



Sample Codes in

C++11 .... C++20

# Mini Handbook of

## Parallel Computing

### Skills for Today & Tomorrow

#### (Covering Parallel Programming for HPC)

By: Pouya Ebadollahyvhed

#### 1. Table of Contents

- Difference between **Vectorization** vs **Data Parallelism** vs **Functional Parallelism**
- Difference between **Thread** vs **Process** vs **Task** vs **Job**
- **Symmetric & Asymmetric Multiprocessing** (**Shared Memory** vs **Distributed Memory**)
- Parallel Programming Techniques: **POSIX thread**, **std::thread** class, openMP, MPI & ... **OpenMP**
  - Architectures & parallel programming techniques in super-computers & HPC
- What is Intel **oneAPI**? toolkit for heterogeneous parallelism: **DPC++**, oneDPL & oneTBB
- Concepts like: **IPC**, **Mutex**, **Lock** & **Amdahl's law**
- Easy **code samples** in **C++** in Linux (RHEL), Windows & CUDA for all topics



For Thermal Reasons, Increasing Clock Frequency of CPUs / GPUs more than 4 GHz is not Feasible so to Increase Processing Power, We Need to Increase CPUs /GPUs and Their Cores Hence **We Need to Run Codes in Parallel** and the demand is increasing.

## 2. What is Parallel Computing?

The two terms: **Parallel Computing** and **Parallel Programming** are often used interchangeably and by a simple definition it means breaking a large computational task into several smaller sub-tasks (often independent or has little dependency) and then executing these sub-tasks simultaneously. In other words, dividing a large piece of code into several smaller pieces of codes and running these codes in parallel.

Since these days even on simple laptop and desktop computers, there are several CPU / GPU cores, parallel computing plays a significant role to decrease software execution time hence increases performance.

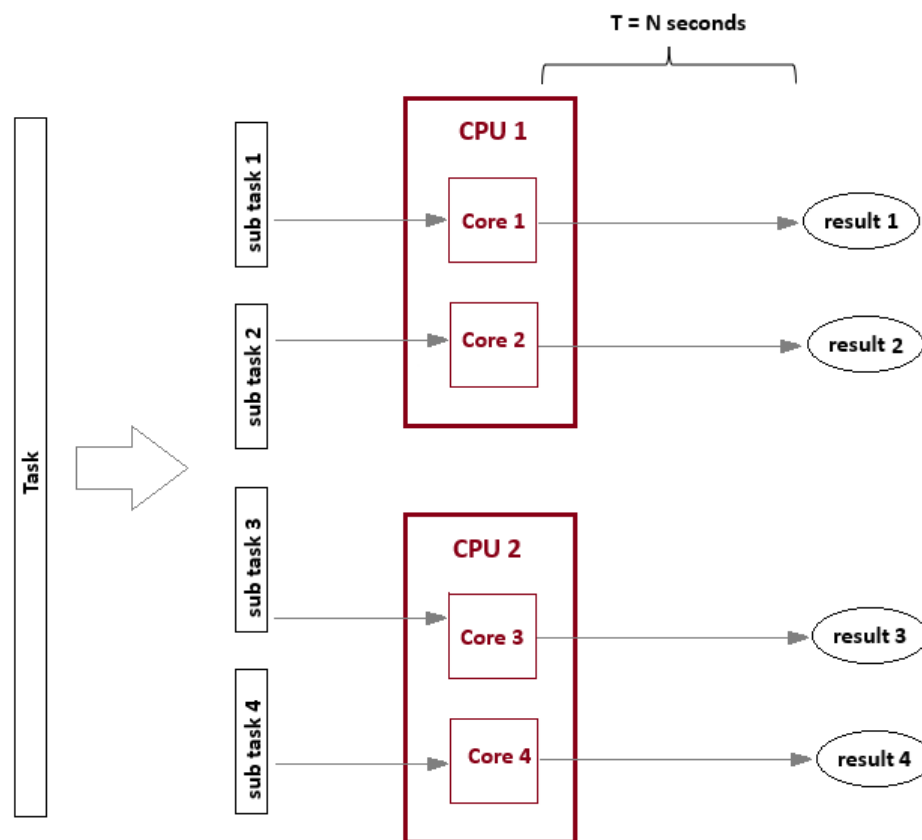


Figure 1 Running the Complete "Task" would take  $4 \times N$  seconds, while breaking it into 4 sub-tasks and running them in parallel takes only  $N$  s

### 3. Parallelism Types: Data Parallelism vs Functional Parallelism

When it comes to parallel programming, a great number of programmers think only about Functional Parallelism (like Multi-Threading) while, in reality there are 2 major types of parallelism:

- 1- **Data Parallelism:** When we want to run an algorithm (like sort) on a set of data, we can do it in 2 different ways. Run it in classic way which means running the algorithm on the entire data set (also called sequential way) OR we can do it in parallel. In this parallel way, we break the dataset into several sub-sets and then run the same algorithm on each sub-set in parallel and then we collect all the results and if it is needed, we process the results a bit more to produce the final result; this is also called Data Parallelism. Since C++17, most of algorithms (like `std::sort()`, `std::find()` and `std::merge()`) have optional 1<sup>st</sup> argument (parameter) which is called **execution policy**. If `std::execution::par` is passed as the 1<sup>st</sup> argument to these functions, it requests the algorithm to process the dataset in parallel as described earlier. Example:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++17
// On Windows    : Visual Studio (MSVC): use switch /std:c++17
//
#include <iostream>
#include <vector>
#include <execution>

int main()
{
    std::vector<int> nums{ 9, 4, 0, 3, 1, 6, -4, 7 };

    std::sort(std::execution::par, nums.begin(), nums.end()); // Data Parallel

    for(auto n : nums)
        std::cout << " " << n;
}

// Printed Result: -4 0 1 3 4 6 7 9
```

2- **Functional Parallelism**: or **Task Parallelism** is something that most of the programmers are already familiar with some forms of it. **Multi-threading** and **multi-processing** are two very popular ways of Functional Parallelism. In Functional Parallelism, programmers develop different pieces of codes and run these codes independently and in parallel although codes can exchange data and messages. A very common approach to functional parallelism is developing different functions and then run them in parallel. Each function may call some other normal functions as well. Since **C++11**, class `std::thread` is added to **C++** so developing multi-threaded codes gets easier and 100% portable (the same code can be compiled with different compilers and can run on different Operating Systems). Example:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++11
// On Windows    : Visual Studio (MSVC)
//

#include <iostream>
#include <thread>

void f1()
{
    std::cout << " from Thread 1 \n";
}

void f2()
{
    std::cout << " from Thread 2 \n";
}

int main()
{
    std::thread t1(f1); // Creating thread 1 and running it
    std::thread t2(f2); // Creating thread 2 and running it in parallel

    // Now 3 Threads are running in parallel: t1, t2 and the main thread (program)

    t1.join();          // Wait for thread 1 to finish
    t2.join();          // Wait for thread 2 to finish
}

// Printed Result: Both messages "from Thread 1" and "from Thread 2" will be
//                printed but the order of printing depends to execution and
//                is not determined (and is not fixed)
```

## 4. What is **Vectorization** and How is it Different from **Data Parallelism**?

**Vectorization** is similar to **Data Parallelism** but it is different. Modern CPUs have instruction sets which are called **SIMD**. Each of these instructions can access multiple data items when executed. Vectorization simply means compiling high level codes (written in **C/C++**) to machine code by using such instructions so when the executable (binary) code is running, one instruction can access more than 1 data item; actually, one instruction accesses several data items at the same time so it reduces the execution time and causes the whole program (code) to run faster.

**C++** supports vectorization in a built-in way since **C++20**. Two pages ago, it is mentioned that execution policy of `std::execution::par` causes C++ algorithms to run in parallel, but if programmer wants to run algorithms in vectorized way, then instead of `std::execution::par` policy, another policy named `std::execution::unseq` should be passed to the algorithm.

In the following example we changed only 1 line of code in comparison to the example of 2 pages ago.

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++20
// On Windows    : Visual Studio (MSVC): use switch /std:c++20
//

#include <iostream>
#include <vector>
#include <execution>

int main()
{
    std::vector<int> nums{ 9, 4, 0, 3, 1, 6, -4, 7 };

    std::sort(std::execution::unseq, nums.begin(), nums.end()); // vectorized

    for(auto n : nums)
        std::cout << " " << n;
}

// Printed Result: -4 0 1 3 4 6 7 9
```

## 5. What is **Process, Thread, Task** and **Job** & Their Difference?

- 1- **Thread:** Thread has clear and strong definition in programming. A thread is a sequence of instructions which can be executed independently on a computational core like a CPU or GPU core. A Thread cannot be divided to run on more than 1 computational core, so it is executed sequentially. Usually, a thread is implemented in the form of a function (or member function of a class) which can call other functions inside itself.
- 2- **Process:** Process has clear and strong definition in programming although it is also used in the context of OS. A process is an instance of running code; it can be the whole program. A Process has its own memory space which is dedicated to it by OS and is separated from memory space of other processes. A process may contain any number of threads (inside itself).
- 3- **Task:** Task doesn't have a clear definition. It is often used in scheduling context and usually points out to a unit of execution. a Task simply may mean, what is currently running on a computational node (like CPU). In some texts the term "Task" is used as synonym to Process & Thread.
- 4- **Job:** Job has a bit ambiguous definition. It is often used in scheduling and OS contexts and it points out to a complete unit of work to be executed. Usually, job comprises a group of tasks, so when a job is running, it is in the form of a group of processes. For example, a job in SLURM scheduler can be defined as: loading some modules, then running a multi-process program (code), take some logs from the output of the program (in files) and then unload modules.

Table 1 Brief Comparison Between Process and Thread

Process	Thread
A process is heavyweight	A thread is lightweight (also called LWP)
A process has its own memory space with specific security context	A thread runs inside memory space of its process so all threads of the process share the same memory
communication between processes is slower because their memory is isolated, methods: <ul style="list-style-type: none"> <li>Pipes (unnamed &amp; named or FIFO)</li> <li>Message Queues</li> <li>Sockets</li> <li>Signals</li> <li>Shared Files &amp; Memory</li> </ul>	communication between threads (of a process) is faster because they have access to each other's variables and memory space, methods: <ul style="list-style-type: none"> <li>Directly accessing to the same memory space and variables</li> </ul>
Context switching between processes is slower, since execution jumps from one memory space to another (with different security context)	Context switching between threads is faster and has lower overhead
We can assign lower priority to processes, so when they are inactive, OS moves their memory to disk and frees the memory for other processes (swapping) It provides better memory management.	When the memory (RAM) is low, multi-thread programming doesn't help to cope with this issue.
<b>To Create a Process:</b>  <code>fork()</code> in Unix / Linux <code>CreateProcess()</code> in Windows	<b>To Create a Thread:</b> <code>pthread_create()</code> in Unix / Linux <code>CreateThread()</code> in Windows <code>std::thread</code> in C++ which maps to one of above <code>std::jthread</code> in C++20 similar to above

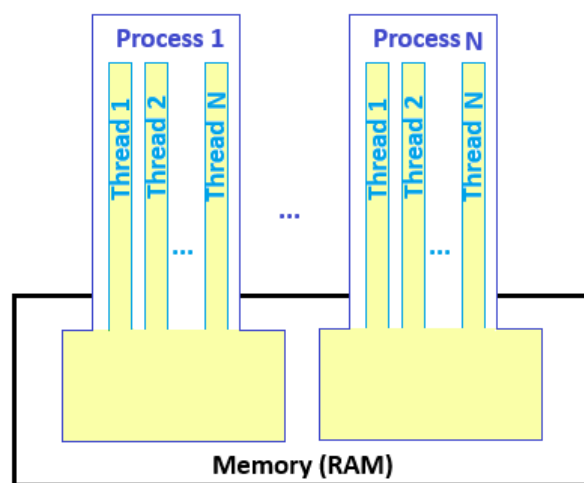


Figure 2 Running Multiple Processes and Multiple Threads inside each Process. Each Process has its own Memory Space but Threads of a Process Share the Same Memory Space

## 6. How to Create and Run a New Process (Multi-Processing)?

**C++** doesn't have built-in support for creating and running processes so we need to use OS system functions (also called system calls) to create and run processes & for this reason developing multi-process codes is OS dependent and porting a code from UNIX-like systems (including BSD and Linux) to Windows (or vice versa) is not an easy job.

In UNIX-like systems there is a system function which is called `fork()`. Once this function is called, it creates a new process and return twice. It returns with non-zero value in the main (or parent) process and for the second time it returns with a return value of 0 in the child process (newly created). By checking it's return value in the code, we can determine if we are in the main process or in the child process. The main process and the child process run in parallel.

An example for creating and running a new process in Linux / UNIX:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp
//

#include <iostream>
#include <unistd.h>

int main()
{
    if (0 == fork())
    {
        std::cout << " from Process 2 (Child) \n";
    }
    else
    {
        std::cout << " from Process 1 (Parent) \n";
    }
}

// Printed Result: Both messages "from Process 1 (Parent)" &
//                  "from Process 2 (Child)" will be printed but the order of
//                  printing depends to execution & is not determined (& not fixed)
```



Creating a new process in Windows is very different from UNIX-like systems and a bit more complicated.

In the following example a new process is created and an executable binary code is introduced to Windows to run in this process. To make it easy, we defined a string variable named `cmd[]` to keep the name of executable file, which in this example is the standard calculator of Window.

An example for creating and running a new process in Windows:

```
// Pre-requirement : None
//
// On Windows      : Visual Studio (MSVC)
//

#include <iostream>
#include <Windows.h>

int main()
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);

    ZeroMemory(&pi, sizeof(pi));

    TCHAR cmd[] = TEXT("calc");//The name of an executable file to run as new proc.

    // Start the child process.
    if (!CreateProcess(NULL, cmd, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
    {
        std::cout << " creating the New Process Failed\n";
        return -1;
    }

    // Wait until child process exits.
    WaitForSingleObject(pi.hProcess, INFINITE);

    // Close process and thread handles.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}

// Result: Creates and Runs a New Process Which is Windows Calculator
```

## 7. How to Create and Run a New Thread (Multi-Threading)?

Unlike processes, for creating, running and managing threads, **C++** has strong and full built-in support (Since **C++11**). Following are 2 examples:

```
// Sample code on page 4, is a multi-threading example for C++11
```

Following code is a bit changed version of the above example. We replaced `std::thread` class with `std::jthread` class which makes the code simpler but the latter class is introduced in **C++20**.

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++20
// On Windows    : Visual Studio (MSVC): use switch /std:c++20
//
#include <iostream>
#include <thread>

void f1()
{
    std::cout << " from Thread 1 \n";
}

void f2()
{
    std::cout << " from Thread 2 \n";
}

int main()
{
    std::jthread t1(f1); // Creating thread 1 and running it
    std::jthread t2(f2); // Creating thread 2 and running it in parallel

    // Now 3 Threads are running in parallel: t1, t2 and the main thread (program)
} // Now all 3 Threads are finished

// Printed Result: Both messages "from Thread 1" and "from Thread 2" will be
//                printed but the order of printing depends to execution and
//                is not determined (and is not fixed)
```

Although C++ has built-in classes and function to create, run and manage threads but sometimes programmers decide to directly use OS system functions (system calls) to work with threads. Following is an example of creating and running two threads in Linux:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -lpthread
//

#include <iostream>
#include <pthread.h>

void* f1(void*)
{
    std::cout << " from Thread 1 \n";
    return NULL;
}

void* f2(void*)
{
    std::cout << " from Thread 2 \n";
    return NULL;
}

int main()
{
    pthread_t tID1, tID2;

    pthread_create(&tID1, NULL, &f1, NULL); // Creating thread 1 and running it
    pthread_create(&tID2, NULL, &f2, NULL); // Creating thread 2 and running it

    // Now 3 Threads are running in parallel: t1, t2 and the main thread (program)

    pthread_join(tID1, NULL); // Wait for thread 1 to finish
    pthread_join(tID2, NULL); // Wait for thread 2 to finish
}

// Printed Result: Both messages "from Thread 1" and "from Thread 2" will be
//                  printed but the order of printing depends to execution and
//                  is not determined (and is not fixed)
```

Above example is the re-written version of examples on pages 4 and 10 but using Linux (UNIX) system functions. The performance and the outputs are identical to these 2 examples.

Following is the example of previous page re-written for Windows. It creates and runs 2 threads using **Windows API** functions. So, the following example and examples of pages 4, 10 and 11 are working in the same way.

```
// Pre-requirement : None
//
// On Windows      : Visual Studio (MSVC)
//

#include <iostream>
#include <Windows.h>

DWORD WINAPI f1(LPVOID lpParam)
{
    std::cout << " from Thread 1 \n";
    return 0;
}

DWORD WINAPI f2(LPVOID lpParam)
{
    std::cout << " from Thread 2 \n";
    return 0;
}

int main()
{
    HANDLE h_thread[2];

    // Creating thread 1 and running it
    h_thread[0] = CreateThread(NULL, 0, f1, NULL, 0, NULL);
    h_thread[1] = CreateThread(NULL, 0, f2, NULL, 0, NULL);

    // Now 3 Threads are running in parallel: t1, t2 and the main thread (program)

    // Wait for threads 1 and 2 to finish
    WaitForMultipleObjects(2, h_thread, TRUE, INFINITE);

    CloseHandle(h_thread[0]);
    CloseHandle(h_thread[1]);
}

// Printed Result: Both messages "from Thread 1" and "from Thread 2" will be
//                  printed but the order of printing depends to execution and
//                  is not determined (and is not fixed)
```

So, we have 2 ways to create and manage threads:

- Using C++ built-in classes and functions
- Using OS system functions (system calls)

But which one is better and more efficient?

The answer is simple: there is no significant difference. Actually, C++ built-in functions are being mapped to OS functions when they are compiled.

Using C++ built-in classes and functions has one great advantage: **The code will be portable** to any other compiler and to any other platform (any other OS and any other machine).

Using OS system calls also has a small advantage for professional coders: it gives a bit more flexibility and more information about threads but the **execution time (performance) remains the same** in comparison to the situation when using C++ built-in functions to create threads.

## 8. Communication Between Threads

### 8.1. Sharing Variables, Arrays, Objects and Buffers Between Threads

Since threads (of a process) are running in the same memory space they can have access to the same entities (variables, arrays, objects and buffers); so, they can read / write the same entities easily. In the following example, we have an array of 4 numbers and we created 2 threads, each thread calculates summation of 2 elements of the array and in the main thread (program body) these 2 sub-totals are being added together to print the grand total (summation of all 4 numbers in the array):

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++11
// On Windows    : Visual Studio (MSVC)
//

#include <iostream>
#include <thread>

int numbers[] = { 33, 12, -5, 15 };
int subtotals[2];

void f1()
{
    subtotals[0] = numbers[0] + numbers[1];
}

void f2()
{
    subtotals[1] = numbers[2] + numbers[3];
}

int main()
{
    std::thread t1(f1); // Creating thread 1 and running it
    std::thread t2(f2); // Creating thread 2 and running it in parallel

    t1.join();          // Wait for thread 1 to finish
    t2.join();          // Wait for thread 2 to finish

    int grand_total{ subtotals[0] + subtotals[1]};

    std::cout << "Sum of all numbers is: " << grand_total << std::endl;
}
// Printed Result: 55
```

The code of previous page has two big disadvantages:

- 1- Both `f1()` and `f2()` functions are doing the same job (summing 2 integers), so the code is repeated which is not according to software engineering concepts, so let's remove one of these functions and keep only one function named `f()` to calculate the summation. We still can create 2 threads (run in parallel) but both threads will run this function (there will be 2 instances of it).
- 2- There are 2 integer arrays defined in the global scope which violates data encapsulation, so let's move them inside scope of the `main()` function and then pass their references to threads by function arguments. After fixing above problems, our code will look like this:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++11
// On Windows    : Visual Studio (MSVC)
//

#include <iostream>
#include <thread>

void f(int *numbers, int *sum)
{
    *sum = *numbers + *(numbers+1);
}

int main()
{
    int nums[] = { 33, 12, -5, 15 };
    int subTot[2];

    std::thread t1(f, &(nums[0]), &(subTot[0])); // Creating thread 1 & running it
    std::thread t2(f, &(nums[2]), &(subTot[1])); // Creating thread 2 & running it

    t1.join();           // Wait for thread 1 to finish
    t2.join();           // Wait for thread 2 to finish

    int grand_total{ subTot[0] + subTot[1] };
    std::cout << "Sum of all numbers is: " << grand_total << std::endl;
}
// Printed Result: 55
```

In the example of previous page, we created 2 threads but binary code of both threads is the same code of functions `f()`, so 2 instances of this function (= 2 copies in the memory) will be created and run (in parallel) but still they are the same function so for example these instances of function `f()` share the same static variables (as a rule from classic C). Example:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++11
// On Windows    : Visual Studio (MSVC)
//

#include <iostream>
#include <thread>

void f(int* numbers, int* sum, int* counter)
{
    static int this_function_called_howmany_times{};

    *counter = ++this_function_called_howmany_times;

    *sum = *numbers + *(numbers + 1);
}

int main()
{
    int nums[] = { 33, 12, -5, 15 };
    int subTot[2], counter{};

    std::thread t1(f, &(nums[0]), &(subTot[0]), &counter); // counter will be 1
    std::cout << "Thread 1 is created" << std::endl;

    std::thread t2(f, &(nums[2]), &(subTot[1]), &counter); // counter will be 2

    t1.join();
    t2.join();

    std::cout << "function f() is called " << counter << " times" << std::endl;

    int grand_total{ subTot[0] + subTot[1] };
    std::cout << "Sum of all numbers is: " << grand_total << std::endl;
}

// Printed Result: function f() is called 2 times
//                  grand_total = 55
```



## 8.2. Mutexes & Locks for Exclusive Access of Threads to Objects & Vars

As mentioned earlier threads (of a process) have access to the same memory space so they can have access to the same entities (objects, arrays, variables, buffers and so on). **This common access to entities and sharing them is OK until all threads are only reading them** but once they try to write in these memory locations and for example change values of the same variables, the problem begins. Example:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++11
// On Windows    : Visual Studio (MSVC): use switch /std:c++11
//
#include <iostream>
#include <thread>

void f(int* numbers, int* sum)
{
    int total_this_thread = *sum;

    total_this_thread += *numbers;
    total_this_thread += *(numbers + 1);

    std::cout << "Total of this thread is: " << total_this_thread << std::endl;

    *sum = total_this_thread;
}

int main()
{
    int numbers[] = { 33, 12, -5, 15 };
    int grandTot {};

    std::thread t1(f, &(numbers[0]), &grandTot); // Creating thread 1 & running it
    std::thread t2(f, &(numbers[2]), &grandTot); // Creating thread 2 & running it

    t1.join();           // Wait for thread 1 to finish
    t2.join();           // Wait for thread 2 to finish

    std::cout << "Sum of all numbers is: " << grandTot << std::endl;
}

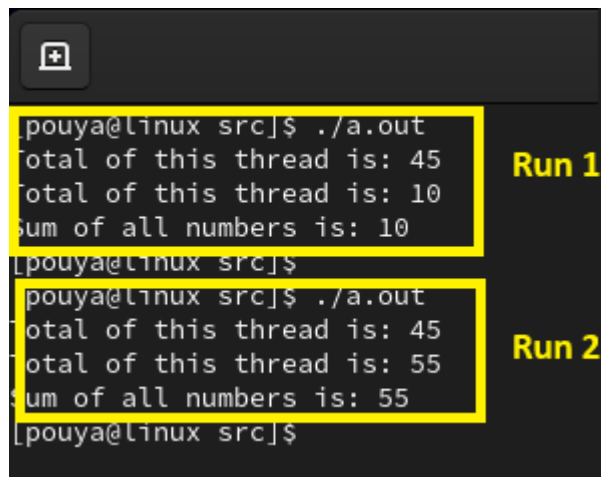
// Result (grandTot) can be different in different runs; it can be 10 or 45 or 55
```

Code of the previous page is a bit changed version of the codes on pages 14, 15 and 16. The only important difference is that in the codes of pages 14, 15, and 16:

Each thread was calculating a sub-total and writing it in a separate memory location (separate elements of `subTot[]` array). It is not efficient; if we have like 1000 threads then we need to create an array of 1000 elements and then the main thread needs to calculate summation of these 1000 elements (numbers) so it is waste of memory space and waste of execution time.

A better solution is that we define only 1 variable like `grandTot` to store the grand total and all threads just calculate sub-totals and add to this variable, so each thread calculates summation of its numbers and then updates `grandTot`. It is what we tried to do in the example of the previous page.

Following screenshot shows output of an actual run of the code of the previous page:



```
pouya@linux src]$ ./a.out
Total of this thread is: 45
Total of this thread is: 10
sum of all numbers is: 10
[pouya@linux src]$
pouya@linux src]$ ./a.out
Total of this thread is: 45
Total of this thread is: 55
sum of all numbers is: 55
[pouya@linux src]$
```

As it can be seen, the **output is not deterministic** and in each run the printed output may change, the reason is simple: we have 2 running threads (in parallel and most surely simultaneously), and both are **racing** to read from and write to the same memory location (variable `grandTot`).

In “**Run 1**” of the previous page screenshot the sequence of events is like:

- 1- Thread 1 reads **grandTot** variable (inside thread: **sum**), the value is “0”
- 2- Thread 1 calculated its total (variable **total\_this\_thread**), the value is “45”
- 3- Thread 2 reads **grandTot** variable (inside thread: **sum**), the value is “0”
- 4- Thread 2 calculated its total (variable **total\_this\_thread**), the value is “10”
- 5- Thread 1 replaces **grandTot** (inside thread: **sum**), with its calculated value which is “45”
- 6- Thread 2 replaces **grandTot** (inside thread: **sum**), with its calculated value which is “10”
- 7- **grandTot** is “10” in the main thread (body of the **main** function) after both threads are done

In “**Run 2**” of the previous page screenshot the sequence of events is like:

- 1- Thread 1 reads **grandTot** variable (inside thread: **sum**), the value is “0”
- 2- Thread 1 calculated its total (variable **total\_this\_thread**), the value is “45”
- 3- Thread 1 replaces **grandTot** (inside thread: **sum**), with its calculated value which is “45”
- 4- Thread 2 reads **grandTot** variable (inside thread: **sum**), the value is “45”
- 5- Thread 2 calculated its total (variable **total\_this\_thread**), the value is “55” (= 10 + 45)
- 6- Thread 2 replaces **grandTot** (inside thread: **sum**), with its calculated value which is “55”
- 7- **grandTot** is “55” in the main thread (body of the **main** function) after both threads are done

Although the same code was executed in both “Run 1” and “Run 2” but as it can be seen the results are very different because in “Run 1” thread 1 was a bit slower (could be because the CPU core which was executing thread 1 was busy with some other tasks or the OS scheduler) so “green line” happened before “blue line” and in “Run 2”, thread 2 was a bit slower so “blue line” happened before “green line” and this change, totally changed the final result.

To prevent problems similar to the problem described in the last 3 pages, programmers use Mutexes and Locks. Let's jump into an example and then describe how it works:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -std=c++11
// On Windows    : Visual Studio (MSVC): use switch /std:c++11
//

#include <iostream>
#include <mutex>
#include <thread>

void f(int* numbers, int* sum)
{
    static std::mutex m;
    int total_this_thread{};

    total_this_thread += *numbers;
    total_this_thread += *(numbers + 1);

    // The first thread reaching here, locks the lock (myLock) and continues running
    // All of the next threads reaching here will wait till the lock, is being unlocked
    std::unique_lock<std::mutex> myLock(m);

    total_this_thread += *sum;

    std::cout << "Total of this thread is: " << total_this_thread << std::endl;

    *sum = total_this_thread;

    // When a thread unlocks a lock, one another thread (if wants) can lock it again
    myLock.unlock();
}

int main()
{
    int numbers[] = { 33, 12, -5, 15 };
    int grandTot{};

    std::thread t1(f, &(numbers[0]), &grandTot); // Creating thread 1 & running it
    std::thread t2(f, &(numbers[2]), &grandTot); // Creating thread 2 & running it

    t1.join();           // Wait for thread 1 to finish
    t2.join();           // Wait for thread 2 to finish

    std::cout << "Sum of all numbers is: " << grandTot << std::endl;
}

// Printed Result: grandTot = 55 (always)
```

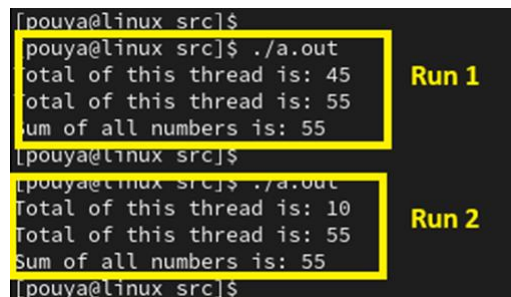
How the code of previous page works:

This code uses an old computer engineering concept named **Mutexes** and **Locks**. Since **C++11**, built-in support has been added to **C++**, to support mutexes and locks.

The rules for working with mutexes and locks are simple (in the scope of our example code):

- 1- Mutex is an object instantiated from class `std::mutex`
- 2- Any mutex object can be locked and unlocked (by an object of type `std::unique_lock`)
- 3- When a mutex object is created it is in unlocked state
- 4- A mutex object can be **locked only once**
- 5- When a code tries to lock a mutex
  - a. If the mutex was unlocked, it will be locked and the execution will continue normally
  - b. If the mutex was locked before, the execution waits at that point (forever) until the mutex is being unlocked (for example by another thread), then locks it and continues execution

Following screenshot shows output of an actual run of the code of the previous page. As it can be seen, like the code on page 17 (which was not using mutexes and locks) sometimes Thread 1 is faster than Thread 2 and sometimes slower but the output (value of **grandTot**) is always correct, because each thread calculates sub-total then enters to a locked block of code where it reads value of **grandTot** variable, adds sub-total to it and then updates the value of **grandTot** and during this operation other threads are not allowed to read/write **grandTot** variable, so only 1 thread can work with it (at a time).



```
[pouya@linux src]$  
[pouya@linux src]$ ./a.out  
total of this thread is: 45  
total of this thread is: 55  
sum of all numbers is: 55  
[pouya@linux src]$  
[pouya@linux src]$ ./a.out  
Total of this thread is: 10  
Total of this thread is: 55  
Sum of all numbers is: 55  
[pouya@linux src]$
```

## 9. Multi-threading on GPU

All of the example-codes from the beginning of this writing to this point are codes developed to run on CPU cores but the following is a different example to show how to run a multi-threaded code on GPU:

```
// Pre-requirement : Installing CUDA and integrating it to Visual Studio
//
// On Windows      : Visual Studio,
//                  Create CUDA type project or add CUDA to project:
//                  Right Click on Project in the Solution Explorer then
//                  Built Dependencies | Built Customizations... | check CUDA
// Note            : file name should have .cu extension

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>

__global__ void f_in_GPU(int* nums_in_GPU, int *subTot_in_GPU)
{
    int threadNo = threadIdx.x;
    subTot_in_GPU[threadNo] = nums_in_GPU[threadNo*2]+nums_in_GPU[threadNo*2 + 1];
}

int main()
{
    int nums[] = { 33, 12, -5, 15 }, subTot[2], *nums_in_GPU, *subTot_in_GPU;
    int sNums{std::size(nums)*sizeof(int)}, sSubTot{std::size(subTot)*sizeof(int)};

    if (cudaSuccess != cudaSetDevice(0)) return -1; //Select a GPU to run code on it
    if (cudaSuccess != cudaMalloc((void**)&nums_in_GPU, sNums)) return -2;
    if (cudaSuccess != cudaMalloc((void**)&subTot_in_GPU, sSubTot)) return -3;

    if (cudaSuccess != cudaMemcpy(nums_in_GPU, nums, sNums, cudaMemcpyHostToDevice))
        return -4;

    f_in_GPU <<<1, 2 >>> (nums_in_GPU, subTot_in_GPU); // creates 1 * 2 = 2 THREADS

    if (cudaSuccess != cudaDeviceSynchronize()) return -5; // Wait threads finish

    if (cudaSuccess != cudaMemcpy(subTot, subTot_in_GPU, sSubTot, cudaMemcpyDeviceToHost))
        return -6;

    cudaFree(nums_in_GPU);
    cudaFree(subTot_in_GPU);

    cudaDeviceReset();

    int grand_total{ subTot[0] + subTot[1] };

    std::cout << "Sum of all numbers is: " << grand_total << std::endl;
}
// Printed Result: 55
```

The code of the previous page runs only on Nvidia GPUs. It is written by using CUDA library. CUDA is a library provided by Nvidia to develop multi-thread codes to run on GPUs of this company.

CUDA library can be freely downloaded from<sup>1</sup>: <https://developer.nvidia.com/cuda-downloads>

Code of the previous page is re-written version of the codes on pages 14, 15 and 16 to run on GPU.

Here are some changes & comments about the code:

- 1- The function `f_in_GPU()` has the same functionality of `f()` in the codes of pages 14, 15 and 16
  - a. `__global__` identifier shows that the function is called from CPU code but runs on GPU
  - b. `__device__` identifier shows that the function is called from GPU code and runs on GPU
  - c. `__host__` identifier shows that the function is called from CPU code and runs on CPU
- 2- GPUs have their own memory (usually installed on Graphics Card), it is called “**Device Memory**”
- 3- The memory which is installed on motherboard (mainboard) and used by CPU is called “**Host Memory**”
- 4- To run a code on GPU, all of the GPU accessed variables, arrays & objects should be in the Device Memory
  - a. To store a variable, array or object in the “Device Memory” we need to do memory allocation
  - b. `cudaMalloc()` allocates memory in the Device Memory
  - c. We can copy variables, arrays, buffer & objects from host memory to device memory & vice versa
  - d. `cudaMemcpy()` copies a memory buffer from device to host memory & vice versa
  - e. `cudaFree()` frees previously allocated buffer in the Device Memory
- 5- Unlike CPUs, Nvidia GPUs comprises of several SMs (Stream Processors) and Each SM has several cores
  - a. You may consider SM as a small processor
  - b. For example, my machine has 36 SMs & each SM has 128 cores so in total I have 4608 cores
  - c. In CUDA architecture multi-thread codes comprises of Thread Blocks and Threads per Block

---

<sup>1</sup> **Note:** Prior to installing CUDA, you need to install proper driver for your GPU (= Video Adaptor / Graphics Card). Usually installing Nvidia drivers on Windows is straight forward also on some distributions of Linux like RHEL it is easy but installing it on some distributions like Debian is a bit challenging.

6- Creating threads in CUDA is simple, and can be done like this:

- a. `function_name <<<N, M>>> (arg1, arg2, ...)`
- b. Above N is number of thread blocks and M is number of threads per block
- c. Above total number of created threads will be  $(N * M)$
- d. Maximum number of N is 1024

7- `cudaDeviceSynchronize()` is similar to `join()` function, it waits for all threads to finish execution

8- function `cudaSetDevice()` selects which GPU the code should run on.

- a. It is usually the first function which is called in CUDA codes
- b. This function has one integer argument where “0” means 1<sup>st</sup> GPU, “1” means 2<sup>nd</sup> GPU and so on

Following table shows a simple comparison between running codes on CPUs, GPUs and NPUs.

Table 2 Comparison of Processor Types

Processor	Major Market Players	Advantage / Specification
CPU	Intel, AMD	<b>Low Latency</b>  Cores of desktop versions: typically, 4 – 20 Cores of server versions: typically, 20 - 150
GPU	Nvidia, AMD	<b>High Throughput</b>  Cores of desktop versions: Thousands Cores of datacenter versions: Tens of Thousands
NPU	Intel, AMD	<b>Low Power</b>
FPGA	Intel (former ALTERA) AMD (former XILINX)	Unlike all other processors which are “Sequential Logic Circuits”, FPGAs are “Combinational Logic Circuit”, so designers can actually design a circuit to do a specific job using design languages like VHDL or VERILOG



## 10. Communication Between Processes (IPC)

Unlike Threads (of a process) which run in the same memory space, processes run in different and isolated memory spaces so they cannot access the same variables, arrays and objects in a way that threads are accessing and sharing with each other.

Processes can still communicate with each other and pass information to each other but with different methods. These methods are discussed under the term IPC (Inter Process Communication).

Later in this chapter we will have a quick review on the most common and widely used IPC techniques in UNIX-like systems. Details and more professional materials can be found in the following books which are also listed on the last page of this wiring:

- 1- Richard Stevens: **UNIX Network Programming, Volume 2: Interprocess Communications** 2<sup>nd</sup> edition. Prentice Hall, 1999.
- 2- Richard Stevens & Stephen A. Rago: **Advanced Programming in the UNIX Environment**. Pearson Addison Wesley, May 2013.

Writing Multi-Process codes in Windows is possible but is Not common, so in this chapter we review IPC techniques and methods in UNIX like Operating Systems (like Linux and BSD) only; although Windows has similar methods too.

Following example, demonstrates how working with local and global variables (even through their references) are different in processes (in comparison to threads):

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp
//

#include <iostream>
#include <unistd.h>

int i;

void f(int* arg)
{
    (*arg)++;
    std::cout << "Inside function, i= " << ++i << ", arg= " << *arg << std::endl;
}

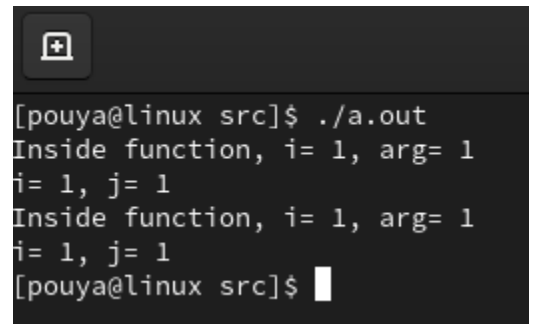
int main()
{
    int j{};

    i = 0;

    if (0 == fork())
        f(&j);
    else
        f(&j);

    std::cout << "i= " << i << ", j= " << j << std::endl; // will be printed twice
}
```

Opposite screenshot shows output of an actual run of the above code. As it can be seen both **i** and **j** variables are incremented just once. The reason is that once a new process is created (by **fork**), it creates a new memory space identical to the main



```
[pouya@linux src]$ ./a.out
Inside function, i= 1, arg= 1
i= 1, j= 1
Inside function, i= 1, arg= 1
i= 1, j= 1
[pouya@linux src]$
```

process and copies memory of the main process to this newly created memory and then each process runs in their own memory space independently so **the i in process 1 is different from the i in process 2.**

## 10.1. Pipes

Working with pipes is easy and pretty much similar to working with files. UNIX / LINUX System function (system call) of `pipe()` creates a pipe and returns a pair of file descriptors. The 1<sup>st</sup> descriptor is for “reading only” and the 2<sup>nd</sup> one is for “writing only”. As it can be seen pipes are **unidirectional**. Writing to & Reading from pipes can be easily done by standard functions like `write()` & `read()`. Example:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp
//

#include <iostream>
#include <cstring>
#include <unistd.h>

int main()
{
    int pipe_fd[2];

    if (pipe(pipe_fd) < 0) // a pipe is created (with 2 descriptors)
        return -1;

    pid_t pid{ fork() }; // a new process is created
    if (pid < 0)
        return -2;

    if (0 == pid)
    {
        close(pipe_fd[0]);
        char msg[]{ "a Message from Process 1 (Child Process)" };

        write(pipe_fd[1], msg, std::strlen(msg)); // writing to pipe
    }
    else
    {
        const int MAXLEN{ 100 };
        char buffer[MAXLEN];

        close(pipe_fd[1]);
        ssize_t lenActual{ read(pipe_fd[0], buffer, MAXLEN) }; // reading from pipe

        buffer[lenActual] = 0;
        std::cout << buffer << std::endl;
    }
    std::cout << "Done \n";
}
// Printed Result on the Screen: "a Message from Process 1 (Child Process)"
```

## 10.2. FIFO (Named Pipes)

As it is mentioned in the previous section, pipes are pretty much similar to files but to open (or create) them we don't need to mention any names, so we can say **pipes are unnamed**. Not having a name is OK until we are using pipes inside a program, in other words when 2 processes have common ancestor (created inside the same program), pipes can be used with no problem (like the example of previous page) but if 2 processes belong to two different programs (2 different executable files), using pipes will not be possible, in this case we need to use FIFOs. FIFOs are similar to pipes but they have names (like files).

Having names makes FIFOs more similar to files:

- They can be opened like files (with a name and a path)
- They can be read like files
- They can be written like files

A FIFO can be created inside the code by calling `mkfifo()` system function (example of the next page) or it can be even created inside shell by `mkfifo` command. In the example of the next page, we create a FIFO with the name (and path) of `/tmp/my_fifo`. After 1<sup>st</sup> run of the program of the next page, by using `ls /tmp` shell command, the FIFO will be listed.

Since FIFOs have names, they can be opened and used inside processes regardless processes have common ancestors or no. Processes those want to use FIFOs just need to know their paths (like files).

Example of the next page, shows how to create and use a FIFO between 2 processes. It is a re-written version of the example of the previous page (for pipes):

```

// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp
//

#include <iostream>
#include <cstring>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    int fifo_fd;

    if (mkfifo("/tmp/my_fifo", S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH) < 0)
        std::cout << "Failed to Create FIFO, maybe it is already created \n";

    pid_t pid{ fork() };
    if (pid < 0)
        return -1;

    if (0 == pid)
    {
        if ((fifo_fd = open("/tmp/my_fifo", O_WRONLY)) < 0)
            return -2;

        char msg[]{ "a Message from Process 1 (Child Process)" };

        write(fifo_fd, msg, std::strlen(msg));
        close(fifo_fd);
    }
    else
    {
        const int MAXLEN{ 100 };
        char buffer[MAXLEN];

        if ((fifo_fd = open("/tmp/my_fifo", O_RDONLY)) < 0)
            return -3;

        ssize_t lenActual{ read(fifo_fd, buffer, MAXLEN) };

        buffer[lenActual] = 0;
        std::cout << buffer << std::endl;

        close(fifo_fd);
    }
    std::cout << "Done \n";
}
// Printed Result on the Screen: "a Message from Process 1 (Child Process)"

```

## 10.3. Message Queues

Although “Pipes” & “FIFOs” are popular and easy to use but they have one small problem: in the time of communication, both sender and receiver processes should be in running mode (not terminated) so they are online communication methods while Message Queues can be used both in online & offline modes.

Example:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp
//

#include <iostream>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msgBuffer // structure for message to pass to queue
{
    long msgType;
    char msgText[200];
};

int main()
{
    if (0 == fork())
    {
        key_t key{ ftok("/home", 22) }; // generating a unique key
        int msgid{ msgget(key, 0666 | IPC_CREAT) }; // creating a message queue

        msgBuffer m1{ 2, "Hi From Process 1" }; // the message to sent
        msgsnd(msgid, &m1, sizeof(m1), 0); // sending the message

        std::cout << "Message Sent: " << m1.msgText << std::endl;
    }
    else
    {
        key_t key{ ftok("/home", 22) }; // generating a unique key
        int msgid{ msgget(key, 0666 | IPC_CREAT) }; // creating a message queue

        msgBuffer m2;
        msgrcv(msgid, &m2, sizeof(m2), 2, 0); // receiving the message

        std::cout << "Message Received: " << m2.msgText << std::endl;

        msgctl(msgid, IPC_RMID, NULL); // destroy the message queue
    }
}

// Result: "Hi From Process 1" will be printed twice: from sender & receiver
```

As the name indicates, a “Message Queue” is a queue (or better say a linked-list) for messages and this queue is handled by OS kernel. Since it is handled by OS kernel, sender can send a number of messages and quit (terminate execution) and after some time receiver can access the queue and still receive messages (offline communication). This gives a **great advantage** to “Message Queues” in comparison to “pipes” and “FIFOs”. [You can see all of the message queues on your machine by ipcs shell command.](#)

**Another advantage** of “Message Queues” is that, each message has a “Message Type” (long msgType in the example of the previous page), when sender sends a message, it can set “message type” to any number and receiver can ask the kernel to receive only messages with this type. It is an easy but efficient filtering. It is useful when multiple processes send and receive different types of messages to and from the same queue.

**Disadvantage** of “Message Queues” in comparison to “pipes” and “FIFOs” is that their overhead is more and working with them is not as easy as “pipes” & “FIFOs”.

To work with “Message Queues” as it can be seen in the example of the previous page:

- A key is generated by calling `ftok` system function, it takes any existing path and a project-id number (can be any number) and combines them to make a unique identifier which is called key
- The generated key in the previous step is passed to `msgget` system function, to create (or open if created before) the message queue
- After the message queue is created (or opened), functions like `msgsnd` & `msgrcv` can be used to send and receive messages.
- `msgctl` system function can be used to send commands to the queue, in the example of the previous page we used this function to remove the queue (at the end of the code)
- Messages has a fixed format: first a long number which indicated message type (it is just a number and the meaning is defined by programmer) and then the message itself (any size).

## 10.4. Shared Memory

Shared Memory is an IPC mechanism provided by OS Kernel. Example:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp
//

#include <iostream>
#include <chrono>    // only for sleep function
#include <thread>    // only for sleep function
#include <cstring>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    const int SizeSMS { 1024 }; // SMS stands for      Shared Memory Segment

    if (0 == fork())
    {
        key_t key{ ftok("/home", 23) };           // generate a unique key
        int shmid{ shmget(key, SizeSMS, 0644 | IPC_CREAT) }; //allocates SMS

        if (-1 == shmid) return -1;

        char* data1{ (char*)shmat(shmid, NULL, 0) }; // attach SMS to a pointer
        if((void*)-1 == data1) return -2;

        std::strcpy(data1, "Hi From Process 1");    // write to SMS

        shmdt(data1);                             // detach pointer from the SMS
    }
    else
    {
        // wait for other process to finish writing to SMS
        std::this_thread::sleep_for(std::chrono::milliseconds(80));

        key_t key{ ftok("/home", 23) };           // generate a unique key
        int shmid{ shmget(key, SizeSMS, 0644 | IPC_CREAT) }; //allocates SMS

        if (-1 == shmid) return -1;

        char* data2{ (char*)shmat(shmid, NULL, 0) }; // attach SMS to a pointer
        if((void*)-1 == data2) return -2;

        std::cout << data2 << std::endl; // read from SMS

        shmdt(data2);                             // detach pointer from the SMS
        shmctl(shmid, IPC_RMID, NULL);             // destroy the shared memory segment
    }
}

// Result: "Hi From Process 1" will be printed from the receiver process
```



**Although different processes run in different memory spaces but kernel of UNIX-like operating systems (like Linux and BSD) provides a mechanism to allocate a segment of memory and give access to different processes so these processes can have access to this memory segment at the same time.**

Shared memory mechanism is the fastest way of communication between processes but it has one big disadvantage: “synchronization between processes”.

We already discussed about “synchronization” problem between threads in section 8.2 of this writing. We have the same problem here too, since “shared Memory” is a resource so reading and writing of processes from/to it should be managed (synchronized). In the example of the previous page, we used very simple and handy method for the synchronization: the reader process waits 80 milliseconds before reading the shared memory and the writer process must finish its writing job withing this 80 milliseconds period.

In the next sections of this writing (10.5), we will review better and more professional mechanisms for synchronization (since processes run in different memory spaces, we cannot use mutexes and locks in the way that we discussed in section 8.2).

To work with “Shared Memory” as it can be seen in the example of the previous page:

- A key is generated by calling **ftok** system function. This function is already discussed 2 pages ago
- The generated key in the previous step is passed to **shmget** system function, to allocate (or open) a shared memory segment
- **shmat** system function, attaches the allocated memory to a local pointer
- The local pointer can be used as a normal C++ pointer to read from and write to memory
- After finishing the entire shared memory operations, the pointer is detached by **shmdt**
- **shmctl** system function can be used to send commands to the shared memory, in the example of the previous page we used this function to destroy the memory segment (at the end of the code)

## 10.5. Semaphores

Semaphores are pretty much similar to Mutexes (discussed in section 8.2), let's start with an example:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp    (older systems may need -lrt)
//

#include <iostream>
#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>

int main()
{
    if (0 == fork())
    {
        sem_t* sem{ sem_open("/mySem", O_CREAT, 0644, 1) };
        if (NULL == sem) return -1;

        std::cout << "Inside Process 1" << std::endl;

        sem_wait(sem); // Beginning of Critical Section, decrement the semaphore
        std::cout << "Process 1 - Inside Critical Section Begining\n";

        sleep(4);

        std::cout << "Process 1 - Inside Critical Section End\n";

        sem_post(sem); // End of of Critical Section, increment the semaphore
        sem_close(sem);

        sem_unlink("/mySem"); // Ask to remove the semaphore
    }
    else
    {
        sem_t* sem{ sem_open("/mySem", O_CREAT, 0644, 1) };
        if (NULL == sem) return -1;

        std::cout << "Inside Process 2" << std::endl;

        sem_wait(sem); // Beginning of Critical Section, decrement the semaphore
        std::cout << "Process 2 - Inside Critical Section Begining\n";

        sleep(6);

        std::cout << "Process 2 - Inside Critical Section End\n";

        sem_post(sem); // End of of Critical Section, increment the semaphore
        sem_close(sem);
    }
}

// Result: Only 1 process immediately enters to its critical section,
//          The other process waits at the beginning of its critical section,
//          Once the 1st process exits its own critical section,
//          Then the 2nd process enters to its critical section
```

Semaphore is an old computer engineering concept. Semaphores are entities to control synchronization between processes (& threads). Semaphores are pretty much similar to Mutexes (discussed in section 8.2). By easier definition semaphore is just a number which initially can be set to any value and then it can be incremented (++) or decremented (--).

Software developers use semaphores to control synchronization between processes to access to shared resources like to Shared Memory (discussed in section 10.4). Piece of the code which access shared resources are put in a block which is called “Critical Section” and access to this “Critical Section” is controlled by a Semaphore (or a Mutex in multi-threaded codes).

How the code of previous page works:

- We create a semaphore by **sem\_open** function (**O\_CREAT** argument). Each semaphore has a unique name. When this function is called, if the semaphore doesn't exist, it will create the semaphore and initializes it (here to **1**), but if the semaphore already exists, it just opens it (and ignores its initialization). So, in our example only one of the processes creates the semaphore and then the other one only opens it
- At the beginning of the critical section, we have **sem\_wait** function. If value of semaphore is positive, this function decreases its value and continues the execution (to next lines) but if the value is zero, this function waits at this point (forever) till another process (or thread) increases value of the semaphore
- At the end of the critical section, we have **sem\_post** function. It increments semaphore value.
- Function **sem\_close** closes the semaphore, so usually it is the last function each process calls
- Function **sem\_unlink** destroys the semaphore. If it is in use, once all processes close it, it will be destroyed. It is enough that only one of the processes destroys the semaphore

As it can be seen Semaphores are pretty much similar to Mutexes but they have some differences as following:

- **C++** has built-in support for Mutexes since **C++11** (discussed in section 8.2), but to use semaphores, we need to use some libraries; in the example of this section, we used POSIX semaphores which is supported in UNIX-like systems (like Linux and BSD)
- Mutexes (C++ built-in) can be used only in multi-threaded codes. Since multi-process codes have separate and isolated memory spaces, they cannot be used here but semaphores can be used both in multi-threaded and multi-process codes.
- Although semaphores can be used in multi-threaded codes but it is not recommended because C++ built-in Mutexes are portable and more flexible
- The biggest difference between Mutexes and Semaphores is that, Semaphores have values and if it is set to N, then it permits (maximum) N processes to access the critical section simultaneously while Mutexes are binary, they are either locked (0) or not-locked (1), so we can say Mutexes are Semaphore with the initial value of 1

There are several types of Semaphores:

- **System V semaphores**
- **POSIX semaphores** (easier to use and have better performance, so we covered in this writing)
  - **Un-named**: They exist in memory only so processes need to access to the same memory area. This means they can be used only by threads (of the same process). Since C++ has built-in support for Mutexes, un-named semaphores have limited application
  - **Named**: They can be used in multi-process codes and we covered them in this section

## 10.6. Sockets

Sockets are communication end-points. They are commonly used in network-based communication (like IP based, TCP/IP based and UDP/IP based). We can consider them as file descriptors (or handles), so we can open them, send and receive data through them and at the end close them.

Developing codes by using sockets is also called socket programming and it is very common among programmers. Although sockets are well supported on almost all operating systems but UNIX-like systems (like Linux and BSD) have stronger support of them. Sockets can be used in a variety of scenarios like network communication as well as IPC.

When it comes to IPC, communication through sockets is pretty much similar to communication through **pipes** and **FIFOs** but **sockets are bi-directional** (full duplex) and it is their advantage.

In the next page we have a simple example of how to communicate between 2 processes by using sockets.

How the code of previous page works:

- We created a pair of sockets by using **socketpair** system function
- We used UNIX-domain sockets (**AF\_UNIX**) so we don't need to define any IP address or Port number
- Two processes read from and write to sockets by standard **read** and **write** system functions, the same way as pipes and FIFOs
- At the end sockets are closed by calling standard **close** function

Example for Inter-process communication using UNIX-domain sockets:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp
//

#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int main()
{
    int sockets[2];
    char msgBuff[1024];

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0)
        return - 1;

    if (0 == fork())
    {
        const char msg1[] { "Hi From Process 1" };

        close(sockets[0]);
        if (read(sockets[1], msgBuff, sizeof(msgBuff)) < 0)
            std::cout << "Process 1: Socket Reading Error\n";

        std::cout << "Rceived in Process 1: " << msgBuff << std::endl;

        if (write(sockets[1], msg1, sizeof(msg1)) < 0)
            std::cout << "Process 1: Socket Writing Error\n";
        close(sockets[1]);
    }
    else
    {
        const char msg2[] { "Hi From Process 2" };

        close(sockets[1]);
        if (write(sockets[0], msg2, sizeof(msg2)) < 0)
            std::cout << "Process 2: Socket Writing Error\n";

        if (read(sockets[0], msgBuff, sizeof(msgBuff)) < 0)
            std::cout << "Process 2: Socket Reading Error\n";

        std::cout << "Rceived in Process 2: " << msgBuff << std::endl;

        close(sockets[0]);
    }
}

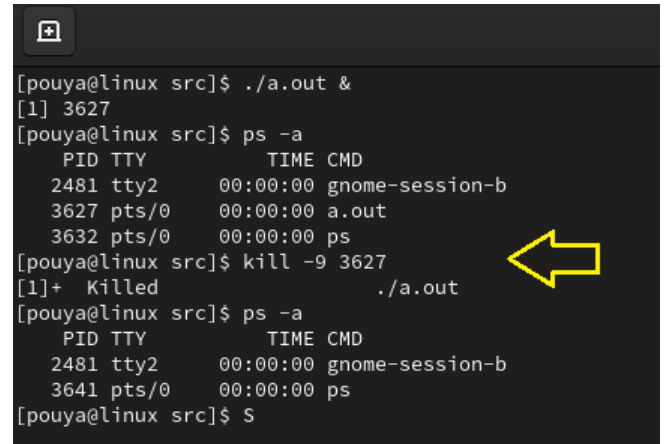
// Printed Result on Screen:
//
// "Rceived in Process 1: Hi From Process 2"
// "Rceived in Process 2: Hi From Process 1"
```

## 10.7. Signals

Signals in software are equivalent to interrupts in hardware. Most of the UNIX / LINUX operators are already familiar with this concept.

For example, when a user wants to terminate a process, he/she sends “Kill Signal” to the process.

In the opposite screenshot, user used shell command **kill** to send signal number 9 to process 3627. Signal number 9 is “Kill Signal” and once it is sent to a process, it terminated the process.



```
[pouya@linux src]$ ./a.out &
[1] 3627
[pouya@linux src]$ ps -a
  PID TTY          TIME CMD
 2481 tty2      00:00:00 gnome-session-b
 3627 pts/0      00:00:00 a.out
 3632 pts/0      00:00:00 ps
[pouya@linux src]$ kill -9 3627
[1]+  Killed                  ./a.out
[pouya@linux src]$ ps -a
  PID TTY          TIME CMD
 2481 tty2      00:00:00 gnome-session-b
 3641 pts/0      00:00:00 ps
[pouya@linux src]$ S
```

A yellow arrow points to the `kill -9 3627` command in the terminal output.

There are around 50 system signals defined in UNIX-like systems. Each signal has its own unique number and a related name. in [C/C++](#), programmers can use values (numbers) of signals or their names (names are defined as **#define** macros).

Any process is able to handle a signal by defining a handler function for that signal. Some special signals like kill signal (**SIGKILL**) cannot be handled by programs; they are always and only handled by OS, but most of the other signals can be handled by processes. When a process defines a handler function for a special signal, when that signal rises, OS calls this handler function asynchronously (it means even if the main process is busy with doing some jobs, this function will be called). Source of the signal is not important; the signal can be sent from shell (using kill command) or it can be sent from inside of another process (or even the process itself).

Any unhandled signal by process will be handled by OS.

Following is an example of communication between 2 processes using signals:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp
//

#include <iostream>
#include <unistd.h>
#include <signal.h>

void f(int sigNo) // Signal Handler
{
    std::cout << "Signal Received in the Main (Parent) Process, Number: " << sigNo;
    std::cout << std::endl;
}

int main()
{
    if (0 != fork())
    {
        signal(SIGUSR1, f); // Defining Signal Handler for Signal SIGUSR1
    }
    else
    {
        sleep(1);
        kill(getppid(), SIGUSR1); // Sending SIGUSR1 signal to parent process
    }

    sleep(2);
}

// Printed Result on the Screen:
//      "Signal Received in the Main (Parent) Process, Number: 10"
```

Here is how the above code works:

- The main (parent) process defines function `f()`<sup>2</sup> as the handler for signal `SIGUSR1`
- Child process gets process ID of its parent process by calling `getppid()` function
- Child process sends `SIGUSR1` signal to its parent process
- OS calls function `f()` in the parent process, since this function is the signal handler
- Both processes finish

---

<sup>2</sup> Signal handler functions should be re-entrant. They should be developed to safely handle re-entrance



## 11. What is Intel oneAPI?

About 5 years ago, Intel released an open and super-rich “Computational Programming” base toolkit named Intel oneAPI (latest release: 31 October 2024). The most important features of oneAPI can be listed as:

- It is developed around the concept of “**Heterogeneous Programming**” which means developing a code which can run on different processors (CPUs, GPUs of different vendors, ...) with no or minimal changes. “Heterogeneous Programming” is the topic of next article (post)
- It is developed to fully support “**Parallel Programming**”
- It includes a great **C++** compiler named DPC++ which fully supports SYCL
  - SYCL is one of the most popular programming models for “Heterogeneous Programming”
  - It also integrates with Visual Studio
- It is multi-platform, it can be installed on Windows, Linux and supports different hardware
- oneAPI has another toolkit for **HPC** (topic of the next chapter)
- It has tools & libraries to support developing **AI** (Artificial Intelligence) applications
  - Including Neural Networks (check my post on 2024-April-22)
- Most of its tools and libraries are performance tuned
  - Like IPP library for cryptography (check my benchmarking post on 2024-October-23)

Although both for Functional Parallelism and Data Parallelism (& Vectorization), C++ Language and Operating Systems provide a great support but almost all of this support is for running codes on CPUs only.

**Intel oneAPI** provides similar support but for Parallelism but on CPUs, GPUs and ...

The most commonly used libraries of **Intel oneAPI** are:

- **oneDPL**: Library for “Data Parallelism” including algorithms like search and sort
- **oneTBB**: Library for “Functional Parallelism” like handling threads and managing parallel sections
- **oneMKL**: Library for mathematical calculations like matrix algebra and FFT
- **oneDAL**: Library for Machine Learning and Data Analytics
- **oneDNN**: Library for working with Neural Networks (Deep Learning)
- **oneCCL**: Library for communication patterns for distributed deep learning
- **oneVPL**: Library for Video Processing like real-time video encoding, decoding & transcoding
- **IPP**: Performance tuned library for cryptography

Next in this chapter we will review 2 introductory level example codes in **C++** developed base on **oneDPL** and **oneTBB** (since they are about parallel programming and related to the topic of this writing).

To run these codes (or any oneAPI code), first you need to download and install intel oneAPI from:

<https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-download.html>

After installing oneAPI (it integrates to Visual Studio if you are installing it on Windows and you already have Visual Studio installed), you will have **2 options to build oneAPI codes**:

- Use Intel oneAPI DPC++ compiler: run "**Intel oneAPI command line for Visual Studio 2022**" command line and run the compiler like: `C:\> icpx /std:c++17 myFile.cpp`
- Open Visual Studio and create a new project of "DPC++ Console" type and then in the solution explorer go to "Project Properties | Configuration | Intel libraries for oneAPI" and change value of "Use oneDAL" to "Default Linking Method" (for oneDPL projects) and change value of "Use oneTBB" to "yes" (for oneTBB projects).

Following is an example code for Data Parallelism using **oneDPL**. If you compare the following code with the codes on pages 3 and 5, you will see a great similarity. Codes on pages 3 and 5 are developed based on standard **C++** language libraries (called STL) but the following code is developed using oneDPL. Pay attention to **oneapi::dpl** namespace and the execution policies defined in this namespace.

Beauty of oneDPL library is that algorithms (like following **sort()**), execution policies and other facilities are designed pretty much similar to **C++** language & libraries (STL). It makes oneDPL learning easier and faster, also codes migration from standard **C++** to oneDPL becomes easier and quicker.

```
// Pre-requirement : Installing Intel oneAPI
//
// On Windows opt1 : icpx file_name.cpp
// On Windows opt2 : Visual Studio
//      Create "DPC++ Console" type project then in the solution
//      explorer, right click on the project and in the
//      "Project Properties|Configuration|Intel libraries for oneAPI"
//      change value of "Use oneDAL" to "Default Linking Method"
//
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> nums{ 9, 4, 0, 3, 1, 6, -4, 7 };

    oneapi::dpl::sort(oneapi::dpl::execution::par, nums.begin(), nums.end());

    for (auto n : nums)
        std::cout << " " << n;

    std::cout << std::endl;
}

// Printed Result: -4 0 1 3 4 6 7 9
```

**oneTBB** is mainly developed to support Functional Parallelism so in comparison to oneDPL it is more complicated. Let's start with an example. In the following example we want to show 2 different features of oneTBB: "**Parallel For**" (similar to Parallel For in openMP) and Running 2 functions in parallel (threads). If you are already familiar with "**Lambda Functions**" in **C++**, the following example will look like easy.

```
// Pre-requirement : Installing Intel oneAPI
//
// On Windows opt1 : icpx file_name.cpp
// On Windows opt2 : Visual Studio
//                      Create "DPC++ Console" type project then in the solution
//                      explorer, right click on the project and in the
//                      "Project Properties|Configuration|Intel libraries for oneAPI"
//                      change value of "Use oneTBB" to "yes"
//
#include <vector>
#include <iostream>

#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#include "tbb/parallel_invoke.h"

void f(int a)
{
    std::cout << "From Function f(), a = " << a << std::endl;
}

int main()
{
    std::vector<int> v(10);

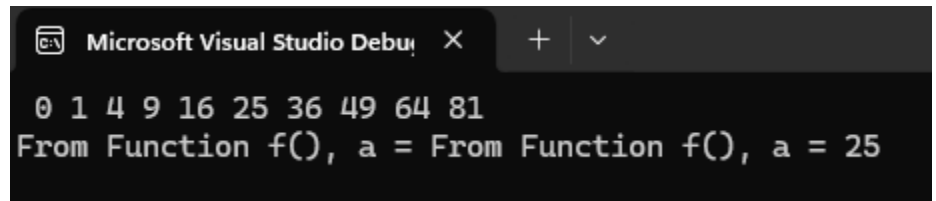
    tbb::parallel_for(tbb::blocked_range<int>(0, v.size()),
        [&](tbb::blocked_range<int> r)
        {
            for (int index = r.begin(); index < r.end(); ++index)
                v[index] = index * index;
        });

    for (auto n : v)
        std::cout << " " << n;
    std::cout << std::endl;

    tbb::parallel_invoke([] {f(2); }, [] {f(5); });
}

// Printed Result on Screen: 0 1 4 9 16 25 36 49 64 81
//                          From Function f(), a = 5
//                          From Function f(), a = 2
```

Following screen-shot is an actual output of a run of the previous page's example code:



```
0 1 4 9 16 25 36 49 64 81
From Function f(), a = From Function f(), a = 25
```

As it can be seen, one of the threads printed **"From Function f(), a= "** on the screen but then the value of variable **"a"** is not printed, because the 1<sup>st</sup> thread which printed this sentence didn't have time to print value of argument **"a"**; before it succeeds to print the value of **"a"**, 2<sup>nd</sup> thread printed its sentence of **"From Function f(), a= 2"** and then the first thread got the chance to print the value of variable **"a"** which is **"5"** and the final result is that messy output of **"From Function f(), a= From Function f(), a= 25"**. Here oneTBB did nothing except creating and running 2 threads (both with code of function **f()**).

The other functionality tested in the example code of the previous page is **"Parallel For"** section where we defined a parallel for to assign values to elements of a vector. Since it is a **"Parallel For"**, oneTBB may run it in parallel which means assigning values to different elements of vector will be done in parallel so this job will finish much sooner than a normal for where all iterations are done sequentially.

As it can be seen in the result, all elements of vector correctly received their values which are square of their indices. Functions and classes of oneTBB are located in **tbb** namespace.

We will talk more about **"Parallel For"** sections (of **C++** algorithms & openMP) on pages 51 & 52.

## 12. Parallel Programming for HPC

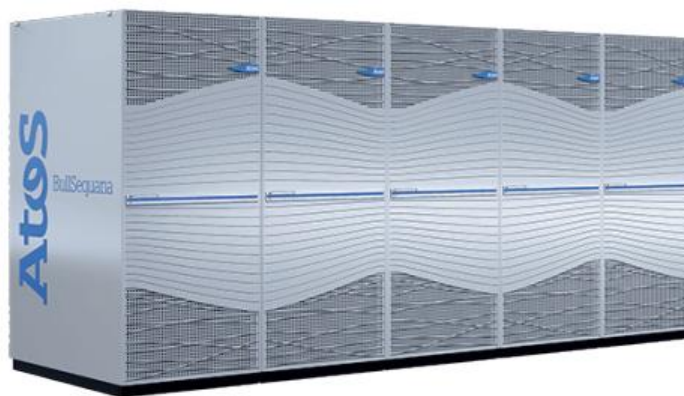
### 12.1. What is HPC?

**HPC** (High Performance Computing) is practice of aggregating computational resources (CPUs, GPUs, Memory, Storage) in a way to run codes with much higher performance than single machines (like **servers**).

When we are talking about HPC, we are talking about a set of hardware, **software** (including Parallel Programming) & networking technologies which makes this “High Performance Computing” possible.

**Almost always hardware infrastructure of HPC is super-computers.**

Just as an example of scale of supercomputers, “Aurora” a supercomputer installed in Argonne National Laboratory in US has more than **9,000,000** computing cores (<https://top500.org/lists/top500/list/2024/06>).



*Figure 3 Bull-Sequana XH2000 from Atos is used to make several super-computers in the world like Leonardo, in Italy*

One of The Key Features of **HPC** is Super-Fast Interconnection Between its Nodes.

If Computational Nodes (CPUs + GPUs + Memories) are Connected Through Ordinary Technologies, it is Called **HTC** (High Throughput Computing)

## 12.2. Symmetric (SMP) vs Asymmetric Multi-Processing (AMP)

Parallel Programming (Computing) is all about breaking and running codes on several “computing cores” in parallel. These computing cores can be:

- Cores of a CPU or a GPU
- Cores of several CPUs / GPUs on the same machine (like a server)
- Cores of CPUs / GPUs on different computational nodes (like network-connected computers)

For each hardware architecture, only a number of parallel programming techniques could be suitable, for example if all “computing cores” are in one CPU / GPU, most probably we can use multi-threading but if these “computing cores” are located inside several network-connected computers, then for sure multi-threading cannot be used because memories of these “computing cores” are not shared.

In chapter 5, we discussed that “Memory Model” is very important in parallel programming and it is one of the fundamental differences between “Processes” and “Threads”.

There are 2 different Memory Architectures:

- **Distributes Memory Architecture**: where computing cores have their own local memory but these memories are not connected to each other, like memories of network-connected computers
- **Shared Memory Architecture**: where all computing cores have access to the same memory space
  - **UMA** (Unified Memory Access): There is one flat memory space and all cores have full access to it as their local memory like memory in a **laptop** (see figure 3 on page 7).
  - **NUMA** (Non-Unified Memory Access): Cores have their own local memory but their memories are connected with fast bus (like in **servers**) so each core has access to other memories and the entire memory appears as a single piece (see figure 4 on the next page).

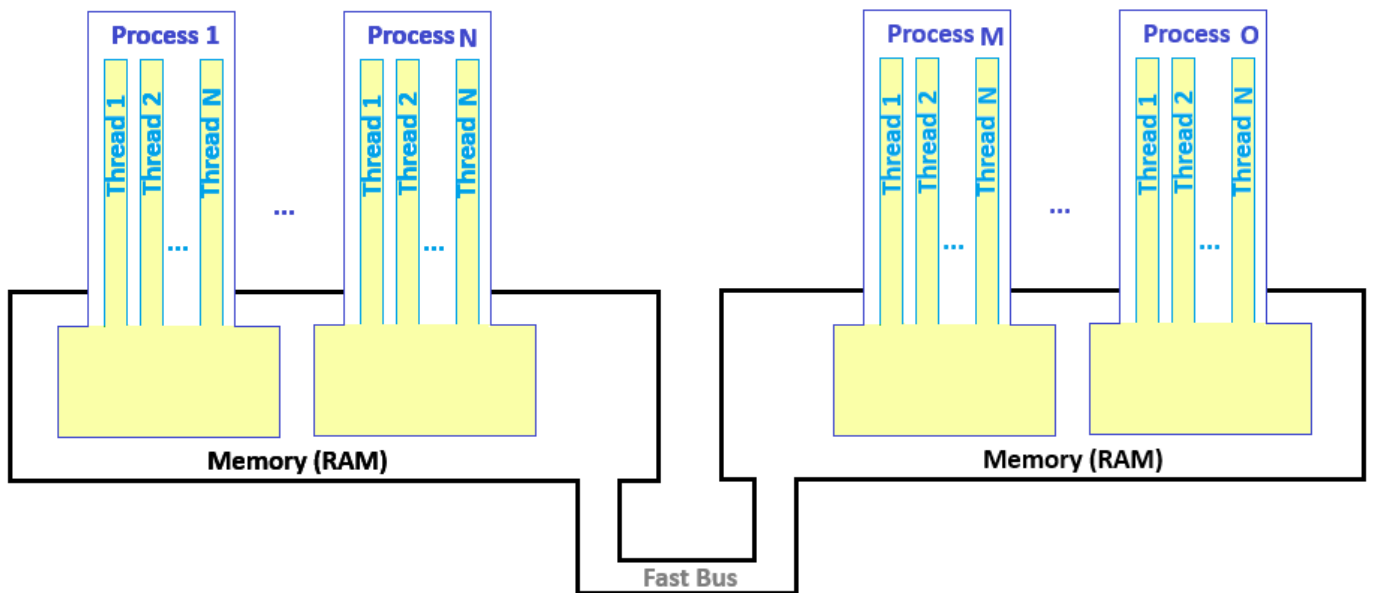


Figure 4 NUMA Architecture: Processor Cores Have Their Own Local Memory but Memories are Connected with Fast Bus so the whole Memory Appears as a Single Piece and in High Level it is Considered as a Single Shared Memory, in Some Server Machines with Multiple CPUs on Multiple Sockets, this Architecture is Used.

**Note:** There is also **Hybrid Architecture** where some small computational nodes with shared memory architecture are connected together to build a bigger distributed system.

- Multi-processing in Shared Memory Environment is called Symmetric Multi-Processing (**SMP**)
- Multi-processing in Distributed Memory Environment is called Asymmetric Multi-Processing (**AMP**)
  - In AMP environment all communications (between processes) are done through network
  - For technical reasons, architecture of big systems is distributed so we need to use AMP
  - Almost always, AMP is way slower than SMP

Shared Memory Dominant Parallel-Programming Standard is **openMP**

openMP supports Just 3 Languages: **Fortran** and **C/C++**

Distributed Memory Dominant Parallel-Programming Standard is **MPI**



As mentioned earlier, super-computers are hardware infrastructure for HPC and they are just super-huge computers. When working with HPC machines we need to be familiar with their building blocks:

- **Node** (or Computational Node): It is pretty much similar to a computer motherboard (main board) with some CPUs or GPUs and memory installed on them. Each node may have tens to hundreds of cores and Gigabytes of RAM. There are different types of nodes like Nvidia-GPU-nodes, Intel-CPU-nodes, AMD-GPU-nodes and AMD-CPU-nodes
- **Partition**: Several nodes are connected to each other with very high-speed bus to build a partition, usually HPC codes (programs) run on 1 partition (or part of a partition)
- **Cluster**: Several partitions together form a cluster

As an example, there is a great HPC machine in Island named **Elja**, its composition is like following:

1- Cluster **Elja**

- a. Partition **HPC-Elja** (95 nodes)
- b. Partition **Jötunn** (5 nodes)
- c. Another partition

2- Cluster **Mimir**

- a. Partition **Mimir** (9 nodes)
- b. Partition **Himem-mimir** (1 node)

3- Cluster **Stefnir**

- a. Partition **Stefnir** (32 nodes)

Source: <https://irhpcwiki.hi.is/docs/Partitions/HPC-Elja/Elja/>

## 12.3. openMP

**OpenMP** (Open Multi Processing) is the dominant standard in parallel programming in **shared** memory environments. Although OpenMP is mainly used to develop HPC codes but it is also supported by most of the **C/C++** compilers like MSVC (Visual Studio) and **gcc/g++**, so programmers can develop codes to run on **Windows / Linux** local machines (**laptops / desktops**). These codes then can be ported and executed on **HPC** machines (if their architecture is shared memory).

### To enable openMP in the Visual Studio 2022:

- For example, if there is a C++ “Windows Console App” project, in the solution explorer, right click on the project & then in the “Project Properties | Configuration Properties | C/C++ | Language” change the value of “OpenMP Support” to “yes”
- Visual Studio (as of today) supports OpenMP version 2 (it is claimed that v4.5 is supported by */openmp:experimental* switch but using this switch adds some limitations to the project)

### To compile openMP codes by gcc/g++

- Simply add **-fopenmp** switch
- As of today, latest version of RHEL (v 9.5) includes gcc 11.5 which supports openMP 4.5
- As of today, latest version of gcc (v 14.2) supports openMP 5.2 partially

Latest version of openMP standard is version 6.0 and its technical documents can be found at:

<https://www.openmp.org/>

Now, let's have an example for openMP:

```
// Pre-requirement : None
//
// On Linux      : g++ ./file_name.cpp -fopenmp
// On Windows    : Visual Studio (MSVC),
//                In the solution explorer, right click on the project & in the
//                "Project Properties | Configuration | C/C++ | Language"
//                change value of "openMP Support" to "yes"
//
#include <iostream>
#include <chrono>    // only for sleep function
#include <thread>    // only for sleep function
#include <vector>
#include <omp.h>

int main()
{
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            std::cout << "1 Start" << std::endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
            std::cout << "1 End" << std::endl;
        }

        #pragma omp section
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(30));
            std::cout << "2 All" << std::endl;
        }
    }

    std::vector<int> v(10);

    #pragma omp parallel for
        for (int index = 0; index < 10; index++)
            v[index] = index * index;

    for (auto n : v)
        std::cout << " " << n;

    std::cout << std::endl;
}

// Printed Result on Screen: 1 Start
//                          2 All
//                          1 End
//                          0 1 4 9 16 25 36 49 64 81
```

As it can be seen in the example code of the previous page, openMP programming is mainly based on `#pragma` directive. In this code, we demonstrated 2 features of openMP in an introductory way:

- **Parallel Sections:** in openMP, we can define a block of “parallel sections” and inside it we can define as many as “section” blocks. openMP runs these sections in parallel. In other words, each of these sections will be run as a thread. Actually, in the background openMP created threads and then assigns code of each “section” to a thread to run; so it is just multi-threading in openMP way.
- **Parallel For:** It is pretty much similar to “Parallel For” of Intel oneAPI-**oneTBB** as demonstrated & discussed on pages 44 and 45. Any ordinary “for loops” in C++ can be defined as “Parallel For” in openMP, in this case openMP tries to run iterations of the loop in parallel (if iterations don’t have correlation with each other), if openMP cannot run iterations in parallel (for any reason), it simply runs loop-iterations sequentially. In the example of previous page, all elements of vector `v` are initialized with the value of square of their indices but this initialization is done in parallel which means elements 0, 1, 2, ...9 are initialized almost at the same time. While if the “for loop” was an ordinary “loop” these elements would be initialized one after another (sequentially).

There is another standard similar to **openMP** which is called **openACC**. openACC is developed to run codes in parallel on **GPUs**. To the best of my knowledge, openACC is not supported by Intel and Microsoft C++ Compilers (Visual Studio), it is often used on HPC machines (running Linux).

**Note:** beside oneTBB and openMP, since **C++17** Parallel-For loops are built-in supported in C++ as well.

Example: `for_each(v.begin(), v.end(), [](int& a) {static int i{ 0 }; a = i * i; i++; });`

## 12.4. MPI

**MPI** (Message Passing Interface) is the dominant standard in parallel programming in **distributed** memory environments. Although MPI is designed to develop HPC codes but it is possible to install some implementations of it on any local machine and develop codes on laptops / desktops.

MPI can be used both in “Shared Memory” and “Distributed Memory” environments although in “Shared Memory” environments, using openMP is preferred due to its **higher performance**.

MPI fully supports Fortran and C/C++ languages although some other programming languages are also supported (by some extend) through using libraries and so.

MPI standard is defined and developed by MPI forum. Its technical documents can be found at <https://www.mpi-forum.org/>

Although all of the communications between processes in MPI is handled through network but MPI is not a network communication library (like sockets as it is discussed in section 10.6). MPI doesn't apply any security for communication because it is supposed that all the communication are carried out inside HPC nodes and partitions, but it provides much higher flexibility in comparison to sockets or similar tools.

Usually, operating systems do not have built-in support for MPI, so one of MPI implementations should be downloaded and installed prior using it.

One of the open source and popular implementations of MPI is **openMPI** which should not be mixed with openMP since they have nothing to do with each other.

## To install openMPI on Linux RHEL 9:

- 1- To check if openMPI is available & installed (check both "Installed" & "Available" groups):

```
# sudo dnf info openmpi
```

- 2- To install available openMPI package:

```
# sudo dnf install openmpi
```

- 3- After installation check if both **openmpi** and **openmpi-devel** packages are installed

- 4- Usually, libraries of openMPI are located at `/usr/lib64/openmpi/bin`

- 5- Above path should be added to path var: `export PATH=$PATH:/usr/lib64/openmpi/bin`

- 6- To permanently add above path, you may update bash profile at `/etc/profile`

- 7- Compiling MPI codes (with openMPI compiler)

```
# mpicc file_name.c
```

```
# mpic++ file_name.cpp
```

- 8- **Running MPI based codes:**

```
# mpirun --use-hwthread-cpus -n 4 ./a.out
```

In above line, number after `-n` is number of processes that openMPI will create (in this case 4 processes will be created). Number of processes never should exceed number of actual processing (real) cores on machine. openMPI doesn't consider HT (Hyper-Threading), but if you want to force openMPI to consider HT and count all cores (with HT), you may use `--use-hwthread-cpus` switch.

Now, let's have an example for MPI:

```
// Pre - requirement : openMPI (as described in the previous page)
//
// On Linux           : mpic++ ./file_name.cpp           (to compile)
//                     mpirun --use-hwthread-cpus -n 4 ./a.out (to run)
//

#include<iostream>
#include<mpi.h>

int main(int argc, char* argv[])
{
    int size, rank;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get unique ID of each process
    MPI_Comm_size(MPI_COMM_WORLD, &size); // get total number of processes

    switch (rank)
    {
    case 0:
        std::cout << "from FIRST Process of " << size << " Processes\n";
        break;

    case 1:
        std::cout << "from SECOND Process of " << size << " Processes\n";
        break;

    case 2:
        std::cout << "from THIRD Process of " << size << " Processes\n";
        break;

    case 3:
        std::cout << "from FOURTH Process of " << size << " Processes\n";
        break;

    default:
        std::cout << "from ANOTHER Process of " << size << " Processes\n";
        break;
    }

    MPI_Finalize();
}

// Printed Result:Following sentences will be printed but with random order
//                 from FIRST Process of 4 Processes
//                 from THIRD Process of 4 Processes
//                 from SECOND Process of 4 Processes
//                 from FOURTH Process of 4 Processes
```

## Difference Between Developing Parallel Codes for Single Machines Vs in HPC Environment

**on Single Machines:** Programmer Determines Number of Processes / Threads

**in HPCs:** Programmer Doesn't Know Number of Processes, The Operator Who Runs the Code Determines It (Based on The Hardware Specifications), so HPC Codes Should be Designed to be **Scalable** to Run with **Any Number of Processes**

**Note:** To run codes developed based on openMPI as described in the previous pages we can use `mpirun` command but in real HPC environment, MPI codes are run through “Job Scheduling” (workload manager) systems like SLURM. **Number of processes are defined in SLURM configuration files.**

The example of the previous page is the simplest example to demonstrate how MPI works. The next step is to make communication between processes using functions like `MPI_Send()`, `MPI_Recv()` and `MPI_Bcast()` which are out of scope of this writing. The other advanced topic which comes after inter process communication is “Cartesian Communicators” which interested readers may find more details in MPI web site.

The Art of Parallel Programming Hides in the Skill of Perfect (& Scalable)

**Decomposition** of a Big Problem to Many Smaller Problems (Amdahl's Law)



## 13. Amdahl's Law

All along this writing we talked about different techniques, technologies and methods for parallel computing also we mentioned that, to decrease execution time of a computer program and to increase its performance, we can try to break down the code to more parallel sections (tasks) and run them in parallel.

It worth to mention that codes (programs) comprise of two types of sections:

- **Sequential Section:** it is a section that can **NOT** be broken into smaller sub-sections hence it should be run sequentially, for example the section of the code which asks for some inputs from user
- **Parallel Section:** it is a section that can be broken into smaller sub-sections (sub-tasks) in a way that these sub-sections can run in parallel, for example Matrix multiplication calculations

Now the main question is this:

If we break a program into more parallel sub-section, the total execution time of parallel sections will be reduced (we call it parallel section speed up). Since just a proportion of each program is parallel sections (and of course there are some sequential sections as well), if we increase performance of parallel sections how much it will affect the performance of the whole program (called overall speedup factor):

**Amdahl's Law answers this question:**

$$\text{Overall Speedup Factor (of a program)} = \frac{1}{(1 - p) + \frac{p}{s}}$$

Where:

$p$  is the proportion of the program which can run in parallel (range 0 to 1)

$s$  is the speedup factor of the parallel sections

For example, if in a program only 10% (= 0.1) of the code runs in parallel, by improving execution performance of this part to run 2 times faster, the whole program will run 5.3% faster (= 1.053 times).

Amdahl's law presented by American computer scientist, Gene Amdahl, in 1967 but still is in center of focus in the domain of parallel computing.

Following diagram shows theoretical maximum speed-up of a computer program when running on a machine with different number of computing cores. In this diagram speed-up factor of several programs are depicted in different colors. Each color represents a program with different “parallel section” proportion (0.5 to 0.95).

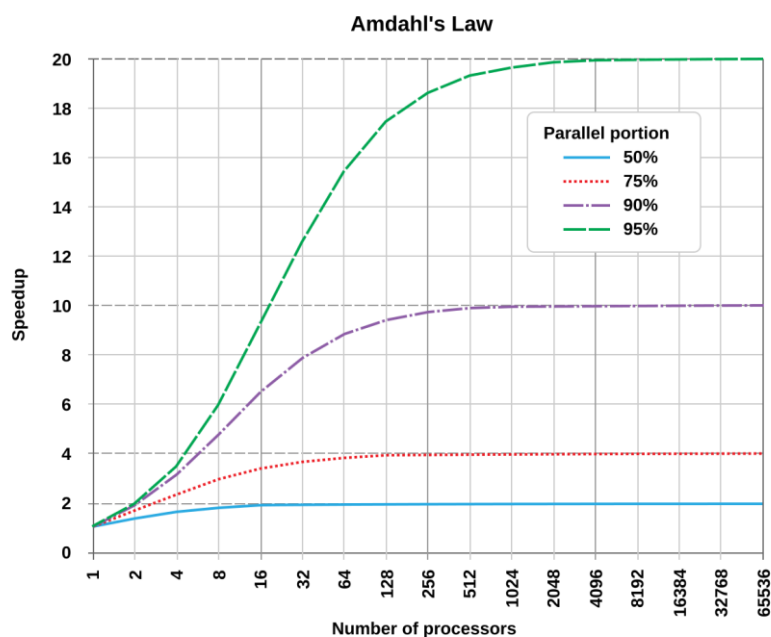


Figure 5 Source: [https://en.wikipedia.org/wiki/Amdahl%27s\\_law](https://en.wikipedia.org/wiki/Amdahl%27s_law)

There are other related laws like **Gustafson's law** (or Gustafson–Barsis's law) those also talk about speed up factor in parallel programming but from different perspectives or by considering different factors.

## 14. Some Advanced Topics to Study

In this book: “**a Tour of C++, Third Edition updated for C++20**” (more information about this book can be found on the last page of this writing), following topics are discussed regarding **parallel programming**:

- 1- `std::async`
- 2- `std::shared_mutex`
- 3- `std::shared_timed_mutex`
- 4- `std::scoped_lock`
- 5- `std::lock_guard`
- 6- `std::unique_lock`
- 7- `std::shared_lock`
- 8- `std::atomic`
- 9- `std::condition_variable`
- 10- `std::promise` and `std::future`
- 11- `std::packaged_task`

In this book: “**Advanced Programming in the UNIX Environment**” (more information about this book can be found on the last page of this writing), following topics are discussed regarding **IPC**:

- 1- `wait()`, `waitpid()` and `waitid()`
- 2- `nice()` and `getpriority()`

- 3- times()
- 4- popen() and pclose()
- 5- struct ipc\_perm
- 6- struct msqid\_ds
- 7- struct shmid\_ds
- 8- mmap()
- 9- sem\_trywait()
- 10- sem\_timedwait()
- 11- sem\_init() and sem\_destroy
- 12- sem\_getvalue()
- 13- socket()
- 14- shutdown()
- 15- bind()
- 16- getsockname()
- 17- connect()
- 18- listen()
- 19- accept()
- 20- send() , sendto() , sendmsg() , recv() , recvfrom() and recvmsg()
- 21- setsockopt() and getsockopt()
- 22- alarm() and pause()

## 15. Test platform (Hardware & Software Specification)

<b>Machine</b>	Laptop Asus TUF Gaming F15
<b>CPU</b>	Intel Core i7 12700 (6 P-Core + 8 E-Core, in total 20 cores) @ 4 GHz
<b>RAM</b>	32 GB @ 3.2 GHZ
<b>GPU</b>	Nvidia RTX 4070 (36 SMs, each has 128 cores, in total 4608 CUDA cores)
<b>GRAM</b>	8 GB @ 8 GHZ
<b>OS</b>	Windows 11 Enterprise 64-bit (24H2) Linux RHEL V12.5 64-bit, Kernel V5.14 ( <i>installation on Hyper-V VM</i> )
<b>Language(s)</b>	ISO C++11, C++14, C++17, C++20
<b>Compiler</b>	<i>For Windows:</i> Visual Studio 2022 V17.12 (MSVC) nvcc 12.5 for CUDA (GPU) Intel icpx (dpc++) 2024.2.1 <i>For Linux:</i> gcc/g++ V11.5
<b>Compile Mode</b>	64-bit

## 16. Table of Acronyms and Abbreviations

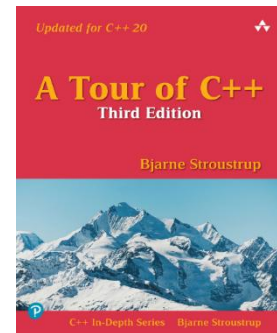
Acronym / Abbreviation	Stands For
AI	Artificial Intelligence
AMP	Asymmetric Multiprocessing
API	Application Programming Interface
ASMP	Asymmetric Multiprocessing
BSD	Berkeley Software Distribution
ccNUMA	Cashe Coherent Nonuniform Memory Access
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPC++	Data Parallel C++
E-Core	Efficient Core (CPU)
FIFO	First In, First Out
FFT	Fast Fourier Transform
G	Giga (almost 10 to the power of 9)
GCC	GNU C Compiler (GNU stands for GNU Not Unix)
GHZ	Giga Hertz
GPU	Graphics Processing Unit
GP-GPU	General Purpose Graphics Processing Unit
HPC	High Performance Computing
HT	Hyper Threading
HTC	High Throughput Computing
IPC	Inter-process Communication
ISO	Independent System Operator
Jan.	January
LWP	Light Weight Process
M	Mega (almost one million)
MHz	Mega Hertz (Mega means million)
MPI	Message Passing Interface
MSVC	Microsoft Visual C
mV	milli Volt

NPU	Neural Processing Unit
NUMA	Non-Uniform Memory Access
nvcc	Nvidia CUDA Compiler
oneAPI	one Application Programming Interface
oneDPL	one Data Parallel Library
openMP	Open Multi Processing
openMPI	open Message Passing Interface
oneTBB	one Threading Building Blocks
omp	openMP (Open Multi Processing)
OS	Operating System
P-Core	Performance Core (CPU)
POSIX	Portable Operating System Interface for Unix
RAM	Random Access Memory
RHEL	Red Hat Enterprise Linux
RPM	Revolution Per Minute
RTX	Ray tracing Texel eXtreme - Ray Tracing eXperience
s	Seconds
SIMD	Single Instruction Multiple Data
SLURM	Simple Linux Utility for Resource Management
SM	Streaming Multiprocessors
SMP	Symmetric Multi-Processing
Std	standard
STL	Standard Template Library (C++)
SYCL	System wide Compute Language
TM	Trade Mark
UMA	Unified Memory Access
V	Version
Var	Variable
VERILOG	VERification & LOGic
VHDL	VHSIC (Very high-speed integrated circuit) Hardware Description Language
VM	Virtual Machine

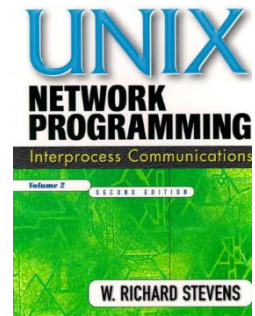
## 17. Books & References about Parallel Programming

- 1- Bjarne Stroustrup: **a Tour of C++, Third Edition updated for C++20.**

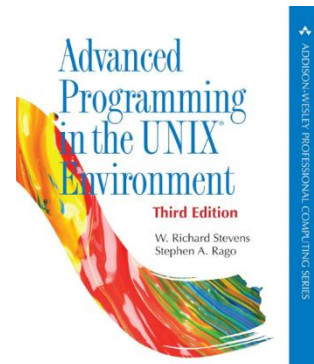
Pearson Addison Wesley, October 2022.



- 2- W. Richard Stevens: **UNIX Network Programming, Volume 2: Interprocess Communications** 2<sup>nd</sup> edition. Prentice Hall, 1999.



- 3- Richard Stevens & Stephen A. Rago: **Advanced Programming in the UNIX Environment.** Pearson Addison Wesley, May 2013.



- 4- **openMP** Specification and Programming Guide: <https://www.openmp.org/>
- 5- **MPI** Specification and Programming Guide: <https://www.mpi-forum.org/>
- 6- **CUDA C++** Programming Guide: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- 7- **HPC** Training Course of Prof Dr - Ing Morris Riedel in 2024:  
[https://www.youtube.com/watch?v=\\_Z0JPlu3d8Y&list=PLmJwSK7qduwVAnNfpueCgQqfchcSIEMg9](https://www.youtube.com/watch?v=_Z0JPlu3d8Y&list=PLmJwSK7qduwVAnNfpueCgQqfchcSIEMg9)