Universität Zürich[UZH]
Physik-Institut

Lecture 10: Parallel programming in C/C++

1 Introduction to parallel programming

Today, we will discuss parallel programming. Two major approaches to parallelism are *parallel processing* and *multithreading*. In *parallel processing*, independent tasks are distributed across multiple CPU cores or processes. Depending on the programming model, each task may operate on its own data, so little or no communication between the tasks is required.
The second major approach is *multithreading*. A single process can contain multiple threads, which share the process's memory and resources. Modern CPUs can execute multiple threads concurrently, either by running them on different CPU cores or, on some processors, by allowing more than one hardware thread to share a single core. This can improve the utilisation of available CPU resources, especially when one thread is temporarily waiting for data or another resource.
Recall our data-processing lecture, where you calculated the mean, median, and standard deviation while processing the data in a single loop. A CPU core contains multiple components dedicated to different types of operations, such as arithmetic operations and comparisons. The precise way in which a CPU processes instructions depends heavily on its architecture and design, which we will not cover in this lecture. For a deeper understanding of how CPUs work internally, I encourage you to watch, e.g., this video. For this course, the key takeaway is that *multithreading* allows a single process to handle multiple tasks concurrently, potentially improving efficiency and performance.

2 Parallel processing

Parallel processing is a clean and straightforward way to distribute a task across multiple CPU cores. However, before getting started, make sure to check the following::

1. How many CPU cores does your computer have?
2. How much RAM is available on your system?
3. How much RAM will each instance of your program require?

You can easily find the answers to the first two questions by querying your system. For example, 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 indicates that you have two CPU cores, each with two *threads*. This gives you a total of four *threads*, which are sometimes referred to as four *logical cores*.

For checking available memory, the simplest way is to use the command `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.

The output above shows that the system has approximately 4 GiB of RAM available to the system, of which 409 MiB is currently in use. The system also has 8 GiB of swap space. Swap provides additional virtual memory by using storage space when physical RAM becomes scarce. However, accessing data from swap is much slower than accessing data from RAM. If your program requires more memory than is available in RAM and starts relying heavily on swap, its performance can therefore decrease dramatically. In extreme cases, heavy swapping can make the entire system very slow and unresponsive.

For the last item, you can determine how much RAM your program will use either by performing a manual calculation or by writing your own code to estimate memory usage before executing the program.

Important: You are strongly encouraged to use your own computer for this lecture. `ohmXX.physik.uzh.ch` hosts each have only two logical cores and 8 GiB of RAM). As a result, there are very limited resources available for parallel programming, especially when multiple users are working on the same host.

For the purposes of our course, we will use a `bash` package called `gnu-parallel` (https://www.gnu.org/software/parallel/), which is already installed on `ohmXX.physik.uzh.ch`. On your own `Linux` system, you can find it in your default package manager (such as `YAST`, `apt-get`, `aptitude`, `yum` and etc.). On `MacOS`, it should be available through `Homebrew` (`brew`). The installation typically takes only about one minute.

Let's start with a very trivial example:

```cpp
// test_CPU.cxx
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
  if (argc == 3){
    string identifier = argv[1];
    long n_operation = strtol(argv[2], NULL, 10 );
    cout << identifier << " executes " << n_operation << " operations" << endl;
    long dummy;
    for (long i = 0; i < n_operation; i++){
      if (i % 2){ dummy += i;}
      else { dummy -= i; }
    }
    cout << identifier << " has completed " << n_operation << " operations" << endl;
  }
  return 0;
}
```

First, we want to execute this program three times with different arguments *serially*. To do so, we place the commands into a `bash` script:

```
#!/bin/bash
# run_test_CPU_serial.sh
./run_test_CPU.exe A 10000000000;
./run_test_CPU.exe B 10000000000;
./run_test_CPU.exe C 10000000000;
```

Then we run the above script and measure the execution time:

```
$ time ./run_test_CPU_serial.sh
A executes 10000000000 operations
A has completed 10000000000 operations
B executes 10000000000 operations
B has completed 10000000000 operations
C executes 10000000000 operations
C has completed 10000000000 operations

real        0m41,649s
user        0m41,511s
sys         0m0,105s
```

As expected, we find that `A`, `B`, and `C` are executed one after the other.

Then we use `gnu_parallel` instead to run all three cases in parallel and measure the execution time. Again, we write the commands into a `bash` script:

```
#!/bin/sh
# run_test_CPU_parallel.sh
parallel ::: \
         './run_test_CPU.exe A 10000000000' \
         './run_test_CPU.exe B 10000000000' \
         './run_test_CPU.exe C 10000000000'
```

Running the above script, we notice that the *real* time, i.e. the *wall-clock* time, is reduced to almost one third of the previous (some increase in *user* and *sys* times is expected):

```
$ time ./run_test_CPU_parallel.sh
C executes 10000000000 operations
C has completed 10000000000 operations
A executes 10000000000 operations
A has completed 10000000000 operations
B executes 10000000000 operations
B has completed 10000000000 operations

real        0m14,378s
user        0m42,156s
sys         0m0,536s
```

We further observe that the order of `A`, `B`, and `C` may be interchanged. Moreover, we find that by the output of each of the three processes is only written at the very end, which is the expected default behaviour of `gnu_parallel`.

Now, we consider the following code, which also involves some I/O:

```
// writerandom_and_average.cxx (requires subdirectory "randoms/")
#include <iostream>
#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[]){
  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(1000, namecounter.str().c_str());
      writeaverage(namecounter.str().c_str(), "averages");
          }
  }
  return 0;
}
```

It also needs to be run with two arguments (i.e. `argc == 3`, including the executable itself):

```
./run_writerandom_and_average.exe name 5000
```

With these arguments, the program writes `5000` files `name.<X>.txt` with <X> ranging from `0` to `4999`. Each file contains `1000` random numbers, and the program logs the average of the random numbers in each file, together with the corresponding filename, in `randoms/averages.txt`.This has no particular practical application, but it provides a useful example for running processes in parallel. First, we want to execute this program three times with different arguments sequentially. To do so, we place the commands in a `bash` script:

```
#!/bin/bash
#run_wraa_serial.sh
./run_writerandom_and_average.exe A 1000
./run_writerandom_and_average.exe B 5000
./run_writerandom_and_average.exe C 3000
```

We run the script above and measure its execution time. The file `randoms/averages.txt` confirms that the processes are executed sequentially. The measured execution times are:

```
$ time ./run_wraa_serial.sh

real        0m3,673s
user        0m1,469s
sys        0m2,202s
```

Now, let's use `gnu_parallel` to run all three cases in parallel and measure the execution time. Again, we put the commands into a `bash` script:

```
#!/bin/bash
#run_wraa_parallel.sh
parallel ::: \
         './run_writerandom_and_average.exe ran_A 1000' \
         './run_writerandom_and_average.exe ran_B 5000' \
         './run_writerandom_and_average.exe ran_C 3000'
```

When we run the above, we see that it still reduces the execution time, but not as significantly as in the previous example:

```
$ time ./run_wraa_parallel.sh

real        0m2,217s
user        0m1,716s
sys        0m2,288s
```

Given that the three processes with our input values are not expected to take the same amount of time, our parallelization cannot be as effective as in the previous example. There can also be an additional performance impact from all three processes reading from and writing to the disk simultaneously, which can lead to a significant increase in the *sys* time. Depending on the hardware, disk I/O can become a bottleneck when many processes try to read and write simultaneously.

Writing out each command individually quickly becomes tedious. Additionally, we may want more control over how many processes are run simultaneously. To gain better control over the number of concurrent processes, we can write the commands into a temporary file called `parameters_wraa.tmp`:

```
# parameters_wraa.tmp
./run_writerandom_and_average.exe ran_A 1000
./run_writerandom_and_average.exe ran_B 5000
./run_writerandom_and_average.exe ran_C 3000
```

We then use the following script to run these commands in parallel:

```
#!/bin/bash
# run_wraa_parallel_script.sh
cat parameters_wraa.tmp | parallel -j 3
```

Note that the `-j 3` option specifies the maximum number of jobs that may run simultaneously. The value `3` does not need to match the total number of commands; it only determines how many jobs may run in parallel at any given time. `gnu_parallel` is generally flexible about whitespace, but its command-line syntax must still be used correctly. In general, increasing the number of concurrent jobs can reduce the total execution time, provided that sufficient CPU, memory, and I/O resources are available.

Now, let's put this into practice.

We will now start a 60-minute practical programming session.

3 Practical Session – Part I

Practice parallel processing

1. Write your own `writerandom_and_average.cxx` and test it in serial mode.
2. Write a `bash` script to run the program for 10 different file names with 2000 random numbers each, in serial mode, and measure the execution time.
3. Write a `bash` script to do the same using `gnu_parallel` and measure the execution time.
4. Modify your code so that it generates all random numbers, stores them in RAM, and writes them to files only at the end of the program. Test this version and measure its execution time to compare with previous results.
5. Visit https://www.gnu.org/software/parallel/parallel_tutorial.html for more information, and try running your programs in parallel using different syntaxes not covered in the lecture.

We will now continue with a 15-minute theory session.

4 Parallel threading

Threading has been available for many years, and various libraries and frameworks provide different interfaces for creating and managing threads. Common options include `Boost`, `Qt QThread`, the `C++` standard library's `std::thread`, and POSIX threads (`pthread`). Their interfaces and available features differ, and some may be more suitable than others depending on the operating system and application. For further information, see the following resources:

- POSIX Threads Programming
- C++11 Multithreading
- C++ threading library reference

For this course, we will use POSIX threads (`pthread`).

To use POSIX threads, include the appropriate header at the top of your code as usual:

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

When compiling the program, you also need to link against the POSIX threads library. With `gcc`, this is typically done using the `-pthread` option:

```
-pthread
```

Your `Makefile` should now look something like the following:

```
# This is the directory of YOUR source code
sourcedirectory=./
#These are your source codes and components
myfunctions=myfunctions.cxx
myobjects=myobjects.cxx
mymainbasic=pthreader_basic.cxx
mymain=pthreader.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=$(shell date +"%Y_%m_%d_T-%H_%M_%S" )
SFLAG=-Wall
PFLAG=-pthread
ALLFLAG=$(SFLAG) $(PFLAG)
# When a Makefile is executed, by default it tries the option "all"
all: clean functions objects main
# We tell the makefile to clean, then to compile the functions and objects, and eventually
↪  to compile main and to link it with the other components.
functions:
# Compile the functions part
        @echo Compiling $(sourcedirectory)$(myfunctions:.cxx=.o)
        $(CPP) -c -o $(myfunctions:.cxx=.o) $(SFLAG) $(sourcedirectory)$(myfunctions)
        @echo Successfully compiled $(sourcedirectory)$(myfunctions)
        @echo The unlinked compiled code is $(myfunctions:.cxx=.o)
objects:
# Compile the objects part
        @echo Compiling $(sourcedirectory)$(myobjects)
        $(CPP) -c -o $(myobjects:.cxx=.o) $(SFLAG) $(sourcedirectory)$(myobjects)
        @echo Successfully compiled $(sourcedirectory)$(myobjects)
        @echo The unlinked compiled code is $(myobjects:.cxx=.o)
main:
# Compile main and link it with functions and objects
        @echo Linking $(myobjects:.cxx=.o) and $(myfunctions:.cxx=.o) with $(mymainbasic)
```

```
36          $(CPP) -o $(addprefix run_,$(mymainbasic:.cxx=.exe)) $(mymainbasic)
            ↪  $(sourcedirectory)$(myobjects:.cxx=.o) $(sourcedirectory)$(myfunctions:.cxx=.o)
            ↪  $(ALLFLAG)
37          @echo Successfully compiled $(sourcedirectory)$(mymainbasic)
38          @echo Everything is linked and compiled into $(addprefix
            ↪  run_,$(mymainbasic:.cxx=.exe))
39          @echo Linking $(myobjects:.cxx=.o) and $(myfunctions:.cxx=.o) with $(mymain)
40          $(CPP) -o $(addprefix run_,$(mymain:.cxx=.exe)) $(mymain)
            ↪  $(sourcedirectory)$(myobjects:.cxx=.o) $(sourcedirectory)$(myfunctions:.cxx=.o)
            ↪  $(ALLFLAG)
41          @echo Successfully compiled $(sourcedirectory)$(mymain)
42          @echo Everything is linked and compiled into $(addprefix run_,$(mymain:.cxx=.exe))
43  clean:
44          @echo $(TIMESTAMP)
45          @echo "Making old/$(TIMESTAMP) directory"
46          $(shell mkdir -p old/$(TIMESTAMP) )
47          @echo "Copying the source to the old directory"
48          $(shell cp -r $(sourcedirectory)/*.cxx old/$(TIMESTAMP) )
49          $(shell cp -r $(sourcedirectory)/*.h old/$(TIMESTAMP) )
50          @echo "Moving all .exe to the old directory"
51          $(shell mv *.exe old/$(TIMESTAMP) )
52          $(shell mv *.o old/$(TIMESTAMP) )
53          @echo "Copying the Makefile to the old directory"
54          $(shell cp Makefile old/$(TIMESTAMP) )
```

The only changes compared to the previous `Makefile` example are the definition of the `PFLAG` variable on line 16, which contains `-pthread`, and its use on lines 36 and 40, respectively, through `ALLFLAG`. The latter, defined on line 17, contains both `SFLAG` and `PFLAG`. Moreover, the commands on lines 36 and 40 compile the executables for the following two examples:

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

Note that the `pthread_create` function is passed a pointer to a `pthread_t` variable (argument 1). Moreover, it requires a function that returns a `void*` (`ThreadChecker` — argument 3) and takes a single `void*` argument (`(void*)i` — argument 4). The example above uses a C-style type conversion to pass the thread index to the `ThreadChecker` function, where it is converted back. The same approach can be used to pass a pointer to an object of any kind to a function that expects `void*` as an argument.

The code above creates the threads and indicates whether each thread can be successfully assigned a task. On my computer, four logical CPUs are available, so up to four threads can execute simultaneously. If I set the number of threads in line 13 to a value greater than four, the additional threads cannot execute simultaneously and must wait for CPU resources to become available. The output of the code above is as follows:

```
Checking thread: 0
Checking thread: 1
Checking thread: 2
Checking thread: 3
```

Everything is working as expected here. However, the thread numbers may not appear in sequential order. This is because the operating system's scheduler determines when each newly created thread gets CPU time. The order in which the threads are created does not determine the order in which they are executed. For example, running the same command again may produce output such as:

```
Checking thread: 0
Checking thread: 3
Checking thread: 1
Checking thread: 2
```

Because the execution order of threads is not deterministic, the order in which the messages appear may vary from one run to another. In addition, output from different threads may become interleaved, as several threads can access cout concurrently. For example, you may occasionally see output such as:

```
Checking thread: Checking thread: 2
Checking thread: 1
0
Checking thread: 3
```

This is a simple example of why concurrent programs can produce output that is difficult to predict. Proper synchronization is required when threads need to coordinate access to shared resources, but this is beyond the scope of this course.

```
1  // pthreader.cxx (requires subdirectory "randoms/")
2  #include <iostream>
3  #include <fstream>
4  #include <sstream>
5  #include <pthread.h>
6  using namespace std;
7  void* ThreadChecker(void *thread_ID){
8    long t_id = (long)thread_ID;
9    cout << "Checking thread: " << t_id << endl;
10   return nullptr;
11 }
12 void writerandom(int variable, string filename){
13   stringstream filename_extender;
14   filename_extender << "randoms/" << filename << ".txt";
15   ofstream outputfile;
16   outputfile.open(filename_extender.str().c_str());
17   for (int i = 0; i < variable; i++){
18     outputfile << rand() << endl;
19   }
20   outputfile.close();
21 }
22 void* assign_thread_writerandom(void* thread_ID){
23   long t_id = (long)thread_ID;
24   stringstream filename_extender;
25   filename_extender << "Written_By_Thread_ID_" << t_id;
26   writerandom(1000000, filename_extender.str().c_str());
```

```
27    return nullptr;
28  }
29  void writeaverage(string inputfilename, string outputfilename){
30    stringstream inputfilename_extender;
31    inputfilename_extender << "randoms/" << inputfilename << ".txt";
32    stringstream outputfilename_extender;
33    outputfilename_extender << "randoms/" << outputfilename << ".txt";
34    ifstream inputfile;
35    inputfile.open(inputfilename_extender.str().c_str());
36    inputfilename_extender.str(string());
37    inputfilename_extender << inputfilename << ".txt";
38    ofstream outputfile;
39    outputfile.open(outputfilename_extender.str().c_str(), ios::app);
40    int temp_int = 0;
41    int sum = 0;
42    int counter = 0;
43    while(inputfile >> temp_int){
44      sum += temp_int;
45      counter++;
46    }
47    int average = sum / counter;
48    outputfile << inputfilename_extender.str() << "\t" << average << endl;
49    inputfile.close();
50    outputfile.close();
51  }
52  void* assign_thread_writeaverage(void* thread_ID){
53    long t_id = (long)thread_ID;
54    stringstream filename_extender;
55    filename_extender << "Written_By_Thread_ID_" << t_id;
56    writeaverage(filename_extender.str().c_str(),"average");
57    return nullptr;
58  }
59
60  int main(int argc, char *argv[]){
61    const int max_threads_used = 4;
62    const int mtu = max_threads_used;
63    pthread_t threads1[mtu];
64    int runcommand;
65    for (long i = 0; i < mtu; i++){
66      runcommand = pthread_create(&threads1[i], NULL, ThreadChecker, (void*)i);
67      if (runcommand){
68        cout << "Error:unable to create thread," << runcommand << endl;
69        exit(-1);
70      }
71    }
72    pthread_t threads2[mtu];
73    for (long i = 0; i < mtu; i++){
74      runcommand = pthread_create(&threads2[i], NULL, assign_thread_writerandom, (void*)i);
75      if (runcommand){
76        cout << "Error:unable to create thread," << runcommand << endl;
77        exit(-1);
78      }
79    }
80
81    pthread_t threads3[mtu];
82    for (long i = 0; i < mtu; i++){
83      runcommand = pthread_create(&threads3[i], NULL, assign_thread_writeaverage, (void*)i);
84      if (runcommand){
85        cout << "Error:unable to create thread," << runcommand << endl;
86        exit(-1);
87      }
88    }
89    for (long i = 0; i < mtu; i++){ pthread_join(threads3[i], NULL); }
90    return 0;
91  }
```

In this example, each task from the parallel-processing demonstration has been modified so that a separate thread can be assigned to each job. If you observe your system's resource monitor while the program is running, you should see that the available logical CPUs are actively utilized. Of course, you can also assign different tasks to different threads.

In this example, however, there are dependencies between the threads: the threads started on line 74 write files that need to be read by the threads started on line 83. Without synchronization, the threads on line 83 may therefore attempt to read the files before the threads on line 74 have finished writing them.

We can ensure that all threads started on line 74 have finished before continuing by adding the following in line 80:

```
80  for (long i = 0; i < mtu; i++){ pthread_join(threads[i], NULL); }
```

This ensures that the program waits until all the specified threads have completed before continuing. For the purposes of this course, we will not cover thread synchronization in more detail, but you should be aware of situations in which one task must finish before another can begin.

When multiple threads access shared resources and the outcome depends on the order in which they execute, a *race condition* can occur. If one operation must complete before another can safely proceed, the threads must be synchronized appropriately. We strongly encourage you to explore this `pthread` tutorial for a deeper understanding.

For now, let's focus on practicing what we've learned so far.

We will now start a 60-minute practical programming session.

5 Practical Session – Part II

Practice parallel threading - Extend your Packages

1. Update your compiler settings to include the `-pthread` flag where required, and recompile all of your code.
2. Write your own `pthreader.cxx` program based on the example shown in the lecture.
3. Compile and run the program, ensuring that you adjust the maximum number of threads to match your system's specifications.
4. For your functions package, create a new header file `assign_thread_functions.h` and a corresponding sources code file `assign_thread_functions.cxx`. Implement functions therein which assign a separate thread to each function in your functions package.
5. Compile your `myfunctions` package, `myobjects` package, `assign_thread_functions`, and `main` together, and test the resulting executable.

We will now conclude with a 15-minute question session.

6 Conclusion

In this block course, we have covered several semesters' worth of programming material in just ten days. We were able to achieve this because we left out many theoretical aspects that are important for a deeper understanding of programming. We can confidently say that we have covered many of the essential tools and concepts provided by `C++`. However, there are many more tools, libraries, and frameworks to explore, especially in the field of parallel programming. For example, `OpenCL` allows you to use GPUs for general-purpose computing, while `OpenGL` provides an API for GPU-accelerated graphics. There is also `CUDA`, `NVIDIA`'s proprietary platform and programming model for GPU computing on `NVIDIA` hardware.

The purpose of this course has been to provide you with practical skills in `C++` programming. However, it was not possible to cover every topic in detail. Be sure to explore online resources and continue learning as you gain more experience writing code. The `C++` language is constantly evolving, with new features, tools, and libraries being developed continuously.

I would like to congratulate all of you who have made it this far, and I hope that the skills you have learned will prove useful to you in the future.

This concludes PHY224 for 2026. Thank you very much for your attention.

Thank you everyone!