Vector Processor

Definition:

Vector processor is basically a central processing unit that has the ability to execute the complete vector input in a single instruction. More specifically we can say, it is a complete unit of hardware resources that executes a sequential set of similar data items in the memory using a single instruction.

We know elements of the vector are ordered properly so as to have successive addressing format of the memory. This is the reason why we have mentioned that it implements the data sequentially (il implémente les données de manière séquentielle).

It holds a single control unit but has multiple execution units that perform the same operation on different data elements of the vector.

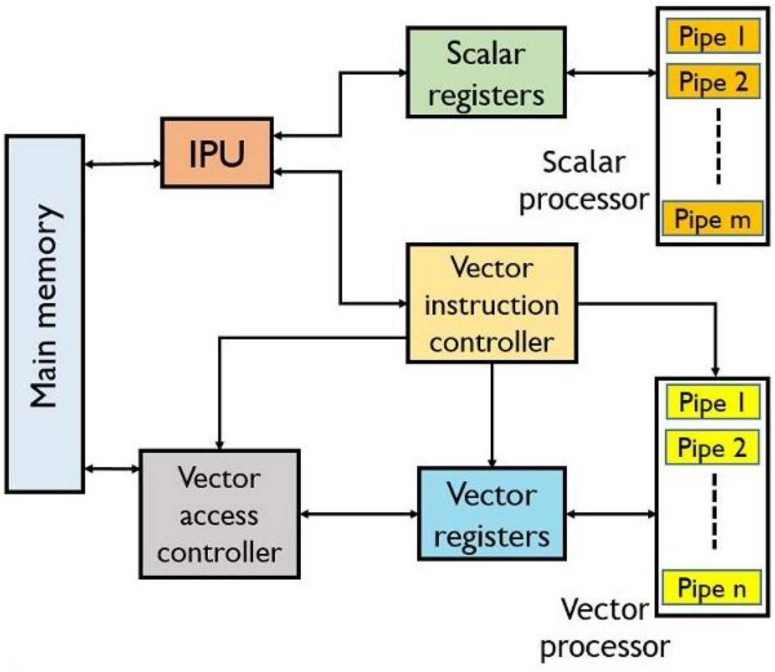
Vectorization converse process :

Unlike scalar processors that operate on only a single pair of data, a vector processor operates on multiple pair of data. However, one can convert a scalar code into vector code ,So, we can say vector processing allows operation on multiple data elements by the help of single instruction.

These instructions are said to be **single instruction multiple data** or **vector instructions**. The CPU used in recent time makes use of vector processing as it is advantageous than scalar processing.

Architecture :

The figure below represents the typical diagram showing vector processing by a vector computer:



The functional units of a vector computer are as follows:

- IPU or instruction processing unit
- Vector register
- Scalar register
- Scalar processor
- Vector instruction controller
- Vector access controller
- Vector processor

how it Work?

As it has several functional pipes thus it can execute the instructions over the operands. We know that both data and instructions are present in the memory at the desired memory location. So, the instruction processing unit i.e., IPU fetches the instruction from the memory.

Once the instruction is fetched then IPU determines either the fetched instruction is scalar or vector in nature. If it is scalar in nature, then the instruction is transferred to the scalar register and then further scalar processing is performed.

While, when the **instruction is a vector** in nature then it is fed to the vector instruction controller. This vector instruction controller first decodes the vector instruction then accordingly determines the address of the vector operand present in the memory.

Then it gives a signal to the **vector access controller** about the demand of the respective operand. This vector access controller then fetches the desired operand from the memory. Once the operand is fetched then it is provided to the instruction register so that it can be processed at the vector processor.

At times when multiple vector instructions are present, then the vector instruction controller provides the multiple vector instructions to the task system. And in case the task system shows that the vector task is very long then the processor divides the task into subvectors.

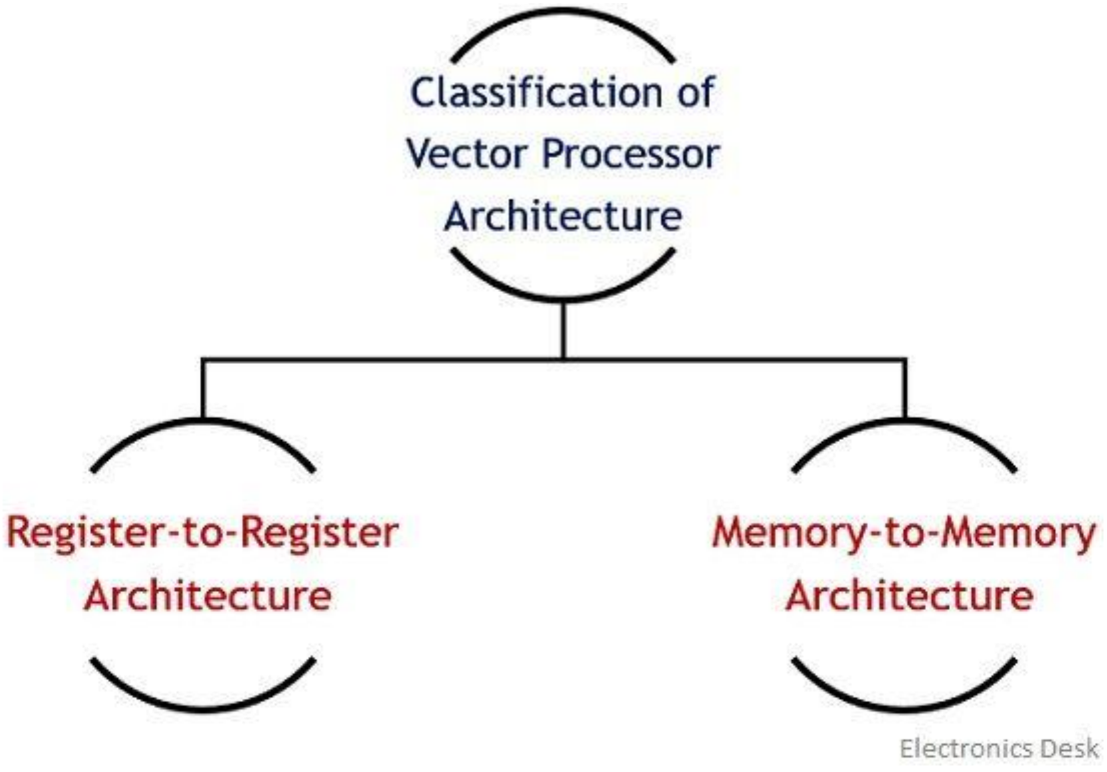
These subvectors are fed to the vector processor that makes use of several pipelines in order to execute the instruction over the operand fetched from the memory at the same time.

The various vector instructions are scheduled by the vector instruction controller.

Classification of Vector Processor

The classification of vector processor relies on the ability of vector formation as well as the presence of vector instruction for processing. So, depending on these criteria,

vector processing is classified as follows:


Classification of Vector Processor Architecture
Register-to-Register Architecture
Memory-to-Memory Architecture
Electronics Desk

Register to Register Architecture:

This architecture is highly used in vector computers. As in this architecture, the fetching of the operand or previous results indirectly takes place through the main memory by the use of registers.

The several vector pipelines present in the vector computer help in retrieving the data from the registers and also storing the results in the desired register. These vector registers are user instruction programmable.

This means that according to the register address present in the instruction, the data is fetched and stored in the desired register. These vector registers hold fixed length like the register length in a normal processing unit.

Memory to Memory Architecture:

Here in memory to memory architecture, the operands or the results are directly fetched from the memory despite using registers. However, it is to be noted here that the address of the desired data to be accessed must be present in the vector instruction.

This architecture enables the fetching of data of size 512 bits from memory to pipeline. However, due to high memory access time, the pipelines of the vector computer requires higher startup time, as higher time is required to initiate the vector instruction.

How Do Graphics Cards Execute Vector Instructions?

Intel announced that together with their new graphics architecture they will provide a new API, called oneAPI, that will allow to program GPU, CPU, and even FPGA in an unified way, and will support SIMD as well as SIMT mode.

1-CPU, scalar

Let's say we write a program that operates on a numerical value. The value comes from somewhere and before we pass it for further processing, we want to execute following logic: if it's negative (less than zero), increase it by 1. In C++ it may look like this:

```
float number = ...;
bool needsIncrease = number < 0.0f;
if(needsIncrease)
  number += 1.0f;
```

If you compile this code in Visual Studio 2019 for 64-bit x86 architecture, you may get following assembly

```
00007FF6474C1086 movss  xmm1,dword ptr [number]   ; xmm1 = number

00007FF6474C108C xorps  xmm0,xmm0                 ; xmm0 = 0
00007FF6474C108F comiss xmm0,xmm1                 ; compare xmm0 with xmm1,
set flags
00007FF6474C1092 jbe    main+32h (07FF6474C10A2h) ; jump to
07FF6474C10A2 depending on flags
00007FF6474C1094 addss  xmm1,dword ptr [__real@3f800000
(07FF6474C2244h)]  ; xmm1 += 1
00007FF6474C109C movss  dword ptr [number],xmm1   ; number = xmm1
00007FF6474C10A2 …
```

There is nothing special here, just normal CPU code. Each instruction operates on a single value.

2-CPU, vector:

Some time ago vector instructions were introduced to CPUs. They allow to operate on many values at a time, not just a single one. For example, the CPU vector extension called Streaming SIMD Extensions (SSE) is accessible in Visual C++ using data types like __m128 (which can store 128-bit value representing e.g. 4x 32-bit floating-point numbers) and intrinsic functions like _mm_add_ps (which can add two such variables per-component, outputting a new vector of 4 floats as a result). We call this approach Single Instruction Multiple Data (SIMD), because one instruction operates not on a single numerical value, but on a whole vector of such values in parallel

Let's say we want to implement following logic: given some vector (x, y, z, w) of 4x 32-bit floating point numbers, if its first component (x) is less than zero, increase the whole vector per-component by (1, 2, 3, 4). In Visual C++ we can implement it like this:

```
const float constant[] = {1.0f, 2.0f, 3.0f, 4.0f};
__m128 number = ...;
float x; _mm_store_ss(&x, number);
bool needsIncrease = x < 0.0f;
if(needsIncrease)
  number = _mm_add_ps(number, _mm_loadu_ps(constant));
```

Which gives following assembly:

```
00007FF7318C10CA  comiss xmm0,xmm1  ; compare xmm0 with xmm1, set
flags

00007FF7318C10CD  jbe    main+69h (07FF7318C10D9h)  ; jump to
07FF7318C10D9 depending on flags
00007FF7318C10CF  movaps xmm5,xmmword ptr [__xmm@(...)
(07FF7318C2250h)]  ; xmm5 = (1, 2, 3, 4)
00007FF7318C10D6  addps  xmm5,xmm1  ; xmm5 = xmm5 + xmm1
00007FF7318C10D9  movaps xmm0,xmm5  ; xmm0 = xmm5
```

This time `xmm` registers are used to store not just single numbers, but vectors of 4 floats. A single instruction - `addps` (as opposed to `addss` used in the previous example) adds 4 numbers from `xmm1` to 4 numbers in `xmm5`.

It may seem obvious, but it's important for future considerations to note that the condition here and the boolean variable driving it (needsIncrease) is not a vector, but a single value, calculated based on the first component of vector number.

Such a single value in the SIMD world is also called a "scalar". Based on it, the condition is true or false and the branch is taken or not, so either the whole vector is increased by (1, 2, 3, 4), or nothing happens. This is how CPUs work, because we execute just one program, with one thread, which has one instruction pointer to execute its instructions sequentially.

3-GPU:

Now let's move on from CPU world to the world of a graphic processor (GPU). Those are programmed in different languages. One of them is GLSL, used in OpenGL and Vulkan graphics APIs. In this language there is also a data type that holds 4x 32-bit floating-point numbers, called `vec4`. You can add a vector to a vector per-component using just '+' operator.

Same logic as in section 2. implemented in GLSL looks like this:

```
vec4 number = ...;

bool needsIncrease = number.x < 0.0;
if(needsIncrease)
  number += vec4(1.0, 2.0, 3.0, 4.0);
```

When you compile a shader with such code for an AMD GPU, you may see following GPU assembly

```
v_add_f32      v5, 1.0, v2      ; v5 = v2 + 1

v_add_f32      v1, 2.0, v3      ; v1 = v3 + 2
v_cmp_gt_f32   vcc, 0, v2       ; compare v2 with 0, set flags
v_cndmask_b32  v2, v2, v5, vcc  ; override v2 with v5 depending on flags
v_add_f32      v5, lit(0x40400000), v4  ; v5 = v4 + 3
v_cndmask_b32  v1, v3, v1, vcc  ; override v1 with v3 depending on flags
v_add_f32      v3, 4.0, v0      ; v3 = v0 + 4
v_cndmask_b32  v4, v4, v5, vcc  ; override v4 with v5 depending on flags
v_cndmask_b32  v3, v0, v3, vcc  ; override v3 with v0 depending on flags
```

Supercomputer

Definition :

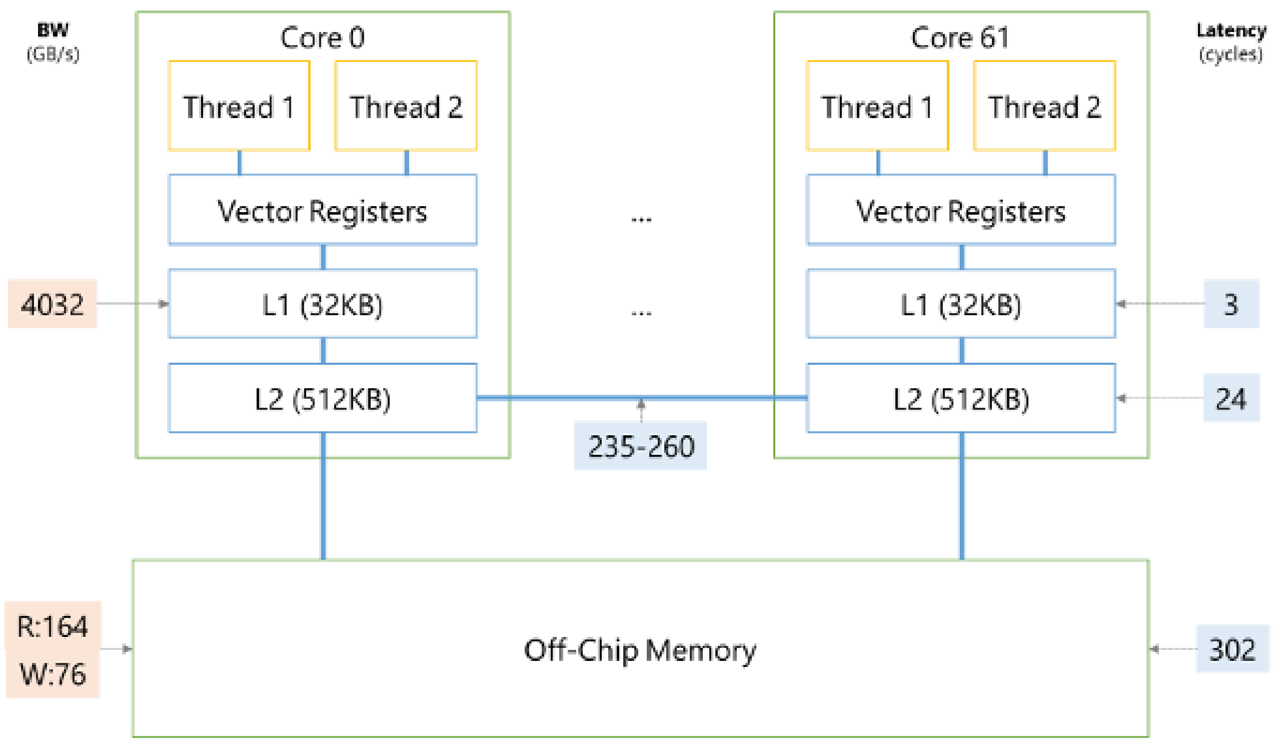
any of a class of extremely powerful computers. The term is commonly applied to the fastest high-performance systems available at any given time. Such computers have been used primarily for scientific and engineering work requiring exceedingly high-speed computations. Common applications for supercomputers include testing mathematical models for complex physical phenomena or designs, such as climate and weather, evolution of the cosmos, nuclear weapons and reactors, new chemical compounds (especially for pharmaceutical purposes), and cryptology. As the cost of supercomputing declined in the 1990s, more businesses began to use supercomputers for market research and other business-related models.

Distinguishing features:

Supercomputers have certain distinguishing features. Unlike conventional computers, they usually have more than one CPU (central processing unit), which contains circuits for interpreting program instructions and executing arithmetic and logic operations in proper sequence. The use of several CPUs to achieve high computational rates is necessitated by the physical limits of circuit technology. Electronic signals cannot travel faster than the speed of light, which thus constitutes a fundamental speed limit for signal transmission and circuit switching. This limit has almost been reached, owing to miniaturization of circuit components, dramatic reduction in the length of wires connecting circuit boards, and innovation in cooling techniques (e.g., in various supercomputer systems, processor and memory circuits are immersed in a cryogenic fluid to achieve the low temperatures at which they operate fastest). Rapid retrieval of stored data and instructions is required to support the extremely high computational speed of CPUs. Therefore, most supercomputers have a very large storage capacity, as well as a very fast input/output capability.

Still another distinguishing characteristic of supercomputers is their use of vector arithmetic—i.e., they are able to operate on pairs of lists of numbers rather than on mere pairs of numbers. For example, a typical supercomputer can multiply a list of hourly wage rates for a group of factory workers by a list of hours worked by members of that group to produce a list of dollars earned by each worker in roughly the same time that it takes a regular computer to calculate the amount earned by just one worker.

Supercomputers were originally used in applications related to national security, including nuclear weapons design and cryptography. Today they are also routinely employed by the aerospace, petroleum, and automotive industries. In addition, supercomputers have found wide application in areas involving engineering or scientific research, as, for example, in studies of the structure of subatomic particles and of the origin and nature of the universe. Supercomputers have become an indispensable tool in weather forecasting: predictions are now based on numerical models. As the cost of supercomputers declined, their use spread to the world of online gaming. In particular, the 5th through 10th fastest Chinese supercomputers in 2007 were owned by a company with online rights in China to the electronic game World of Warcraft, which sometimes had more than a million people playing together in the same gaming world.



Exp :

-in 1988 HiTech, built at Carnegie Mellon University, Pittsburgh, Pa., used 64 custom processors (one for each square on the chessboard) to become the first computer to defeat a grandmaster in a match. In February 1996 IBM's Deep Blue, using 192 custom-enhanced RS/6000 processors, was the first computer to defeat a world champion, Garry Kasparov, in a "slow" game. It was then assigned to predict the weather in Atlanta, Ga., during the 1996 Summer Olympic Games. Its successor (now with 256 custom chess processors) defeated Kasparov in a six-game return match in May 1997.

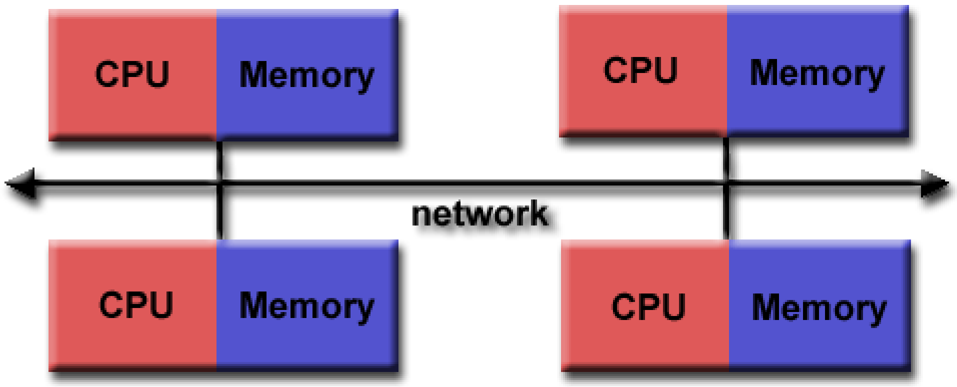
What is MPI?

An Interface Specification:
M P I = Message Passing Interface

- MPI is a ***specification*** for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
- MPI primarily addresses the ***message-passing parallel programming model:*** data is moved from the address space of one process to that of another process through cooperative operations on each process.
- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
    - Practical
    - Portable
    - Efficient
    - Flexible
- The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.x
- Interface specifications have been defined for C and Fortran90 language bindings:
    - C++ bindings from MPI-1 are removed in MPI-3
    - MPI-3 also provides support for Fortran 2003 and 2008 features
- Actual MPI library implementations differ in which version and features of the MPI standard they support. Developers/users will need to be aware of this.

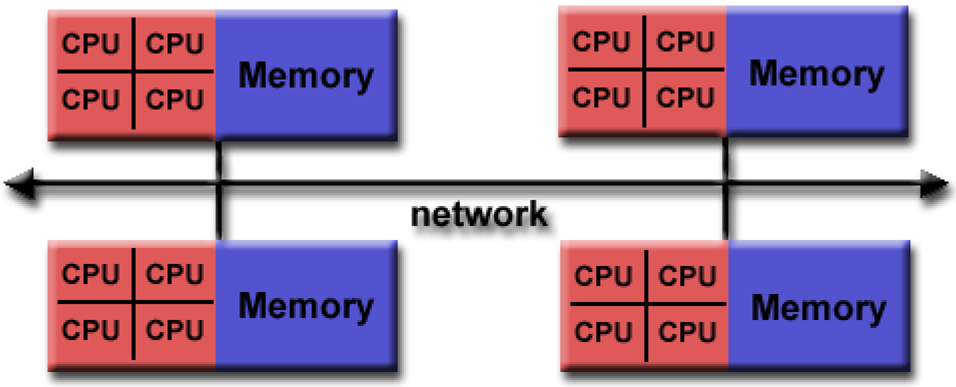
Programming Model:

Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).



As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.

MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.



Today, MPI runs on virtually any hardware platform:

- Distributed Memory
- Shared Memory
- Hybrid

- The programming model clearly_remains a distributed memory model however, regardless of the underlying physical architecture of the machine.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

Reasons for Using MPI:

- **Standardization** - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability -** There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
-
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.
- **Functionality** - There are ***over 430*** routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.

  Most MPI programs can be written using a dozen or less routines
- **Availability** - A variety of implementations are available, both vendor and public domain.

The header file contain the hello world example to print the rank of the process.

```
void sayhello(MPI_Comm comm)
    {
    int size, rank;
    MPI_Comm_size(comm, &size);
    MPI_Comm_rank(comm, &rank);
    printf("Hello, World! "
            "I am process %d of %d.¥n",
            rank, size);
      }
```

swig interface code:

```
// file: helloworld.i
%module helloworld
```

```
%{
#include "helloworld.hpp"
%}

%include /usr/local/lib/python3.6/dist-
packages/mpi4py/include/mpi4py/mpi4py.i
%mpi4py_typemap(Comm, MPI_Comm);

%include "helloworld.hpp"
```

How to compile:

```
swig -c++ -python helloworld.i
mpiCC  -O2 -fPIC -c helloworld_wrap.cxx -I
/usr/include/python3.6 -I /usr/local/lib/python3.6/dist-
packages/mpi4py/include
mpiCC  -shared helloworld_wrap.o -o _helloworld.so
```

The first line produce a helloword_wrap.cxx file.
The second line make the object file helloworld_wrap.o.
and the at the final line shared library file _helloworld.so is made.

How to use:

```
from mpi4py import MPI
import helloworld

helloworld.sayhello(MPI.COMM_WORLD)
```

```
$ mpirun -n 4 python3 runme.py
Hello, World! I am process 0 of 4.
Hello, World! I am process 1 of 4.
Hello, World! I am process 2 of 4.
Hello, World! I am process 3 of 4.
```

master 2 genie informatique

senouci fatima