CSC – Software Design in the Many-core Era

A. Gheata, S. Hageboeck

Introduction

Welcome to the exercises section of *Software Design in the Many-core Era*! We use the standard CSC machines with AlmaLinux 9 to work on the problems.

Please note that it is likely that you will not be able to finish all the exercises in the allocated time. This is not a problem, as they are designed to also include challenges for people who already know a part of the subject. You can leave aside challenging tasks during a first pass if you wish, and you can come back to them later. In any case, you are very welcome to ask the tutors for hints and additional explanations during and after the exercises session. Remember that this is not a competition: these exercises are designed to help you assimilate the concepts to which you were exposed during the lectures, so go at the pace that will allow you to learn the most.

Getting the code Let's now fetch the exercises from the CSC Indico agenda. Please follow the CSC instructions to establish an ssh connection to the CSC machine that was assigned to you. If you have not done this before, you can use the ssh example below, which executes a proxy jump (`-J`) through `lxtunnel.cern.ch`, which might be necessary depending on firewall settings.

```
ssh -J <cern_username>@lxtunnel.cern.ch <cern_username>@XXXX.cern.ch
wget -O ManyCoreExercises.tar.gz https://cern.ch/bw8vd
tar -xf ManyCoreExercises.tar.gz
cd ManyCoreExercises
```

`ManyCoreExercises` contains one subdirectory for every exercise. If applicable, these come also with the solution. Try to make as much progress as possible – and do ask questions to the tutors – before looking at the solutions.

Given the central importance of compilers in the software development process, we decided to let you see and execute the compilation commands yourself. Try to craft the necessary commands yourself, but if in doubt, they can be found on the top of the source files as a comment.

Setting up the environment on non-CSC hosts The CSC machines with Alma9 contain all the software we need. However, if you want to repeat some exercises on a host that doesn't have all the necessary software installed, or if you want a more modern compiler/gdb with nicer error messages, you can set up an LCG software stack from cvmfs:

```
source setupScriptCvmfs.sh
```

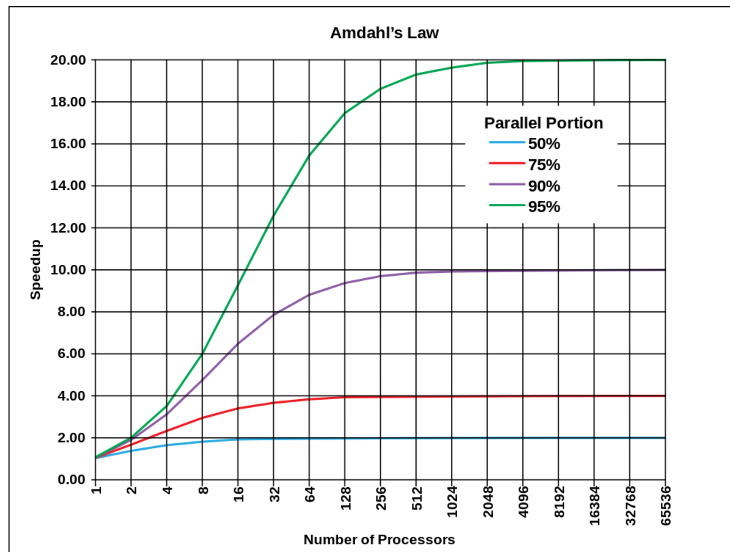


Figure 1: Amdahl's law for different values of P (from Wikipedia)

1 Amdahl's Law

We will start with some warm up – no need to code in this exercise. Let's get practical experience with one of the most relevant concepts in the field of parallel programming: Amdahl's law. As a reminder, see again figure 1 and equation 1:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

where S represents the speedup, N is the number of CPUs available, and P the portion of the program that can run in parallel.

Suppose now to be in charge of the hardware resources of a small manufacturing company. The main quad-core web-server works OK, but the performance of your web application running on it is suboptimal. 75% of the total work done by the program is spent in I/O, which runs purely sequentially. The rest of the operations scale well with the number of available cores. What is the gain you expect moving to an eight-core server? And to a hypothetical machine with 200 cores?

Let's say that the price of server is 2000 Euros. You can decide to pay the same amount of money to a consultant who is expert in software performance and parallel software design. He or she is able, evolving the present design, to reduce the serial part to 70% of the overall time. What would be the most profitable solution? Why?

2 The Map-Reduce Pattern

Let's get hands-on experience with the Map-Reduce pattern in Spark. We will run this on CERN's "Service for web-based analysis" SWAN. You will find the

instructions about how to proceed in the notebook itself.

- If you have never used SWAN before, you might need to create a cernbox account (CERN cloud storage). Go to `https://cernbox.cern.ch` and log in, and your cernbox will be created.
- *Afterwards*, go to SWAN: `https://swan.cern.ch`. The default configuration is fine for this exercise.
- Click on the little cloud icon on the top right, and clone the following repository:

  `https://github.com/hageboeck/CSC_Spark_Exercise.git`

  - If you started the jupyterlab interface in SWAN, use the little git symbol on the left to clone the repository.

3 Modern C++ syntax

In this and the following exercises we will develop some code based on examples reviewed in the lectures. In most of the cases, the compilation command is given at the top of the example code. Solutions to the exercises are given in the Solutions folder. Resist to the temptation to look at it too early. ;-)

C++ provides a few features that make the programmers' life much easier, and we will have a look at three of these features. You might wonder why we call a more than 10-year old standard "modern" C++. This is because many of the codes in HEP were started before this evolution of C++, so you will encounter code that doesn't use the idioms, and it might be a good idea to modernise where possible.

3.1 The auto keyword and range-based loops

In the directory of this exercise you will find a file `classical_looping.cpp` showing two standard ways of looping through vectors. Simplify the `iteratorLoop` by using `auto`. Using range-based loops it can be simplified even further. Try implementing another function: `rangeLooping`, which uses a range-based for loop as an even simpler way to go through the data.

3.2 Programmer mistakes

The file `range_looping.cpp` contains an attempt by an inexperienced programmer to use range-based loops. It seems terribly slow. Can you spot the bugs and fix them?

3.3 Lambdas

As you've learned in the lectures, C++11 and later support lambdas and closures. The file `lambda.cpp` uses a function and a function pointer to increment a simple counter. Replace these with a lambda. Lambdas are helpful in parallel programming, so this brings you in position to put them to good use, for example in the Fibonacci exercise.

4 Introduction to GDB

As stressed during the lectures, debugging can be a crucial step in the development cycle. One efficient way to debug programs is to use GDB. We provided you with a precompiled program `buggy` that crashes. Fortunately, it was compiled with debugging symbols, so you can examine it with GDB.

Hints: to run a program in GDB you can perform these two operations

```
gdb ./myProgram
(gdb) run
```

Remember that once you put a break point in GDB, you can print the content of a variable with `p <variableName>` and you can use the commands `n` and `s` to go to next line or alternatively to step into function calls. You can set break points with `b`. You can see the source with `list`, in particular you can list source code between two line numbers with `list n1,n2`.

When the program crashes, you can show the stack trace with the command `bt` and change to a stack frame with the command `f N` where N is the number of the frame.

In this case the problem is quite simple. You can check the source code in the Solution folder after you found the problem with GDB. But there are situations in which looking at the code in order to find a bug is simply infeasible, for example because of the size or complexity of the code base, or because you need to know the run-time values of all variables.

5 Debugging Parallel Applications

5.1 GDB

GDB offers the possibility to debug parallel applications. In this exercise we will inspect a simple program which increments a counter in two threads.

5.1.1 example1

First, compile the example with the following (plus appropriate warning flags if desired):

```
g++ -o example1 -std=c++17 -pthread -g example1.cpp
```

The additional flag `-pthread` is not always needed, but we want to use the pthread library for multithreading. Some compilers link to it automatically, but it doesn't do any harm to link again explicitly.

Running the program, we see that despite incrementing the counter in line 12 with

```
  ++counter;
```

the final result is still zero. Try to find the problem by setting breakpoints on the lines where the increment happens.

We can also set a breakpoint to inspect `counter` at line 9. However, this time we will set a temporary breakpoint via `tbreak`, that stops the execution only once. Now we can make gdb watch the counter using `watch` (note that we

have to break into the scope where `counter` is defined before we can set this watchpoint):

```
(gdb) watch counter
(gdb) watch (*counter)
(gdb) continue
```

The central feature of `watch` is that it prints the old and the new value of the watched variable. It should now be possible to understand and fix the bug in the lambda. After compiling and executing the fixed program, you might get non-reproducible results: try to execute the program a few times using `watch ./example1`. Note that this is unix watch, not gdb's watch.

If your results are not reproducible, a deeper analysis may be needed! You may have noticed lines like

```
[New Thread 0x7ffff78d5700 (LWP 9017)]
```

in the GDB output (the value is of course not the same for every execution). The GDB session automatically switches to the thread reaching a breakpoint and stops the other threads at the same time. Let's watch the counter again, and observe how it is incremented from the two threads. To know which thread you are currently in you can use `thread`. To know the status of all threads use `info threads`.

This thread switching in the context of the increment of a variable is an alarm – multiple threads modifying the same data, so we would need to somehow protect this resource. If you like, try to fix the bug using what you've learned about atomics during the lectures.

5.1.2 example2

Now, things become a little bit trickier. In this example, the threads are reaching the counter in different functions. You might be tempted to watch counter in line 10, but that may not be enough, since this doesn't cover the other thread. But even here GDB gives us a handle to find out what's happening – by monitoring explicit memory addresses like in the example below:

```
(gdb) tbreak example2.cpp:21
...
(gdb) watch -l counter
```

Now every change to the *memory location* of `counter` will be watched and we see which threads accesses it[1].

Remember this functionality when a memory location seems to change without an obvious reason. It might be another thread or an accidental use of that memory location, and `watch -l` can reveal this.

5.1.3 example3

As last GDB exercise, we change the stopping behaviour of GDB. We will demonstrate that GDB is not only a debugging tool but it can alter the runtime behaviour of an application!

[1] In older GDB versions one has to retrieve first the address and then set a watchpoint to the adress explicitly, so always use modern versions of gdb if possible.

By default, a breakpoint stops all threads. This is called *all-stop* mode. One can instruct GDB to halt only the thread that reaches the breakpoint, though. This is called *non-stop* mode. To see the difference, compile `example3.cpp`, and familiarise yourself with how it uses the counters: set a temporary break point to the line that increments `counter`, and print its value a few times in a row after you hit the breakpoint.

Now restart GDB and issue the following two commands before running the example:

```
set mi-async on
set non-stop on
```

Set again a temporary breakpoint where the threads increment the counter. Run the executable. You will notice that GDB will tell you about the breakpoint being reached, but it will not automatically switch into the halted thread. You have to do this explicitly. Use `info thr` or `info threads` to see what the threads are doing, and change to the halted thread using `thread <id>`. Once you did that, print the value of the `counter` (`p counter`) a few times. The value should be changing. Did you expect that? What is going on? Check again `info threads` if necessary.

5.2 Optional: Thread sanitizer

Thread sanitizer was briefly mentioned in the lecture. Let's go back to `example2`, and try to debug it again using thread sanitizer. Since `libtsan` is not installed on the CSC machines, we need to set up an LCG view (see first page). We can now compile as follows:

```
source setupScriptCvmfs.sh
g++ -std=c++17 -fsanitize=thread -g example2.cpp -o example2
```

You should notice that we asked the compiler to "sanitize" the executable using an additional flag. This will add run-time checks to the program, which make it run slower. Now run the program and see what the result is. If you used `-g` as shown above, you should get a very clear indication where the race condition happens. Try removing `-g` to see the difference if you have the time.

The downside of thread sanitizer is that you have to recompile everything, so using gdb might be the quicker option. Sometimes, however, the programs are so complicated that it's difficult to find a data race with gdb.

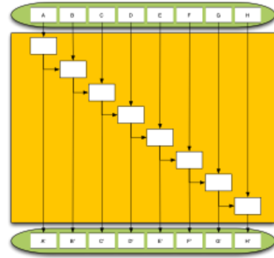
If you finished all other tasks, your last option to debug race conditions is `valgrind --tool=helgrind <program>`. Usually, you don't need to recompile the program, but helgrind can be very slow. Of course, we shouldn't run both helgrind and thread sanitizer, so now you have to recompile if you want to try it.

6 Optional: Computing Fibonacci Numbers in Parallel using Scan

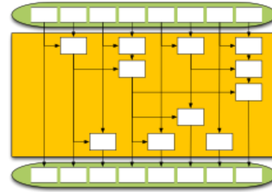
In the lecture we discussed whether Fibonacci numbers can be computed in parallel using scan. A parallel scan needs to know the previous element of the

sequence, but for Fibonacci numbers we need the *two* previous elements:

$$F_n = F_{n-1} + F_{n-2} \tag{2}$$



(a) Simple scan



(b) Parallel scan

However, if we use a more complicated algorithm to compute the numbers, we can fit two Fibonacci numbers into a single element of a new sequence. This is done by applying a 2x2 matrix to a vector of two Fibonacci numbers:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} A = __ & B = __ \\ C = __ & D = __ \end{pmatrix} \cdot \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} \tag{3}$$

The subsequent element can be reached by applying the matrix twice:

$$\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} \tag{4}$$

Therefore, we can reach any number by starting at $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, and repeatedly applying the matrix.

Tasks:

1. Figure out the coefficients of the matrix in eq. 3.
2. Convince yourself that all prerequisites for a parallel scan are fulfilled.
3. Go to the exercise folder, and have a look at the `FiboMatrix` class. Fill the matrix coefficients at the location marked with "Task 3".
4. Test the implementation (see details in the code, marked with "Task 4") To compile the code, you need to link against the TBB library, so the compilation will look like:

   ```
   g++ Fibonacci.cxx -o Fibonacci -O2 -g -std=c++17 -Wall -ltbb
   ```

   To test the algorithms, let's now:

   (a) Implement a trivial sequential Fibonacci computation "Task 4.(a)".

   (b) Run a few steps of the matrix algorithm, and look at the results.

(c) If necessary, correct the matrix until the Fibonacci sequences show up.

5. Now let's use a full scan to compute thousands of elements of the sequence. We don't need to implement scan ourselves, though, since C++ has `inclusive_scan` or `exclusive_scan`. Hints to do this can be found in the code.

   Compare the run times with the trivial Fibonacci algorithm. Since you share the CSC machine with other students, you might be competing for CPUs with them. In order to get stable run times, you can run the Fibonacci program within the tool "watch" (`watch ./Fibonacci`, this is not gdb's watch), which will run your program in a loop, and update its output on the screen. If the output fills the whole terminal, comment out the `print()` functions or run "`watch ./Fibonacci | grep ms`".

   Keep the program running until you are confident that the run times are stable. You should observe that the matrix algorithm is slower. What is the reason? What is the difference between inclusive and exclusive scan?

6. In C++17, scan can be parallelised easily by using the parallel algorithms library. Implement the parallel computation, and measure the speedup using "watch". Again, details on how to do this are given in the code. What speedup did you expect, and what did you achieve?

7. Optional: If you have time, you can use TBB to exert more control over the computation. We can for example change the number of threads by passing them as an argument to the program. Since TBB has a steep learning curve, the implementation is already given. Note how lambdas help us to express the iteration and reduction steps of a scan. Uncomment the code and compile the program. You can now change the number of elements[2] and the number of threads using command line arguments:

   ```
   g++ Fibonacci.cxx -o Fibonacci -O2 -g -std=c++17 -ltbb
   ./Fibonacci <nFibonacci> <nThread>
   ```

   Study the details of the TBB implementation a bit, and ask the tutors if you want to know more. Note that most TBB algorithms have a similar interface, so understanding `parallel_scan` will unlock your understanding of map, reduce, for_each etc.

   Now measure the scaling behaviour using TBB on e.g. 1M to 10M elements, and change the number of threads from 1 to the number of cores of your machine. Does the speedup scale linearly? What did you expect?

[2]Note: The Fibonacci numbers grow very quickly, so if you ask for a large number of elements, this will very quickly overflow our 64-bit integer. We will accept to get wrong numbers, because we want to stress our algorithms a bit, but in a real setting, we would have to move to 128- or 256-bit integers.