TDDD56 Multicore and GPU computing
Lab 3: Parallel sorting

Nicolas Melot
nicolas.melot@liu.se

1st November 2012

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 algorithms of your choice.

2 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 Suze” in the B building. See below for information about these rooms:

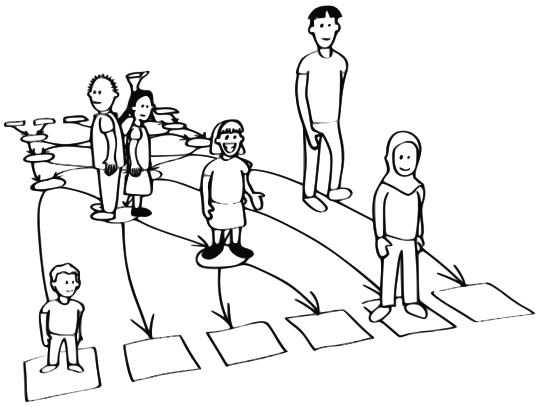

Figure 1: Sorting numbers through a sorting network. (dullhunk, http://www.flickr.com/photos/dullhunk/, CC BY 2.0)

2.1 Konrad Suze (IDA)

- Intel®Xeon™ X5660[1]
  - 6 cores
  - 2.80GHz
- 6 GiB RAM
- OS: Debian Squeeze

During lab sessions, you are guaranteed to be alone using one computer at a time. Note that Konrad Suze 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. You can use Octave to process your measurements and generate result graphs (see section 3).

2.2 Southfork (ISY)

- Intel®Core™ 2 Quad CPU Q9550[2]
  - 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. Southfork does not provide Octave to generate graphs but Matlab is available to run alternative scripts (see section 3).

3 Installation

Fetch the lab 1 skeleton source files from the CPU lab page *http://www.ida.liu.se/~nicme26/tddd56.en.shtml* and extract them to your personal storage space. Copy simple_quicksort.o.{konrad|southfork|ixtab} (choose the one matching the room your work in, or ixtab if you connect remotly to ISY server) to simple_quicksort.o. You also need performance measurement and processing scripts available at http://www.ida.liu.se/~nicme26/measuring.en.shtml. Fetch the script source files and copy the content of the directory "Octave" (or "Matlab" if you work in Southfork) to the lab source directory.

Warning: In Southfork, make sure you start bash and define the global variable *input_dir* to */tmp/$USER before* you run the measurement scripts or they will fail as they try to store input sets in /scratch (type bash, *then* type *export input_dir=/tmp/$USER*).

4 Before the lab

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

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

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

```
# Generate 7 values between 1 and 10 in an increasing
# order, and store it to the file inc-7.txt
nicme26@astmatix:~$ bash inc 1 10 7 > inc-7.txt
# Generate 12 values between 100 and 44 in an decreasing
# order, and store it to the file dec-12.txt
nicme26@astmatix:~$ bash dec 44 100 12 > dec-12.txt
# Generate 24 values between 100 and 200 in a random
# order, and store it to the file rand-24.txt
nicme26@astmatix:~$ bash rand 100 200 24 > rand-24.txt
# Generate 50 times the value 154, and store it to the
# file cons-50.txt
nicme26@astmatix:~$ bash cons 154 154 50 > cons-50.txt
```

Figure 2: How to generate input set files of various size and pattern.

5 During the lab

Take profit of the exclusive access you have to the computer you use in the lab session to measure the performance of you parallel sorting algorithm along different situations: **Warning**: In Southfork, make sure you start bash and define the global variable *input_dir* to */tmp/$USER before* you run the measurement scripts or they will fail as they try to store input sets in /scratch (type bash, *then* type *export input_dir=/tmp/$USER*).

6 Lab demo

Demonstrate to your lab assistant the following elements:

1. Describe the sequential sorting algorithm you chose to implement. This algorithm would sort 4 bytes integers in increasing order.
2. Describe the basic parallelization strategy you implemented
3. Describe all other optimizations you wrote such as load-balancing or the use special instructions (fetch-and-increment, compare-and-swap, etc.)
4. Show the time required to sort 100.000.000 4 bytes integers. The sequential version should sort it in around 6 seconds and execution time of the parallel version should be higher or equal the sequential time divided by number of threads, but lower than sequential time divided by number of threads - 1. In other word, if t_{seq} is execution time for your sequential version and n the number of threads in any parallel version, we must have $t_{seq} \leq \sim 6$ seconds, $t_{n=1} \geq \sim t_{seq}$, $t_n \geq \frac{t_{seq}}{n}$ and $t_n < \frac{t_{seq}}{n-1}$.

7 Helpers

A set of scripts is also available to generate files of increasing, decreasing, constant and random values, that can be used as input; see Fig.2 for an example on how use them. If you lack storage space to generate big enough input sets, you can generate and read input files in /scratch/$USER in Konrad Suze or in /tmp in Southfork. However, be aware that these files may not be available remotely and might be deleted between two lab sessions.

The lab source code provides the complete measurement scripts and code to check the correctness and measure the performance of your implementation. Run *bash start*

compile to compile several suggested variants, then *bash start run <name for experiment>* to run them and measure their performance (note that you must name your experiment). In order to spare your storage place, input data sets are generated in /tmp and might have to be generated again at the following lab session. Finally, run the script *plot_data.m* (*octave plot_data.m* or start Matlab and run the script plot_data.m) to generate graphs showing the behavior of your implementation. Modify the files *compile*, *run*, *variables* and *Makefile* at wish to fit further experiments you may need. You can read documentation about measuring performance and using the script set at the page *http://www.ida.liu.se/~nicme26/measuring.en.shtml*. Alternatively, feel free to use any other mean of measuring performance but in any case, make sure you can explain the measurement process and the numbers you show.

8 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 Suze) 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 Suze room or at http://www-und.ida.liu.se/~nicme26. All source files and measurement scripts are checked for correctness before any measurement is run in order to ensure fairness. If any source file is invalid then the results are not taken into account. You may modify only *sort.c* and *Makefile* and you are not allowed to use *qsort()*.

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 (coefficient 2) as well as the standard deviation across one the same input set (coefficient 2) for a given size, and the standard deviation across different input patterns of the same size (coefficient 1). 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*. Run *bash assess* to individually assess your results and check if your code is correct and will not be rejected. If octave is missing then *bash assess* can only check your code and it cannot compute your score.

On December 9^{th}, the group having the best results will be awarded as best parallel programmers for TDDD56 HT2012 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.