Programming Parallel Computers

Instructions

Hints

When you go to look at the information on your submission, you will see a lot of information there. At first it can be overwhelming, and you can focus on the first parts that contain e.g. information on tests that failed. But later when you get more comfortable with the system and you have got your code working correctly, you can also start to pay attention to the bottom parts of the page. There you will have also more detailed measurements on the performance of your code, and you can even go there and have a look at the assembly code produced by the compiler. This can be especially valuable if you do not have similar hardware available elsewhere, and you are trying to make good use of the features of our computers.

Development environment

When you download the zip file containing the code templates for a task, you should be able to unzip and use it directly on a typical Linux and macOS computers if you have the usual development tools installed, as long as you have got a sufficiently new C++ compiler: GCC version at least 8 or Clang version at least 6 (both were released in 2018).

If you are using Ubuntu Linux 20.04, you can run `sudo apt install g++ python3` to install the relevant tools.

If you are using Ubuntu Linux 18.04, you will also need to install a more recent compiler with `sudo apt install g++-8` — our scripts should automatically find the most recent compiler even if it is called e.g. `g++-8` instead of `g++`, so no other changes besides installing the right package are needed.

If you are using Debian 9, you will probably want to update to Debian 10 to get GCC 8.

If you are using macOS, you should get a working development environment this way:

- Run `xcode-select --install` to install the command line development tools.
- Install Homebrew.
- Run `brew install libomp` to get OpenMP support.
- If you want to use GCC instead of Clang, you can also run `brew install gcc` to install it.

If you are using Windows Subsystem for Linux to use Linux under Windows, please be careful to also unzip the files inside the Linux subsystem. If you unzip in the Windows side and try to use it in the Linux side, e.g. symbolic links will not work correctly.

If you have got an Aalto University user account, you can also use remotely the Linux computers that are available in the Maari building:

- First connect to one of the general-purpose Linux computers at Aalto University, using e.g. `ssh kosh.aalto.fi` (with your Aalto user name and password)
- Then continue to from kosh to one of the computers in Maari-A or Maari-B, e.g. `moa`, by simply running `ssh moa` — it should not ask for any user name or password. If this does not work, run `kinit` in kosh, enter your Aalto password, and it should work again for some time.

If you are using some computer via ssh, to get the code templates there it is probably easiest to copy the download link and use `wget` to download the zip file.

What is going on behind the scenes

You can use the `-v` flag (e.g. `./grading -v test`) to see what are the commands that the grading tool is running. This way you can more easily see what went wrong if the grading tool is not co-operating in your local environment, and you can also see exactly how your code is compiled.

The grading tool will compile your own code (e.g. `cp.cc`) together with our tester code into one executable file (e.g. `cp`). You can also run the executable file directly from the command line, or under a debugger or profiler if needed. Again, use `./grading -v test` to find the right command to use.

Convenient utilities

In the zip file with the code templates, you will find some convenient definitions in the file `.ppc/ppcgrader/include/vector.h`. You can simply use `#include "vector.h"` in your submission to make use of these. They are automatically available both when you test your code locally and when you submit to our automatic grading system.

Best practices

It is highly recommended that you keep all of your code in your own private Git repository. You can this way easily collect all of the code templates in one place, and keep track of your own solutions.

Debugging correctness-related issues

If your code unexpectedly crashes, please consider these:

- Our submissions system will automatically use AddressSanitizer to try to catch many common bugs related to memory management and indexing (e.g. accessing arrays out of bounds). If you get error messages from the AddressSanitizer, it is great, please read the message carefully, it usually directly indicates exactly where things went wrong.
- If you do not get errors from the AddressSanitizer, but your code still crashes, please double-check that you do not have stack overflows. Please keep in mind that a typical maximum stack size on Linux computers is only 8MB. Do not allocate large arrays on the stack. If you need to allocate storage for megabytes of data, use the heap.
- Please also ensure that you have got the right memory alignment for arrays of vectors — please see our course material for more details.

If you get wrong results, please consider these:

- You might be reading memory that is not initialized. In C and C++, memory allocation functions typically do not guarantee that memory is initialized with zeros. However, it is easy to forget to initialize newly allocated memory, and in many cases your program may accidentally work correctly as newly allocated memory often happens to contain all zeros.
- If you are using CUDA, please remember to check for errors in all CUDA API calls — please see our course material for examples of how to do that.