Universität Zürich

Physik-Institut

Programmieren in C++
FS 2023

Lecture 10: Parallel programming in C/C++

1 Introduction to parallel programming

Today we will go over parallel programming. There are two major types of parallel programming you need to be aware of. The first one we will go over is called parallel processing. This means that each CPU core is given an independent task. Each core gets to use a given amount of RAM, and if programmed properly, the cores do not need to communicate with each other to perform its tasks.
The second major type of parallel programming is called parallel-threading. Modern CPUs have more than one communication line between the CPU cores and the rest of the computer. Each communication line is called a thread. The advantage of having multiple thread on a single core of the CPU is that the CPU can perform more than a single task in one "cycle". Recall our data processing lecture, you were asked to calculate the mean, median and standard deviation and etc. under a single loop. A CPU core has multiple components which are responsible for specific operation, such as addition, or comparing two values. The exact works of how a CPU works depends largely on the model and design of the specific CPU and we will not cover that in this lecture. For more details regarding the operation of a CPU, I encourage you to watch this video. For this course, what you need to know is that you can perform multiple tasks out of a single core by using multi-threading feature.

2 Parallel processing

Parallel processing is rather clean and an easy way to divide a task between CPU cores. You need to make sure of the following before you start however.

1. How many CPU cores does your computer have?
2. How much RAM does your computer have?
3. How much RAM will each instance of your program use?

The first two, you can simply ask your system. Check the file /proc/cpuinfo

```
cat /proc/cpuinfo |grep -i core
model name      : Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz
core id         : 0
cpu cores       : 2
model name      : Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz
core id         : 1
cpu cores       : 2
model name      : Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz
core id         : 0
cpu cores       : 2
model name      : Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz
core id         : 1
cpu cores       : 2
```

The above means you have two CPU cores with 2 threads each. You have therefore 4 threads, or sometimes called 4 logical cores. For memory, it's best to use "free"

```
free -h
              total        used        free      shared  buff/cache
   ↪  available
Mem:          3.8Gi       409Mi       531Mi       199Mi       2.8Gi
↪   2.9Gi
Swap:         8.0Gi          0B       8.0Gi
```

The above means you have 4 GB of RAM installed, and your system is already using up 409 MiB. You also have 8 GiB of "Swap" which are the "virtual" memory using the hard-drive. If you ever need to use SWAP in your program, you lose all performance advantage of using RAM. Also, it typically destabilizes the systems functions as the processes will be slowed down greatly.

If you are using one of the spinorXX.physik.uzh.ch, limit yourself to 2 logical cores and 1.5 GB of RAM for total usage. This is because most of you are sharing a computer with 4 GB of RAM. If you use up all of the RAM, the computers will either crash and/or freeze. We strongly encourage you to use your own computer for this lecture.

For the last item, you can calculate how much RAM your program will use by hand-calculation and/or write your own code to perform this calculation before you execute your program.
For the purpose of our course, we will use a bash package called "gnu-parallel" https://www.gnu.org/software/parallel/. This package is already installed in spinorXX.physik.uzh.ch
In your version of Linux, you will be able to find one in the default package manager (such as YAST, apt-get, aptitude, yum and etc.). In Mac, this should available through "brew"
The installation should take about 1 minute.
Now, consider the following code:

```cpp
//writerandom_and_average.cxx
#include <iostream>
#include <stdio.h>
#include <vector>
#include <fstream>
#include <sstream>

using namespace std;

void writerandom(int variable, string filename){
        stringstream filename_extender;
        filename_extender<<"randoms/"<<filename<<".txt";
        ofstream outputfile;
        outputfile.open(filename_extender.str().c_str());
        for (int i=0; i<variable; i++){
                outputfile<<rand()<<endl;
        }
        outputfile.close();
}
void writeaverage(string inputfilename, string outputfilename){
        stringstream inputfilename_extender;
        inputfilename_extender<<"randoms/"<<inputfilename<<".txt";
        stringstream outputfilename_extender;
        outputfilename_extender<<"randoms/"<<outputfilename<<".txt";

        ifstream inputfile;
        inputfile.open(inputfilename_extender.str().c_str());
        inputfilename_extender.str(string());
        inputfilename_extender<<inputfilename<<".txt";
        ofstream outputfile;
        outputfile.open(outputfilename_extender.str().c_str() ,ios::app );
        int temp_int=0;
        int sum=0;
        int counter=0;
        while(inputfile >> temp_int){
                sum+=temp_int;
                counter++;
        }
        int average = sum/counter;
        outputfile<<inputfilename_extender.str()<<"\t"<<average<<endl;
        inputfile.close();
        outputfile.close();
}

int main(int argc,char *argv[]){//Main begins

        if(argc==1){
                string filename="name";
                stringstream namecounter;

                for (int i=0 ; i < 1000; i++){
                namecounter.str(string());
                namecounter<<filename<<i;
                writerandom(100,namecounter.str().c_str());
                writeaverage(namecounter.str().c_str() ,"averages");
                }

        }

        if(argc==2){

                string filename=argv[1];
                stringstream namecounter;

                for (int i=0 ; i < 1000; i++){
                namecounter.str(string());
                namecounter<<filename<<i;
                writerandom(100,namecounter.str().c_str());
                writeaverage(namecounter.str().c_str() ,"averages");
                }
        }

        if (argc==3){

                string filename=argv[1];
                int number_of_random_files=strtol(argv[2], NULL, 10 );
                stringstream namecounter;

                for (int i=0 ; i < number_of_random_files; i++){
                namecounter.str(string());
                namecounter<<filename<<i;
                writerandom(100,namecounter.str().c_str());
                writeaverage(namecounter.str().c_str() ,"averages");
                }
        }

    return 0;
}//Main Ends
```

The code above can be executed in 3 different ways. First one:

```
run_writerandom_and_average.exe
```

This will run argc==1 case, where it writes 100 random integers into nameX.txt file. It will write 1000 nameX.txt file and log the average of nameX.txt files into averages.txt file.

```
run_writerandom_and_average.exe newname
```

This will run argc==2 case, where it writes 100 random integers into 1000 newnameX.txt file and logs the average of newnameX.txt files into averages.txt file

```
run_writerandom_and_average.exe blah 5000
```

This will run argc==3 case, where it writes 5000 blahX.txt file and logs the average of blahX.txt file into averages.txt file. This isn't particularly useful, but we will use the above as example for running parallel processes.
For instance, suppose you wanted to execute all three in series. You should write them into a bash script.

```
#!/bin/sh
#runeverything_serial.sh
run_writerandom_and_average.exe ;
run_writerandom_and_average.exe newname 5000;
run_writerandom_and_average.exe blah 3000;
```

If you run the above script, everything will run in series, one process after another. Even on my high performance laptop, this takes quite a while.

```
> time runeverything_serial.sh
./runeverything_serial.sh: line 3: $'\r': command not found
./runeverything_serial.sh: line 4: $'\r': command not found

real	0m28.450s
user	0m1.281s
sys	0m19.547s
```

Note that the above error message will appear if you are running a bash script using WSL from a Windows 10 computer. However this doesn't affect your operation or performance at all. Now, let's us gnu_parallel to run the three of them and time it. We'll first write the following in a bash script.

```
#!/bin/sh
#runeverything_parallel.sh
parallel ::: './run_writerandom_and_average.exe one 1000'
↪  './run_writerandom_and_average.exe newname 5000'
↪  './run_writerandom_and_average.exe blah 3000'
```

When we run the above, we will notice that we're saving some time. Also, writing down each and every operation gets tedious quickly. We also want to have control over how many logical cores are assigned to the task.

```
time runeverything_parallel.sh
real    0m10.496s
user    0m1.313s
sys     0m12.719s
```

To have a better control of how many logical cores are being written, we should write the commands in a temporary file parameters.tmp:

```
./run_writerandom_and_average.exe one 1000
./run_writerandom_and_average.exe newname 5000
./run_writerandom_and_average.exe blah 3000
```

then write a script as the following

```
#!/bin/sh
cat parameters.tmp | parallel -j 3
```

The last script must end at "3". The script has to have no extra lines or space, otherwise gnu_parallel will give you an error. Gnu_parallel is very sensitive to syntax including where you place an empty space. The 3 denotes that 3 jobs will run simultaneously. More jobs/cores you allow for the job, the faster the execution will be. Now, let's start practice using these.

= The practical programming part of this course will now begin for 60 minutes. =

3 Practice parallel processing

1. Write "writerandom_and_average.cxx" from scratch and test it in serial mode.
2. Write a bash script to run the above for 10 different file names, 2000 random numbers in serial and time it.
3. Write a bash script do the same using gnu_parallel and time it.
4. Modify your code so that you can generate the random numbers, store them in your RAM then write at the end of the program. Test it and run the timer to compare with everything else before.
5. Visit https://www.gnu.org/software/parallel/parallel_tutorial.html for more information, and try to execute your programs in parallel using different syntaxes not mentioned in the lecture.

= The theoretical lecture part of this course will now continue for 15 minutes. =

4 Parallel threading

It's important to note that even though parallel threading can be considered relatively new, there has been enough time for multiple companies and developers to create packages to perform parallel threading differently. There are options such as Boost, QT Qthread, (standard library) thread or (POSIX) pthread. Some are considered more stable in some operating systems than the others. For our course, we will only use pthread. For more information please visit https://www.bogotobogo.com/cplusplus/multithreading_pthread.php, https://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11.php and https://en.cppreference.com/w/cpp/thread.

4.1 Pthread example

In order to use pthread you must include at the top of your code as usual.

```
#include <pthread.h>
```

If you try to compile right away, compilation will fail since you need to give your compiler another flag

```
-lpthread
```

Your Makefile now should look something similar to the following

```
#This is the directory of YOUR source code.
sourcedirectory=./

#These are your source codes and components
trial=$(wildcard *.cxx)
first_part=myobjects.cxx
second_part=main.cxx
component=myobjects.cxx

#Here, we're defining the compilers.
CC=gcc
CPP=g++
NVCC=nvcc

#We're defining systems variable such as "remove" from system and
↪  "timestamp"
RM=rm
TIMESTAMP=$(shell date +"%Y_%m_%d_T-%H_%M" )

SFLAG=-Wall
PFLAG=-lncurses -lpthread
ROOTFLAG=-g $(shell root-config --cflags --glibs)

ALLFLAG= $(SFLAG) $(ROOTFLAG)


objects = $(trial:.cxx=)

#objects =
#When a Makefile is executed, by default it tries the option "all"
all: clean $(objects)
```

```
#We will tell the makefile to clean, compile the first component, second then the third component.


$(objects): %: %.cxx

        @echo Compiling $(sourcedirectory)$<
        @$(CPP) -o $(addprefix run_,$@.exe) $(SFLAG) $< $(PFLAG)
        @echo Successfully compiled $(sourcedirectory)$<
        @echo executable is $(addprefix run_,$@.exe)

first:
#Here, we define what "first_one" will do.
#At sign @ will silence the command appering in the terminal
        @echo Compiling $(sourcedirectory)$(first_part)
        @$(CPP) -o $(first_part).o  $(SFLAG) $(sourcedirectory)$(first_part) $(PFLAG)
        @echo Successfully compiled $(sourcedirectory)$(first_part)
        @echo The compiled object is $(first_part).o
second:
#Let's now compile the second part.
        @echo Compiling $(sourcedirectory)$(second_part)
        @$(CPP) -c -o $(second_part).o  $(SFLAG) $(sourcedirectory)$(second_part) $(PFLAG)
        @echo Successfully compiled $(sourcedirectory)$(second_part)
        @echo The unliked compiled code is $(second_part).o

        @echo Compiling $(sourcedirectory)$(component)
        @$(CPP) -c -o $(component).o $(SFLAG) $(sourcedirectory)$(component) $(PFLAG)
        @echo Successfully compiled $(sourcedirectory)$(component)
        @echo The unliked compiled code is $(component).o
third:
#Let's link the second part
        @echo Linking $(component).o and $(second_part).o
        @$(CPP) -o compiled_program.exe $(SFLAG) $(sourcedirectory)$(component).o $(sourcedirectory)$(second_part).o $(PFLAG)
        @echo Successfully compiled $(sourcedirectory)$(component)
        @echo Everything is linked and compiled into compiled_program.exe
clean:
        @echo $(TIMESTAMP)
        @echo "Making old/$(TIMESTAMP) directory"
        $(shell mkdir -p old/$(TIMESTAMP) )
        @echo "Copying the source to the old directory"
        $(shell cp -r $(sourcedirectory)/*.cxx old/$(TIMESTAMP) )
        @echo "Moving all .exe to the old directory"
        $(shell mv *.exe old/$(TIMESTAMP) )
        $(shell mv *.o old/$(TIMESTAMP) )
        @echo "Copying the Makefile to the old directory"
        $(shell cp Makefile old/$(TIMESTAMP) )
#        £this line is only added to comply with LATEX formatting.
```

The only change you since the last Makefile example is that now PFLAG in line 20 has -lpthread also and that line 37 uses SFLAG instead of ALLFLAG.
Now that we've set the flags, let's go over an example.

```
1  //pthreader.cxx
2  #include <iostream>
3  #include <stdio.h>
4  #include <vector>
5  #include <fstream>
6  #include <sstream>
7  #include <pthread.h>
8
9  using namespace std;
10
11 void *ThreadChecker(void *thread_ID){
12         long t_id;
13         t_id=(long)thread_ID;
14         cout<<"Checking thread: "<<t_id<<endl;
15         pthread_exit(NULL);
16 }
17
18 int main(int argc,char *argv[]){//Main begins
19         const int max_threads_used=8;
20         const int mtu=max_threads_used;
21
22         pthread_t threads[mtu];
23         int runcommand;
24
25         for (int i=0; i<mtu; i++){
26                 runcommand = pthread_create(&threads[0], NULL,
   ↪   ThreadChecker, (void *)i);
27                 if(runcommand){
28                         cout << "Error:unable to create thread," <<
   ↪   runcommand << endl;
29                         exit(-1);
30                 }
31         }
32
33         pthread_exit (NULL);
34     return 0;
35 }//Main Ends
```

The above code will initiate your threads and let you know if is possible to assign the thread a job. In my computer, I have 8 threads. I can process 8 threaded process in parallel instantaneously. If I place a number higher than 8 in line number 19, the program will start running in Serial. The output of the above code is as follows:

```
Checking thread: 0
Checking thread: 1
Checking thread: 2
Checking thread: 3
Checking thread: 4
Checking thread: 5
Checking thread: 6
Checking thread: 7
```

Everything above is in order. However, as ”weird” as it may look, depending on which thread is being used by other component of the computer operations, the thread ID may not be in order. If the above command is run again after a while, I get the following:

```
Checking thread: 0
Checking thread: 1
Checking thread: 2
Checking thread: 3
Checking thread: 4
Checking thread: 6
Checking thread: 7
Checking thread: 5
```

There are also times when some of the threads are busy with another task, and may not be in synchronization with your code, in which case your program will produce core dump and crash. There are ways to prevent such things but we will not cover that in this course. Next, consider the following extension to the previous example:

```cpp
//pthreader.cxx
#include <iostream>
#include <stdio.h>
#include <vector>
#include <fstream>
#include <sstream>
#include <pthread.h>

using namespace std;

void *ThreadChecker(void *thread_ID){
        long t_id;
        t_id=(long)thread_ID;
        cout<<"Checking thread: "<<t_id<<endl;
        pthread_exit(NULL);
}

void writerandom(int variable, string filename){
        stringstream filename_extender;
        filename_extender<<"randoms/"<<filename<<".txt";
        ofstream outputfile;
        outputfile.open(filename_extender.str().c_str());
        for (int i=0; i<variable; i++){
                outputfile<<rand()<<endl;
        }
        outputfile.close();
}

void *assign_thread_writerandom(void *thread_ID){
        long t_id;
        t_id=(long)thread_ID;
        stringstream filename_extender;
        filename_extender<<"Written_By_Thread_ID_"<<t_id;
        writerandom(1000000,filename_extender.str().c_str());
        pthread_exit(NULL);
}

void writeaverage(string inputfilename, string outputfilename){
        stringstream inputfilename_extender;
        inputfilename_extender<<"randoms/"<<inputfilename<<".txt";
        stringstream outputfilename_extender;
        outputfilename_extender<<"randoms/"<<outputfilename<<".txt";
        ifstream inputfile;
        inputfile.open(inputfilename_extender.str().c_str());
        inputfilename_extender.str(string());
        inputfilename_extender<<inputfilename<<".txt";
        ofstream outputfile;
        outputfile.open(outputfilename_extender.str().c_str() ,ios::app );
        int temp_int=0;
        int sum=0;
        int counter=0;
        while(inputfile >> temp_int){
                sum+=temp_int;
                counter++;
        }
        int average = sum/counter;
        outputfile<<inputfilename_extender.str()<<"\t"<<average<<endl;
        inputfile.close();
        outputfile.close();
}

void *assign_thread_writeaverage(void *thread_ID){
        long t_id;
        t_id=(long)thread_ID;
        stringstream filename_extender;
        filename_extender<<"Written_By_Thread_ID_"<<t_id;
        writeaverage(filename_extender.str().c_str(),"average");
        pthread_exit(NULL);
}

int main(int argc,char *argv[]){//Main begins
        const int max_threads_used=8;
        const int mtu=max_threads_used;

        pthread_t threads[mtu];
        int runcommand;

        for (int i=0; i<mtu; i++){
                runcommand = pthread_create(&threads[0], NULL,
↪   ThreadChecker, (void *)i);
                if(runcommand){
                        cout << "Error:unable to create thread," <<
↪   runcommand << endl;
                        exit(-1);
                }
        }

```

```cpp
        for (int i=0; i<mtu; i++){
                runcommand = pthread_create(&threads[0], NULL, assign_thread_writerandom, (void *)i);
                if(runcommand){
                        cout << "Error:unable to create thread," << runcommand << endl;
                        exit(-1);
                }
        }

        for (int i=0; i<mtu; i++){
                runcommand = pthread_create(&threads[0], NULL, assign_thread_writeaverage, (void *)i);
                if(runcommand){
                        cout << "Error:unable to create thread," << runcommand << endl;
                        exit(-1);
                }
        }


        pthread_exit (NULL);
    return 0;
}//Main Ends
```

Notice that each job from parallel processing example has been modified such that a single thread can be assigned to a specific task. If you watch your systems monitor while this program is running, you will notice that all of your threads are being utilized.
Since there are absolutely no dependencies between threads, this code should almost always work. You can of course assign each thread to different jobs.
For the purpose of this course, we will not cover more materials, but you must also consider which task needs to be completed before another. This is called racing condition. If one operation needs to be completed before another, you must interrupt the thread that is ahead of the other and tell it to wait. We strongly encourage you to read https://www.bogotobogo.com/cplusplus/multithreading_pthread.php for more. For now, let's practice what we've covered.

= The practical programming part of this course will now begin for 60 minutes. =

5 Practice parallel threading

1. Change your compiler to include -lpthread and re-compile all of your codes.
2. Write from scratch the pthreader.cxx.
3. Compile and execute the results, be sure to change the number of maximum threads to match your systems specification.
4. Go to your functions package. Copy and create a new .cxx and .h files called "assign_thread_functions".
5. Create functions in the above so that you can assign a thread to each of the functions in your functions package.
6. Compile your functions package, objects package, assign_thread_functions and main all together and test the executable.

6 Conclusion

At this point, you have learned more than 3 semesters worth of programming in 10 days. We were able to do this because we left out a lot of theoretical components that are very important. We can really say that we've touched almost all of the important basic tools available in C++ but there are more tools and packages especially in the branch of parallel programming. There are packages such as OpenCL and OpenGL which allow you to also use the GPU. There is also a comprehensive proprietary GPU programming language called CUDA for NVIDIA graphics cards.
The purpose of this course is to show you the practical skills in C++ programming but obviously we did not have the time go through everything in detail. Be be sure to look online and read other resources as you program more in the future. The C++ language is always evolving and there are new changes and new packages being developed every year.
I wish to congratulate all of you who have been able to follow the course this far, and I hope the course will be helpful for everyone.

This concludes PHY224 in 2022. I wish to thank you all for your attention.

Thank you everyone!

-Steven J. Lee (SJLPHI)