

Practical Session 1

Juan Diego Collazos Mejia

16 August 2025

1 Questions

- Describe the four Flynn Architecture Classifications. Explain why the MISD configuration has traditionally been thought of as non-sensible.



Flynn (1966) classified computer architectures according to the number of instruction streams and data streams they can handle simultaneously:

1. SISD (Single Instruction, Single Data)

- Traditional von Neumann architecture.
- A single processor executes one instruction stream on one data stream.
- Example: classic uniprocessor.

2. SIMD (Single Instruction, Multiple Data)

- One control unit broadcasts the same instruction to multiple processing elements, each working on different data.
- Very efficient for vector/matrix operations, image/video processing.
- Example: GPUs, vector processors.

3. MISD (Multiple Instruction, Single Data)

- Multiple instruction streams operate on the **same** data stream.
- Traditionally considered non-sensible because having several different instructions acting on the same piece of data rarely offers useful computation.
- However, some fault-tolerant systems use MISD-like redundancy (e.g., multiple processors computing the same result in different ways to detect errors).

4. MIMD (Multiple Instruction, Multiple Data)

- Multiple processors execute independent instruction streams on independent data streams.
- Most modern multiprocessors and multicore systems fall into this category.
- Example: multicore CPUs, distributed computing clusters.

► Why MISD is Non-Sensible

- The idea of having many instructions act on the same data point is not useful in practice for general-purpose computing.
 - It wastes hardware resources because you could instead process different data (MIMD) or the same operation on different data (SIMD).
 - Only niche applications like **fault tolerance** or **real-time control systems** justify MISD.
- One of the design goals of Instruction-Level Parallelism (ILP) was to make it transparent to the software layer; however, there are limitations as to how effective it is, and to what degree of aggressiveness or “look-ahead” it can be performed at. What are the causes of the limitations and how do they arise?



ILP tries to exploit parallelism *within a single program stream* by overlapping instruction execution (e.g., pipelining, superscalar execution, out-of-order execution).

Limitations arise from:

1. Data dependencies

- True data dependence: instruction B needs the result of A.

2. Control dependencies

- Branches make it hard to know which instructions should execute next.
- Branch prediction is imperfect.

3. Memory dependencies

- Unknown whether two memory references alias (point to the same address).

4. Hardware complexity

- Aggressive ILP (large reorder buffers, speculation, register renaming) increases power, cost, and design difficulty.
 - Diminishing returns: beyond a certain point, extra complexity doesn't yield much more parallelism.
- What are the micro-architecture and macro-architecture trends and HW design problems that have led to the need for programmers to explicitly express parallelism into their software?



Over time, several hardware design issues forced programmers to take more responsibility for expressing parallelism:

- End of frequency scaling (around mid-2000s): clock rates hit thermal and power walls.
- Limited ILP: hardware cannot “find” enough parallelism automatically in sequential programs.
- Rise of multicore CPUs and GPUs: parallel execution units are abundant, but require explicit parallel software.

- Memory wall: latency of memory access doesn't scale like CPU speed; requires careful programmer-managed parallel memory access (tiling, blocking).
- What are some of the challenges associated with directly exposing the underlying hardware parallelism to the software layer and programmers?



- Complexity for programmers: writing parallel code is harder (synchronization, race conditions, deadlocks).
 - Portability: code tuned for one parallel architecture may not map well to another.
 - Debugging difficulties: nondeterministic execution due to interleavings.
 - Load balancing: programmers must ensure even distribution of work across cores.
 - Memory consistency models: different architectures have different visibility/ordering guarantees.
- What is HyperThreading, and what, if anything, should software programmers do to leverage it as effectively as possible?



Hyper-Threading (HT) is a technology developed by Intel that allows a single physical processor core to execute two threads at the same time.

In simple terms:

- Each physical core presents itself to the operating system as two logical cores.
- This means that if you have a 4-core processor with Hyper-Threading, the operating system will see it as 8 logical processors.

What programmers should do:

- Write multithreaded code that can exploit more logical cores.
- Avoid excessive thread contention for shared resources (locks, memory bandwidth).
- Use standard parallel programming frameworks (e.g., OpenMP, TBB, pthreads) so the OS and runtime can map threads efficiently.

2 Scripting Languages

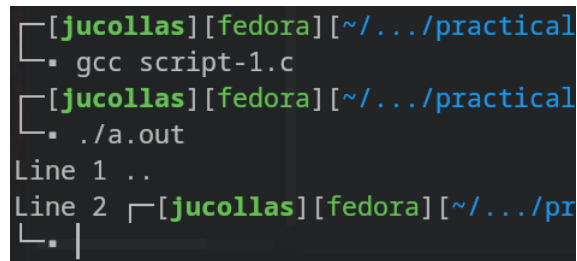
2.1 Difference between output

Execute the following 'C' programs, first interactively, then by redirecting output to a file at the UNIX shell level with a ">".

Program 1

```

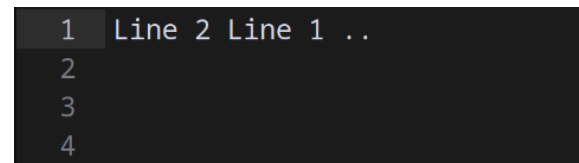
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main (void){
5     printf("Line 1 ..\n");
6     write(1,"Line 2 ",7);
7 }
```



```

[jucollas][fedora][~/.../practical]
└─ gcc script-1.c
[jucollas][fedora][~/.../practical]
└─ ./a.out
Line 1 ..
Line 2 [jucollas][fedora][~/.../practical]
└─ |
```

Figure 1: Interactively - script 1



```

1 Line 2 Line 1 ..
2
3
4
```

Figure 2: Redirecting output to a file - script 1

Explain the difference between the output observed on the terminal and that contained in the target piped file.

► The difference is caused by **buffering**.

- printf uses the C standard I/O library, which is line-buffered when writing to the terminal (it flushes on `\n`) but fully buffered when redirected to a file (data stays in memory until the program ends).
- write is a low-level system call: it writes immediately, without buffering.

So:

- On the terminal, printf flushes right away (because of `\n`), so you see "Line 1 .." first.
- With redirection, printf waits in the buffer while write goes directly to the file, so "Line 2 " appears before "Line 1 ..".

Program 2

```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main (void){
5     printf("Line 1 ..\n");
6     fflush(stdout);
7     write(1,"Line 2 ",7);
8 }

```

```

[jucollas][fedora][~/.../practical
└─ gcc script-2.c
[jucollas][fedora][~/.../practical
└─ ./a.out
Line 1 ..
Line 2 [jucollas][fedora][~/.../pr

```

Figure 3: Interactively - script 2

```

1 Line 1 ..
2 Line 2
3
4

```

Figure 4: Redirecting output to a file - script 2

Explain what has happened with the addition of the fflush system call.

► On the terminal there is no difference because stdout is line-buffered and the flushes the buffer, but with redirection fflush forces the buffer to be flushed, ensuring that "Line 1 ..." is written first and the sequential order in the file is preserved.

2.2 The different execution order on different runs

Program 3

Improve and run the following 'C' program several times interactively.

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 int main (){
6     int pid;
7     int i;
8     for (i=0; i<3; i++){
9         if ((pid = fork()) < 0 ){
10             printf("Sorry...cannot fork\n");
11         }else if (pid == 0){
12             printf("child %d\n", i);
13         }else{
14             printf ("parent %d\n", i);
15         }
16     }
17     exit(0);
18     return 0;
19 }

```

```

1  parent 0
2  parent 1
3  parent 2
4  child 0
5  parent 1
6  parent 2
7  parent 0
8  parent 1
9  child 2
10 parent 0
11 child 1
12 child 2
13 child 0
14 parent 1
15 child 2

```

Figure 5: Execution 1 - script 3

```

1  parent 0
2  parent 1
3  parent 2
4  parent 0
5  parent 1
6  child 2
7  parent 0
8  child 1
9  parent 2
10 child 0
11 parent 1
12 parent 2
13 parent 0
14 child 1
15 child 2

```

Figure 6: Execution 2 - script 3

Note the different execution order on different runs, Why?

► The output order changes because after each **fork()**, both parent and child processes continue running concurrently from the same point. The operating system's scheduler decides which process runs first, and this scheduling is non-deterministic. Therefore, the print statements can appear in different orders on each run.

Program 4

Making the minor changes to program 3 above needed to get the code below, execute the following 'C' program several times interactively.

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/wait.h>
5
6  int main (){
7      int pid;
8      int i;
9      for(i = 0; i < 3; i++){
10         if ((pid = fork()) < 0 ){
11             printf("Sorry...cannot fork\n");
12         }else if (pid == 0){
13             printf("child %d\n", i);
14         }else{
15             wait(NULL);

```

```

16         printf("parent %d\n", i);
17     }
18 }
19 exit(0);
20 return 0;
21 }

```

```

1 child 0
2 child 1
3 child 2
4 child 0
5 child 1
6 parent 2
7 child 0
8 parent 1
9 child 2
10 child 0
11 parent 1
12 parent 2

```

Figure 7: Execution 1 - script 4

```

1 child 0
2 child 1
3 child 2
4 child 0
5 child 1
6 parent 2
7 child 0
8 parent 1
9 child 2
10 child 0
11 parent 1
12 parent 2

```


Figure 8: Execution 2 - script 4

Explain how and why the order of the output from this program is different from that of the after program.

► The use of `wait(NULL)` makes the output deterministic because the parent is forced to pause until its child finishes execution. As a result, each iteration always prints the child's message first, followed by the parent's message, in a fixed order.

2.3 Encrypt and decrypt programs

Create `encrypt_it` and `decrypt_it` programs using C, C++ and bash scripting language. Both receive two arguments: the rotation index and the phrase to be encrypted or decrypted.

 `cipher-cesar.cpp` in GitHub

- Encrypt "Aprendiendo Programación paralela".

```

[jucollas][fedora][~/.../practical-session-1/scripts]
$ ./cipher e 17 "Aprendiendo Programacion paralela"
Rgiveuzveuf Gifxirdrtzfe grircvcr

```

Figure 9: Execution 1 cipher-cesar

- Decrypt "Zú iolxgju Ikygx, latioutg!" with rotation 6.

```
[jucollas][fedora][~/.../practical-session-1/scripts]
└─ ./cipher d 6 "Zu iolxgju Ikygx, latioutg!"
To cifrado Cesar, funciona!
```

Figure 10: Execution 2 cipher-cesar

- Encrypt it with two different sets of arguments and redirect its output to two separate files and show the content.

```
[jucollas][fedora][~/.../practical-session-1/scripts]
└─ ./cipher e 11 "Hello word" >| outfile-cesar1.txt
[jucollas][fedora][~/.../practical-session-1/scripts]
└─ cat outfile-cesar1.txt
Spwwz hzco
```

Figure 11: Execution 3 cipher-cesar

```
[jucollas][fedora][~/.../practical-session-1/scripts]
└─ ./cipher e 23 "bet on all the horses" >| outfile-cesar2.txt
[jucollas][fedora][~/.../practical-session-1/scripts]
└─ cat outfile-cesar2.txt
ybq lk xii qeb elopbp
```

Figure 12: Execution 4 cipher-cesar

- Decrypt the text processed after.

```
[jucollas][fedora][~/.../practical-session-1/scripts]
└─ ./cipher d 11 "$(cat outfile-cesar1.txt)"
Hello word
```

Figure 13: Execution 5 cipher-cesar

```
[jucollas][fedora][~/.../practical-session-1/scripts]
└─ ./cipher d 23 "$(cat outfile-cesar2.txt)"
bet on all the horses
```

Figure 14: Execution 6 cipher-cesar

Questions:


- **Have you reached parallel code? Or parallel behavior?**
 - These programs are sequential, not parallel. Each character is processed one after another.
- **Does the shell-script code perform parallelism or pipelining?**
 - No, it is also sequential. Shell scripts can launch parallel processes using `&` or pipelines `|`, but our encryption runs linearly.
- **But if script-based parallel programming is so easy, why bother with anything else?**
 - Because:
 - Scripts are limited in performance (slow, high overhead).
 - Not suitable for fine-grained parallelism (e.g., character-level).
 - Real parallel programming in C/C++ (with threads, OpenMP, MPI) gives better control, efficiency, and scalability.

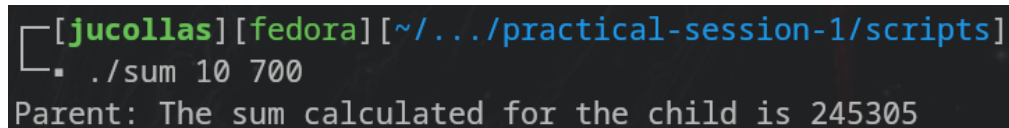
3 POSIX Environment

Another alternative to achieve basic parallelism is the POSIX environment. This includes primitives to manage processes and pthreads. This environment is readily available and widely supported by modern Operating Systems.

Real-world applications using `fork()`, `kill()`, `exit()`, among other primitives to manipulate signals, file descriptors, shared memory segments, and any number of other resources.

- Write a C program that uses `fork` to create a child process.
- The child process should calculate the sum of all numbers within a given range. Meanwhile, the parent process waits for the child to finish. Once the child completes the calculation, it sends the result to the parent using a pipe, and the parent prints the result. (sum.c)

 `sum.c` in GitHub




```
[jucollas][fedora][~/.../practical-session-1/scripts]  
└─ ./sum 10 700  
Parent: The sum calculated for the child is 245305
```

Figure 15: Execution 1 sum.c

- Improve the code and use the `fork` primitive to create $(n/3)$ child processes (adder.c) for a input of length (n)


- Modifies the program to receive the a list of numbers from standard input

 [adder.c in GitHub](#)

```
[jucollas][fedora][±][main ?:1 X][~/.../scripts/cfiles]
└─ ./adder 1 7 8 9 40 10 1456
1531
```

Figure 16: Execution 1 adder.c

- Create a program to generate n number randomly generated and print in standard output (randomizer.c)

 [randomizer.c in GitHub](#)

```
[jucollas][fedora][±][main ?:4 X][~/.../scripts/cfiles]
└─ ./randomizer 9
28 302 346 114 191 376 879 427 701
```

Figure 17: Execution 1 randomizer.c

- Pipeline randomizer and adder programs

```
[jucollas][fedora][±][main ?:4 X][~/.../scripts/cfiles]
└─ ./randomizer 9 | xargs ./adder
4793
```

Figure 18: Execution 1 randomizer and adder

Questions:


- Child processes run in parallel mode? Or concurrently? Why?
 - Child processes run concurrently, and if the system has multiple CPU cores, they can run in parallel. This is because **fork()** creates independent processes scheduled by the operating system, which may interleave them on one core or truly execute them simultaneously on multiple cores.
- Is the running time equal for sum and adder programs? Which is faster? Why?
 - **The running time is not equal.**
 - On a single-core system, the sum program is usually faster because it avoids the overhead of multiple processes and IPC.

- On a multi-core system, the adder program can be faster because the workload is divided among several child processes that may execute in parallel, reducing the effective computation time.
- Do parent and child processes use shared memory? Is it important?
 - By default, parent and child processes do not share memory. Each has its own private address space after **fork()**. Communication requires Inter-Process Communication (IPC) mechanisms such as pipes (used in your code). This is important because it prevents accidental interference between processes but introduces overhead compared to shared memory approaches.

3.1 Programming with threads

A programming model that allows multiple threads to exist in a single process, sharing the same memory space and a Thread is a single sequence of programming instructions. A process may create many threads. In Linux terminology, the main thread is called the leader. We can call all the other created threads followers. All the followers must have the same PID and PPID as the leader. They will however have mutually distinct TIDs. The followers should share almost everything (as in pthread) except the stack.

- Write a program to create two matrices, A and B, of sizes 40x40, which only take binary inputs 0 and 1.
- Create two threads: a thread will compute the summation of the two matrices and the other will compute their difference parallelly. There will be one main thread which will wait for the two parallel threads to complete their task and it will compute the multiplication of the two previously obtained outputs and print the final output.

 `matrices-threads.c` in GitHub