch to *destroy phase* and the pipeline is stopped and destroyed.

5. *Destroy* phase: When a pipeline is destroyed, Drake runs drake_destroy(task_t $*$task) for all tasks. In the skeleton, this makes the root task to check if its output buffers matches the reference sorted array.

The performance of a streaming application depends on the quality of its schedule. The performance is best if the workload difference between cores is minimized. LPT (Longest Processing Time) is a fast scheduling algorithm that computes solution at worst $4/3$ worse than the best possible solution. It works by scheduling first tasks that have a larger amount of work to do (in the lab skeleton, this is described in the



(a) LPT Schedule without tasks



(b) LPT Schedule with taking presort tasks (presort time is not representative)

Figure 3: Caption

taskgraph description file, one of *src/merge*.graphml*, depending on the version compiled), and assign it to the core having received the least load at a given time. For example, scheduling the merge streaming application to 4 cores would result in the the schedule shown in Fig. 3. Since task 1 merges the whole data, its merging task performs twice the work of task 2 or task 3 (each merge half of the data), and so on. Here, LPT finds a perfect schedule without taking presorting into account (Fig. 3(a)). However, the same schedule is actually not very well balanced taking presorting into account (Fig. 3(b)). One solution could consist in updating the work of each task in the taskgraph description and compute a new schedule using LPT, or to elaborate a schedule manually.

3.1 Source files

The section provides useful information to manipulate the lab skeleton such as its file structure, compilation options or useful informations on Drake.

3.1.1 src/sort.cpp:

Your implementation of parallel sort, also used by Drake with NB_THREADS=0 (See Sec. 3.2) for its initial sorting phase. The skeleton calls function sort(**int**∗ array, size_t size), where array is a int pointer to the data to be sorted and size is the number of **int** (4 bytes) elements accessible from array. When the function terminates, the skeleton must find the same size elements as when the function started, arranged in increasing manner.

3.1.2 src/merge.c:

Implementation of a merging task in Drake. You only need to implement the drake_run(task_t∗) function. Keep in mind that this is a streaming task and that it can never hold the entire

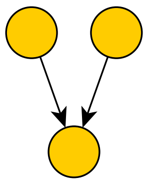


Figure 4: Graphic representation of a 2-levels pipelind merging tree as given in the lab skeleton

data to merge at a given time. This has practical consequences on how to consider if elements of input sequences are the last ones and how to merge them properly.

3.1.3 src/Makefile.fillmein:

Contains compilation settings, such as default number of threads, compilation and linking flags, list of source files to compile. This file also contains the definition of Drake's default streaming application description file, the filename of its XML schedule and its execution platform description.

3.1.4 src/platform-X.dat:

Drake platform description. X stands for the corresponding value of NB_THREADS. It contains the number of cores of the application and the frequencies at which a core can run. Only the number of cores is important. All platform descriptions in the initial lab skeleton denote single core platforms.

3.1.5 src/merge-X.graphml:

Structure of the merging pipeline for X cores where X equals NB_THREADS. The pipeline doesn't need to change with number of cores, but you may gain performance by tuning it for each case. The task skeleton loads data based on the number of tasks having no input link, which is computed assuming a balanced binary tree. The skeleton will fail if the pipeline is not a balanced binary tree. All taskgraph descriptions in the skeleton are 4 levels balanced merging trees. Figure 4 shows a 2 levels merging tree and Listing 3.1.5 shows its equivalent XML representation. The root task (*task_1*) is annoted with twice the workload of tasks 2 and 3. The description also give information on the maximum number of cores the scheduler is allowed to allocate cores to (*max_width*). Note that Drake doesn't support yet parallel tasks. Other fields include the file containing the source code of teh task (*module*) and the parallel efficiency of the task (*efficiency*). Here, all task would suffer from a high penalty if they ran on more than one core. Finally, the file shows two edges from tasks 2 and 3 toward task 1. This file is useful to a scheduler to generate better schedules and to Drake to allocate communication channels between tasks while initializing a pipeline.

```
<data key="g_name">pipelined_merge</data>
        <data key="g_deadline">0</data>
        <node id="n0">
                <data key="v_name">task_1</data>
                <data key="v_module">merge</data>
                <data key="v_workload">32</data>
                <data key="v_max_width">1</data>
                <data key="v_efficiency">exprtk: ... </data>
        </node>
        <node id="n1">
                <data key="v_name">task_2</data>
                <data key="v_module">merge</data>
                <data key="v_workload">16</data>
                <data key="v_max_width">1</data>
                <data key="v_efficiency">exprtk: ... </data>
        </node>
        <node id="n2">
                <data key="v_name">task_3</data>
                <data key="v_module">merge</data>
                <data key="v_workload">16</data>
                <data key="v_max_width">1</data>
                <data key="v_efficiency">exprtk: ... </data>
        </node>
        <edge source="n2" target="n0"/>
        <edge source="n1" target="n0"/>
</graph>
```

In the lab skeleton, each leaf task (in our example, tasks 2 and 3) are added two "virtual" tasks at initialization time, then load and give them a portion of the data to sort. When the pipeline starts, the virtual task runs your implementation of sequential quicksort on the input buffer it received, then it writes the sorted array to the channels of the merging task it is attached to. Because initial sorting tasks do not appear in the pipeline descriptions provided, an automated scheduler lacks data to compute the best schedule and you need to tune schedules by hand.

3.1.6 src/schedule-X.xml:

Schedule for the merging pipeline for X cores (where X equals NB_THREADS) in XML format. Due to some limitations in its implementation, Drake will silently fail if a core receives no task to run. The initial skeleton schedules run all tasks on a single core. You can generate a schedule for an arbitrary number of cores using the skeleton-provided implementation of *LPT* (see Sec. 6), although you may get better performance by hand-tuning schedules. Listing **??** is a schedule for the pipeline shown in Fig. 4 and Listing 3.1.5.

```
<schedule name="lpt" appname="pipelined_merge">
        <core coreid="1">
                <task name="task_1" start="0" frequency="1" width="1" workload="32"/>
        </core>
        <core coreid="2">
                <task name="task_3" start="0" frequency="1" width="1" workload="16"/>
                <task name="task_2" start="16" frequency="1" width="1" workload="16"/>
        </core>
</schedule>
```

3.1.7 variables:

This files holds the batch parameters for Freja. By default, it makes Freja to compile binaries for 1 to 6 threads. If you work in Southfork, you can modify the property nb_thread to vary from 1 to 4 instead.

3.2 Compiling and running

The skeleton can be compiled in two different modes you can enable or disable by passing variables when calling make.

3.2.1 NB_THREADS:

Call make with NB_THREADS=n (make NB_THREADS=n) where n is a non-negative integer. It instructs make to compile a multi-threaded version running n threads; $n = 1$ compiles a sequential version. The true interest of this lab lies with $n > 1$.

3.2.2 Debugging:

This mode disables any compiler optimization and set the flag DEBUG. This enables the use of a debugger and actives traces upon a failure (see Sec. 6). You can enable it by calling make as follow:

```
:~\$ make CONFIG=DEBUG
```

Make sure your sources are clean before you compile in debug mode (make clean). Keep in min that executables compiles in debug mode are significantly slower than in release mode. Compiling the Drake implementation with this command will make use of files src/merge.dat and platform.dat and generate a schedule using LPT; see Sec. 6.3 on how to generate input data.

You can try if your implementation works correctly by making it to sort some input data (see Sec. 6.2 on how to generate input data). The program compiled runs your sorting implementation, displays its execution time in milliseconds, sorts again the same input with a trusted method (gcc's qsort) and reports any inconsistency through stderr. It yields an error if the array sorted with your implementation has a different number of elements that the one sorted with the trusted sorting implementation, or if values from both arrays (sorted with your implementation and sorted with the trusted implementation) at the same index are different. Here is a successful run that you can obtain with a fresh copy of the lab skeleton:

```
:~\$ src/parallel−sort /path/to/file.bin
224.900959
[utils.c:check_sorted:28] Input length OK.
[utils.c:check_sorted:46] No difference between sorted and qsorted arrays.
```

Here is the output of an implementation that produce an empty sorted output array. The program complains that the final sorted array contains 0 elements whereas is should contain 1048576 of them.

```
:~\$ src/drake−sort /path/to/file.bin
203.760067
[utils.c:check_sorted:23] Different length of sorted (0) and reference arrays (1048576).
```

Finally, this is the program output when the size of the sorted array is correct, but when two elements are different from the ones in the reference array sorted with the trusted implementation.

```
:~\$ src/parallel−sort /path/to/file.bin
225.332430
[utils.c:check_sorted:28] Input length OK.
[utils.c:check_sorted:40] Different value at index 56. Got 14, expected 97.
[utils.c:check_sorted:40] Different value at index 772. Got 123, expected 456.
```

3.2.3 Measuring:

This builds binaries without any debugging information and all compiler optimizations enabled. This produces significantly faster binaries. You can compile a binary by yourself with the command make. This generates src/drake−sort and src/parallel−sort with settings defined in src/Makefile.fillmein. You can compile a set of binaries for NB_THREADS between 1 an 6 (4 if you work in Southfork) with Freja (freja compile). After having used Freja to compile binaries, use freja run benchmark to produce performance data. If you run the experiment again, make sure you remove directory benchmark before running Freja again. Finally, use ./drawplots to generate figures from the data.

4 Before the lab

Before coming to the lab, we recommend you do the following preparatory work:

- For each sorting algorithms reviewed during lectures, imagine the most simple way to parallelize it using 2 threads, then using 3 threads.
- Identify the factors that can turn to performance bottlenecks in the parallel sort you imagine. Imagine a way to improve these bottlenecks
  Hint: think about the classic three phases: partition, recursion and merge. Pick techniques studied in lectures and previous lab work. Switch to different sorting techniques while the work progress, such as when no more thread are available

to parallelize the recursive work, input set gets small enough to fit in the cache, etc.

- Given the difficulties and solutions you have carefully investigated earlier, choose one sorting algorithm and implement it. Make sure the sorting time for 3 threads is approximately mid-way between sorting time using 2 cores and sorting time using 4 cores.
- Think of how you can lay out and schedule a merging tree for 1 to 6 cores (4 cores for students in group A), so you can maximize parallelism and reduce unnecessary work. It must be a balanced binary tree, but how many levels should it contain?
- Review the flow of data through the merging pipeline in Drake and how merging tasks are started and how they stop. How would you design a merge C function when all data is available and accessible at the time the function is called? What would you change if the whole data is only accessible through successive calls to the same merge function?
- Implement your algorithm ahead of time you chose so you can measure its performance during the lab session.
- Try using the built-in C++ sorting function instead of your own implementation (provided in the skeleton). Can you take profit of it in your own parallel sorting implementation? in Drake?

You are strongly encouraged to carefully follow these steps as they might allow you to find unexpected difficulties and prevent you from spending a lot of time and energy on issues that could have been otherwise avoided.

5 Lab demo

The lab work consists in implementing parallel sorting in 2 variants: one using classic parallelization tools such as pthread or openMP. The second uses Drake to implement a pipelined mergesort. The demo should include performance measurements in the form of plots so that to make comparisons easy.
Demonstrate the following elements to your lab assistant:

- Describe the sequential sorting algorithm you chose to implement. This algorithm would sort 4 bytes integers in increasing order.
- Describe the basic parallelization strategy you implemented.
- Show the limitations of your parallel implementation. In what situation(s) does it struggle to take profit of all cores available?
- Describe how you made your implementation to run fast on 3 cores.

- Show the sorting time and/or the speedup of your implementation. The performance with 3 cores should be approximately mid-way between performance with 2 and 4 cores.
- Show the pipeline structure you chose for 1 to 6 cores (4 cores for students in Southfork).
- Show the schedule you used for 1 to 6 cores (4 cores for students in Southfork).

6 Helpers

This section describes tools at your disposal to help you solve the laboratory work. This skeleton includes a backtracing tool, an input data generators and a tool to show the content of an input file.

6.1 Debugging helpers

If you compile in debug mode (make CONFIG=DEBUG), the code catches signals when the program is interrupted, either by a signal (ctrl-C) or by an error (segmentation fault, floating-point exception). The program under debugging outputs debugging information a terminates. backtrace takes as parameter the binary executable that crashed and the debugging information (only number between square brackets is useful), and output the call stack at the time the signal was caught. Here is an example:

```
:~\$ src/drake−sort /.../input−1024−i1.bin
[utils.c:bt_sighandler:68] Caught signal 11
[utils.c:bt_sighandler:93] Caught segmentation fault
[utils.c:bt_sighandler:103] Obtained 7 stack frames.
[utils.c:bt_sighandler:104] src/drake−sort() [0x4026e5] /lib/...(...) [0x7ff7f990b340]
      src/drake−sort() [0x404931] /home/...(...) [0x7ff7fa671b9d] /home/... [0
      x7ff7fa8893ff] /lib/... [0x7ff7f9903182] ... [0x7ff7f963047d]
Aborted (core dumped)
:~\$ backtrace src/drake−sort "rt()␣[0x4026e5]␣/lib/...(...)␣[0x7ff7f990b340]␣src/
      drake−sort()␣[0x404931]␣/home/...(...)␣[0x7ff7fa671b9d]␣/home/...␣[0
      x7ff7fa8893ff]␣/lib/...␣[0x7ff7f9903182]␣...␣[0x7ff7f963047d]␣"
/.../src/utils.c:64
??:0
/.../src/merge.c:193
??:0
??:0
??:0
??:0
```

where merge.c is:

```
192  int *val = 0;
193  *val = *parent;
```

and obviously generates a segmentation fault.

6.2 Manipulate input data to sort

After compilation, the binary src/generate can generate data to be sorted of arbitrary size, in one of four pattern:

- Increasing: Generate an increasing sequence of n integers ranging from 0 to $n-1$. Use the command:

  ```
  :~\$ src/generate --increasing --output /path/to/file.bin --size n
  ```

- Decreasing: Generate an decreasing sequence of n integers ranging from $n-1$ to 0. Use the command:

  ```
  :~\$ src/generate --decreasing --output /path/to/file.bin --size n
  ```

- Constant: Generate an constant sequence of n integers of value v. Use the command:

  ```
  :~\$ src/generate --constant --value v --output /path/to/file.bin --size n
  ```

- Random: Generate an random sequence of n integers with a uniform distribution. Use the command:

  ```
  :~\$ src/generate --uniform-random --output /path/to/file.bin --size n
  ```

You can use dump to inspect the content of an input file. For example:

```
:~\$ dump /path/to/file.bin
0 1 2 3 4 5 6 7 8 9
:~\$ cat /path/to/file.bin | dump
0 1 2 3 4 5 6 7 8 9
```

If you lack storage space to generate big input sets, you can generate and read input files in where /scratch/$USER in Konrad Zuse or in /tmp/$USER in Southfork. However, be aware that these files may not be available remotely and might be deleted between two lab sessions.

6.3 Generate schedules

You can generate automatically XML schedules for a given taskgraph and platform descriptions using an implementation of *LPT* provided in the skeleton. You can redirect standard output to the schedule file you want to generate. Below is an example of the

```
:~\$ lpt --taskgraph /path/to/taskgraph.graphml --platform /path/to/platform.dat
    --schedule-output /path/to/schedule.xml
```

7 Investigating further (optional)

We implement a course challenge where participants are invited to implement the best parallel sorting algorithm. If you want to participate, send an email to Nicolas Melot and give him a group name, your lab room (Southfork or Konrad Zuse) as well as the complete path where the source files can be found; make sure that this directory is publicly readable by everyone (run "*chmod 755 .*" from this directory) and everyone can execute all directories in which the source are stored (run "*chmod 711 every/step/to/your/source/once/per/successive/folder*"). The source files are fetched during the night once a day to be evaluated as part of the challenge and results are shown on the screen at the corner of the Konrad Zuse room or at http://www-und.ida.liu.se/∼nicme26.

The test consists in sorting in ascending order several sets of 100.000 or 10.000.000 integers. The sets may be random, already sorted in ascending order and in descending order or constant. Three different random input sets of each patterns are sorted 5 times each. The final score is computed from the average sorting time. You can find in the lab source files the complete measurement scripts as used when measuring each group's code's performance. The assessment assumes that running *make* produces the fastest, multi-threaded binary executable code to the file *sort*.

In December, the group having the best results will be awarded as best parallel programmers for TDDD56 HT2015 and receive a cinema ticket for the movie of their choice. They will have an opportunity to write a short description of their parallel implementation, which will be included in the result page for this course session. This page can later be given as a reference for later applications to job or positions.

8 Lab rooms

During a lab session, you will be the only one using the computers. Outside the lab sessions, the room is open and accessible from 8:00 to 17:00 every day, except when other courses are taught. The computers in this room offer the following resources: During the laboratory sessions, you have priority in rooms "Southfork" and "Konrad Zuse" in the B building. See below for information about these rooms:

8.1 Konrad Zuse (IDA)

- Intel®Xeon™ X5660[3]
  - 6 cores
  - 2.80GHz
- 6 GiB RAM
- OS: Debian Squeeze

[3]http://ark.intel.com/products/47921/Intel-Xeon-Processor-X5660-(12M-Cache-2_80-GHz-6_40-GTs-Intel-QPI)

During lab sessions, you are guaranteed to be alone using one computer at a time. Note that Konrad Zuse is not accessible outside the lab sessions; you can nevertheless connect to one computer through ssh at *ssh <ida_student_id>@li21-<1..8>.ida.liu.se* using your *IDA* student id.

8.2 Southfork (ISY)

- Intel®Core™ 2 Quad CPU Q9550[4]
  - 4 cores
  - 2.80GHz
- 4 GiB RAM
- OS: CentOS 6 i386

Southfork is open and accessible from 8:00 to 17:00 every day, except when other courses are taught. You can also remotely connect to *ssh <isy_student_id>ixtab.edu.isy.liu.se*, using your *ISY* student id.

[4]http://ark.intel.com/products/33924/Intel-Core2-Quad-Processor-Q9550-(12M-Cache-2_83-GHz-1333-MHz-FSB)