CSC2022 – 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 lxplus8 with Centos 8 stream to work on the problems.

NOTE: It is likely that you will not be able to finish all the exercises in the three hours at your disposal. Don't feel frustrated. Just leave aside the optional sessions during a first pass if you wish, you can come back to them later. In any case, we encourage you to ask the tutors for hints and additional explanations also 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.

Let's now fetch the exercises and set up our environment. You have two ways to do this. Either download the tar archive from indico and get it to lxplus, or use `wget` to download directly on lxplus:

```
ssh lxplus8.cern.ch
wget -O csc2022.tar.gz http://cern.ch/go/Wv7k
tar -zxf csc2022.tar.gz
cd csc2022
source setupScriptCvmfs.sh
```

The directory `csc2022` contains one subdirectory for every exercise. If applicable, every exercise subdirectory contains a directory with the solution. Try to make as much progress as possible for every exercise – 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 type the compilation commands. Try to invoke as many compilations as possible yourself, but all compilation command lines are available on the top of the source files as a comment if needed.

Setting up the environment

If you followed the steps above, you did already set up your software environment (such as a modern compiler and debugger). Remember to do this again if you log out and resume later. On every login to lxplus, go to the `csc2022` folder and type:

```
source setupScriptCvmfs.sh
```

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:
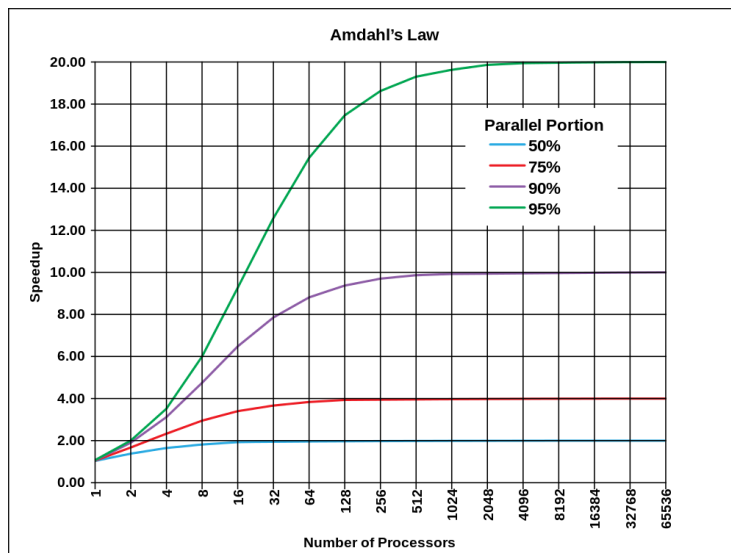

Figure 1: Amdahl's law for different values of P (from Wikipedia)

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

where S represents the speedup, N is the number of CPUs available, and P the portion of the program that be parallelised.

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. The percentage of time spent by this application waiting for I/O requests is 75% and is purely sequential. 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 a new 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 by 5% of the overall time. What would be the most profitable solution? Why?

2 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

in the second line 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 an eleven-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.

2.1 The auto keyword and range-based loops

In the Exercise 2 directory 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`.

2.2 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?

2.3 Lambdas

As you've learned in the lectures, C++11 supports lambdas and closures. The file `lambda.cpp` uses a function and a function pointer to increment a simple counter. Replace the function by a lambda.

3 The Map-Reduce Pattern of Parallelism

Let's get hands-on experience with the Map-Reduce pattern in Spark.

In order to run this exercise you will be using the CERN SWAN service. The exercise is structured as a Jupyter notebook. You will find the instructions about how to proceed in the notebook itself.

Note: If you have never used SWAN before, you might need to create a cernbox account to create your own CERN cloud storage. Go to `https://cernbox.cern.ch` and log in, and your cernbox will be created.

When SWAN starts up, you will be prompted to choose a software environment. You can use the default, which is the same environment that you use on lxplus.

Click here to start a SWAN session and download the exercise: `http://cern.ch/go/t8pP`

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. It was compiled with debugging symbols, though, 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 to 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

GDB offers the possibility to debug parallel applications. In this exercise we will inspect a simple program which increments a counter in two threads. First compile the example with the following (plus appropriate warning flags if desired):

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

We see that despite incrementing the counter in line 10 with

```
  ++counter;
```

the final result is still zero. While the error may be trivial in such a small example, more complex code isn't that nice to us anymore. So let's see how one tackles such problems with GDB. We can again set a breakpoint to inspect `counter` at line 10. However, this time we will set a temporary breakpoint via `tbreak`, that stops the execution only once. We will force further stops with another feature of GDB – `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 easy to understand and fix the bug. After compiling and executing the fixed program, we are stuck with non-reproducible results: try to execute the program a few times. A deeper analysis is needed!

Let's have a closer look. 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 the breakpoint and stops the other threads at the same time. 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.

With `example2.cpp` things become a little bit trickier. The method of watching the counter variable does not work anymore, as the different threads really execute different code. 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].

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! This exercise is somewhat more advanced than the previous two. Per 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. This is called *non-stop* mode. Compile `example3.cpp`, set a watch point to `counter` and print its value after reaching a breakpoint two times in a row. Now restart GDB and issue the following two commands before running the example:

```
set mi-async on   (previously: target-async on)
set non-stop on
```

Define the counter increment breakpoint for the function `incrementCounter`: as you know, you can do that either with the function name or with the line number. 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 via `thread id`, the first column that you see with the command `info thr`. Once you did that, print the value of the `counter` (`p *counter`) a few times. The value is changing. Did you expect that? What is going on? Remember that the operation of printing a variable value needs a finite amount of time to be accomplished.

6 Profiling with IgProf

In this exercise, we will analyse a program that suffers from serious performance bottlenecks, which are due to a faulty implementation. Although the program is *sequential*, the ability to properly measure the performance of sequential programs continues to be crucial in the multicore era. How can you decide that an

---

[1]In older GDB versions one has to retrieve first the address and then set a watchpoint to the adress explicitly.

application is to be parallelised and how if you don't have a clear and detailed picture of its performance?

In order to measure code performance, we will use the IgProf profiler, already introduced during the lectures. The toy example we will discuss in this exercise is a program that reads in data of customers such as name, address and phone number and then organises it in a certain data structure sorting the entries according to the name of the customer.

To start, let's generate some fake data. In the exercise directory, just type:

```
bash ./generateFakeData.sh
```

To compile the analysis program and profile it, you can type:

```
g++ -o customers customers.cpp -std=c++17
igprof -d -pp -z -o customers.pp.gz ./customers bigFakeData.txt
igprof-analyse -d -v -g customers.pp.gz | less
```

Have a look to the self costs chart in *customers_perf.res* (look for "self" in the file). Note that you obtained the profile *without* any code instrumentation or special compilation flags! What are the main offenders (say, are there any unwanted copies going on)? Can you optimise the program? One idea could be to try to reduce the number of copies and allocations. You could achieve this also with more suitable containers for the data, for example the `std::deque`. As an indicator, the number of calls to the copy constructor of `customer` is printed at the end of the execution. Try to reduce it zero.

7   Optional: Putting it all together

In this final exercise, you will be facing the problem of managing letters. We have discussed this during the lectures (see figure 2).
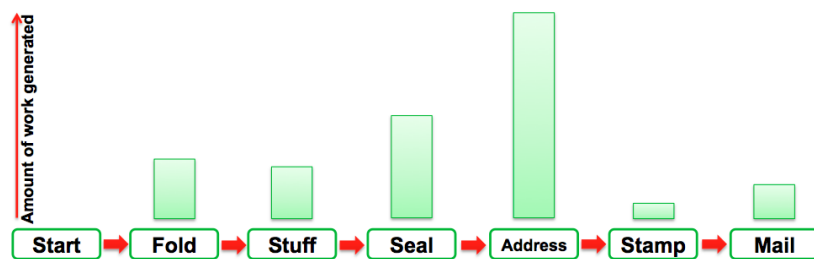

Figure 2: Steps in order to send a letter. The dynamic assignment of workers to the work items is an efficient way to solve the problem.

The idea is to develop a parallel program that mimics a simplified processing of mail. Basically, you would need to have a class representing a mail item going through a chain of states:

Start → Folded → Stuffed → Sealed → Addressed → Stamped → Mailed

You will have to implement a mechanism to operate this state machine (evolve the states of the single items), either with external functions or with methods

of the mail item class. These evolutions are the work items and can be organised as C++ closures and stored into a concurrent stack. You can implement a simple one using locks. Atomics or transactional memory are much harder to get right, even if they are faster.

Since this is an exercise, no real work is actually performed by the workers: they just change the state of the mail items. Since this operation is extremely quick, you can increase the impact of the demonstrator by letting the threads sleep for a short while to achieve a more realistic demo. Threads, the workers, will pull work from the stack (pop an item and simply execute it). You can create a "pulling function" and pass it to the threads. This function returns when there is no more work left to do. As for printouts, you can either adapt the pattern of having a service thread printing messages or you can simply protect the `cout` statement with a lock.

In order to see an animation of the evolution of the state machine described above, you can have a look at `animatedMailItemProcessor`. To steer the simulation you can use the command:

```
animatedMailItemProcessor <Number of Mail Items> \
<Number of Workers>
```

This program is not exactly a trivial "nice to have". It can give you a taste of how a task-based approach works step by step during the program execution.

The exercise is relatively complex, but you can decide the level of difficulty you would like to deal with. You will find three files representing three difficulty steps between a blank screen and the full solution:

- *easy.cpp*: this program compiles with the following command

  ```
  g++ -o easy easy.cpp -g -std=c++17 -pthread -fgnu-tm
  ```

  But it crashes. Try to put to a good use what you learned up to now and fix it.

- *normal.cpp*: some of the implementations are there, some replaced by comments paving the way for you to fill in the missing pieces.

- *hard.cpp*: like normal, but more code has to be put in place.

To facilitate the understanding, the author made some short cuts in the example implementation, including some global variables. To see how a better implementation would look like, the solutions contain an additional `mailItemBetterDesign.cpp` file.