Rust Parallel

Members
Dorian BARET - Lucas CHALOYARD - Victor MALOD - Yael PARA

Tutor
Olivier Richard

Department
INFO 4 - Polytech Grenoble - UGA

I. Analysis of the existing

A. Project presentation

GNU parallel is **a command-line driven utility** for Linux and other Unix-like operating systems which allows the user to execute **shell scripts** or commands in parallel.

This project is inspired by the GNU parallel software to develop **a command executor in parallel** with the **RUST language**. The language is particularly interesting for its performance and the production of robust code. RUST is a demanding language but recognized by experts.

B. Purpose

Our first objective for this project was to **learn the Rust language**. For that we read the Rust book and made the **small projects** that it integrates such as a Minigrep and a Multithreaded Web Server. In addition, we have done **a set of exercises**, called Rustlings, that serves as a great introduction to the Rust language. It covers many concepts from the Rust book by asking you to work through sets of exercises.

Once the basics of Rust were acquired, it was time to start the Rust Parallel project: **expressing the needs**, **analyzing the existing frameworks** to see if they did not do part of the work of what we wanted to implement, then **conceptualize** and finally **implement our solution**.

C. Approach of the problem

Before thinking about a solution, we had to take a deeper look at **GNU Parallel**, which has a wide variety of features. Starting with the format of a command as follow:

```
parallel [options] [command] [arguments or separators]
```

with an example:

```
parallel -j 4 echo counting {}';'wc -l {} ::: *.txt
```

Firstly, according to the format, which can be complex sometimes (with targets "{}" or separators ":::", which will be described later), we thought of using grammar, to parse a valid user input easily.

Secondly, we have decided to take only a few set of options to work with; the main ones presented by GNU in that case:

- **--jobs N** or **-j N:** uses N cores of the CPU to execute the jobs (by default, parallel uses the maximum cores available on the computer).
- **--keep-order**: forces the display of the output using the FIFO order without losing parallelism during the execution.
- **--dry-run:** prints jobs to run on standard output, but does not run them.
- **--pipe (not yet implemented):** spread input to jobs on stdin (standard input). Read a block of data from stdin (standard input) and give one block of data as input to one job.

We also had to take into account the **special characters**, especially the **separators** between the arguments ':::', which let the user create combinations of possible values. For instance the command "`parallel echo ::: S M L ::: Green Red`" creates all the **combinations** possible between one element from {S,M,L} and one element from {Green,Red} to create a couple, and prints all of them (S Green,S Red,M Green,M Red…). Besides, some special characters (**targets**) let the user control the spot of the result. For instance, the command:

```
parallel echo counting {} ';'  wc -l {} ::: *.txt
```

has the following result :

```
counting toto.txt
4 toto.txt
counting titi.txt
3 titi.txt
...
```

While learning the basics of this tool, we realised that we would need to manage:

- The creation of the **different processes**
- The saving of the **arguments** in an AST (**Abstract Syntax Tree**)
- The creation of the **combinations** between the different arguments
- The interpretation of the **special characters**, **separators** and **options**

After learning the basics of the tool, we had to choose the functionalities we wanted to implement in our project.

II. Expression of needs

A. Specifications

Just like GNU Parallel, Rust Parallel must be able to be used from the **command line** and use one or more computers. Furthermore the Rust language also allows us to provide **frameworks** (see https://crates.io/) by declaring a library. This solution is more suitable if a developer wants to use Rust Parallel in his own program.

In terms of usage, the user must be able to specify **execution options** followed by a **job**. Options allow us to run jobs in a special environment (example: number of threads). A job is a small shell script and the typical input is a list of files, a list of hosts, a list of users, a list of URLs, or a list of tables. A job can also be a command that reads from a pipe.

Rust Parallel makes sure **output from the jobs** is the same output as you would get if you would run the commands sequentially. This makes it possible to use output from Rust Parallel as **input for other programs**.

B. Main features

We have prioritized the features to be implemented. Here are the **main features** chosen :

- **parsing** of the command and **interpretation**.
- interpretation of **values** and **injection** into commands: The values must be separated from the command by "`:::`". Each of the following arguments will replace "`{}`" contained in the command (more details in the solution - interpretation section).
- **keep order option**: Keep sequence of output same as the order of input (normally the output of a job will be printed as soon as the job completes).
- **dry run option**: displays the jobs to be executed but without running them.
- **job option**: run jobs in parallel using the specified number of threads.

C. Secondary features

Other features have been kept in case all the main functionalities have been implemented:

- reading of the standard input (pipe)
- execution on a **remote machine**
- input file option: allows parallel to interpret the arguments as file and send it by ssh in case of remote execution

Based on this knowledge and what we learned about Rust, we managed to represent how we imagined the architecture of the project.

III. Solutions

A. Conception

According to Rust's ability to **create objects** from structures (which are classes that can hold methods and implement traits) we have decided to **design a simple but useful class diagram** that had the opportunity to evolve during the project's life, mostly because of technical solutions in the next section. Here is the final diagram with some quick explanations :

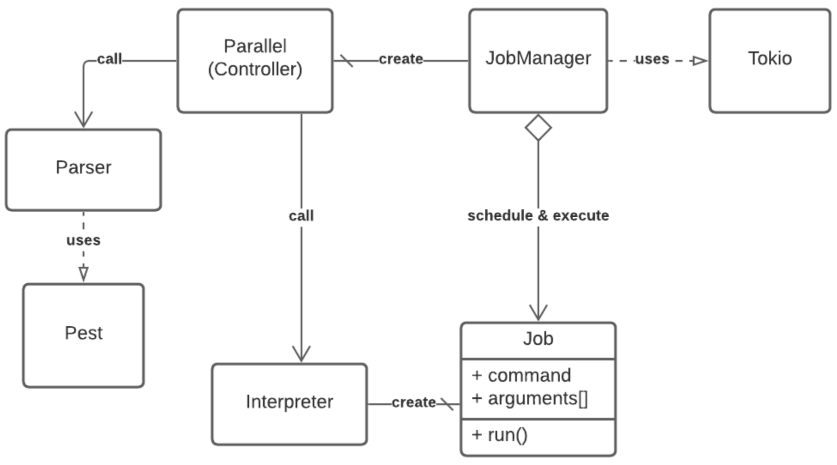

Figure 1: Class diagram conception for Rust parallel

The Controller of the engine will be the **Class Parallel**, it will firstly gather some information on the environment like the used shell and then **create the JobManager** that uses the Tokio library. It will then **call the Parser** that uses the Pest library and **gives the results to the Interpreter** that will **create the Jobs** for the JobManager and **specify options** for the overall execution scheduled by the JobManager. Parser and Interpreter are **not really objects** of classes **but more like static functionalities** to call in the process. This process is well resumed by the sequence diagram below :

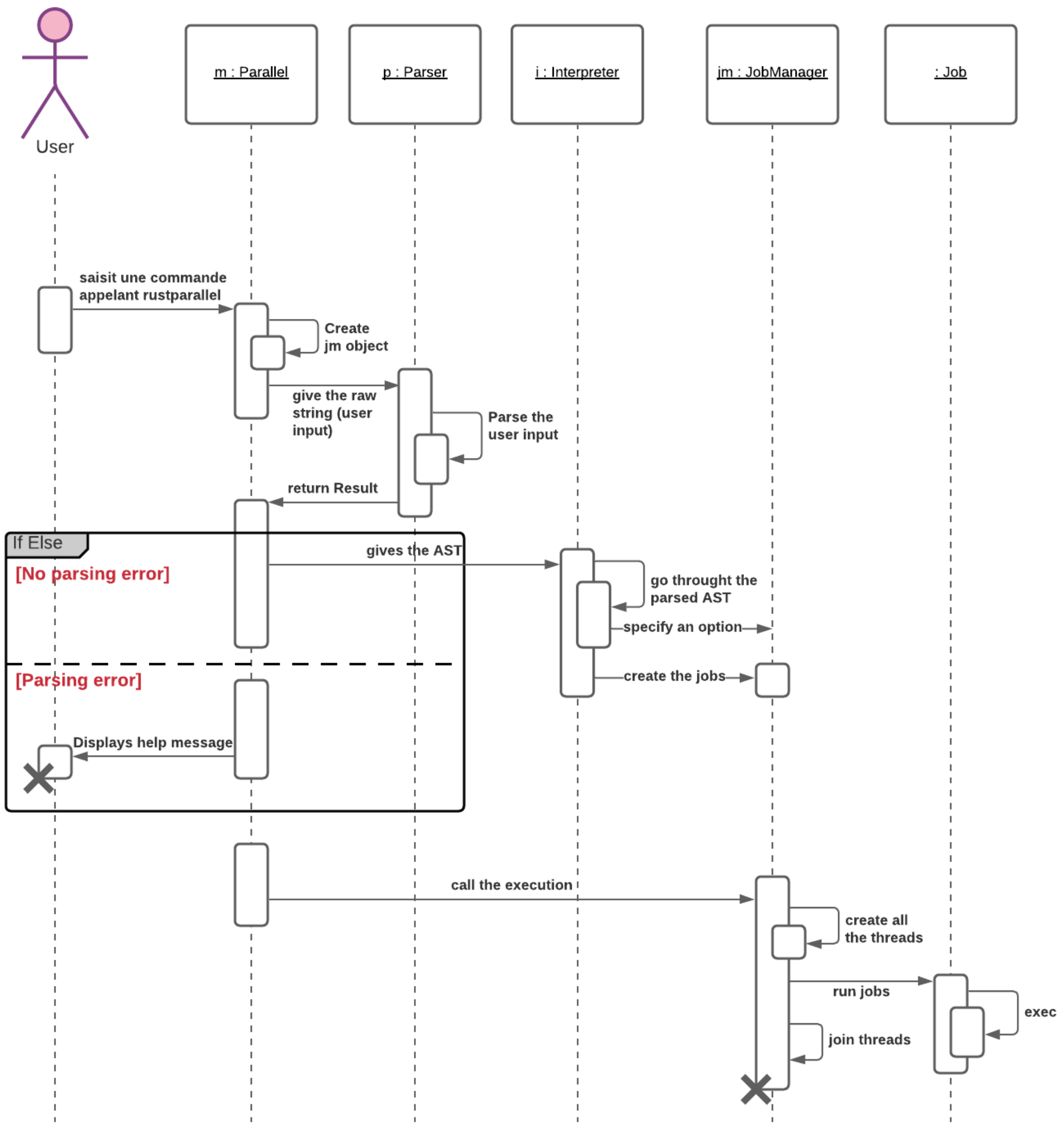

Figure 2: Sequence diagram conception for Rust parallel

The diagram is pretty much **self-explanatory** and **covers the overall process of the execution**, many details are not represented but are not necessary here to begin an approach to the code.

B. Technical Solutions

1. Parser Framework

According to **the advice** that was given to us **by our Project Teacher,** telling us that we should **use an already existing rust parser library** to parse our command line argument: we have started to search **one that would fit our needs**.

After some research on **crates.io**, we have found **two interesting parsers**, here is a table that compares them :

	Clap (Command Line Argument Parser)	**Pest**, the elegant parser
Pros	● Is a **simple-to-use**, **efficient**, and **fully-configurable** library for parsing command line arguments. ● It has a **really good error handling** and can inform users with a **friendly message** and exits gracefully.	● Has a book ● Fitting our needs because the crate is **dynamically building rules from a grammar** file of .pest format following PEG rules. ● Accessibility: **fast & easy to begin with** (one afternoon of work) and do our first tests. ● Other features: Precedence climbing / Input handling / Custom errors / **Runs on stable Rust**
Cons	● **Clap has his own grammar** designed for command line conventions ● **We can't add rules** like the separator ":::" one, without forcing really hard the crate...	● **Hard to build friendly error messages** from the parsing error.
Chosen	No	**Yes**

In the end, Clap was a good **easy-to-start-with** idea **but is not fitting** with our needs. So we moved on to **Pest**, which is meeting our needs really well by building **a nice AST** that our **interpreter will use** to create the different jobs to run.

2. Asynchronous Framework

Before considering the creation of a solution for the asynchronous part of the project, we had to choose a framework which had the features required.

Mio:
Mio is a **low-level I/O library** focusing on non-blocking APIs and event notification for building high performance I/O apps with as little overhead as possible over the OS abstractions.

It provides tools to create an **event loop**, using a mio::Poll which monitors event::Sources, waiting until one or more become "ready" for some class of operations (read/write), and it also provides **non blocking TCP/UDP**.

Tokio:
For this project, we decided to use a well-known framework called **Tokio**. Designed to write reliable network applications without compromising speed,Tokio comes with an amazing set of tools like APIs including **asynchronous code handling**, **TCP and UDP sockets**, and many others. Those tools were very useful for the creation of the Remote execution module, it allowed us to conceive an efficient asynchronous distributed application.

We also had access to a special **runtime**, designed and used by Tokio. This allowed us to run and **manage asynchronous tasks**, using **Tokio's runtime scheduler** and many other functionalities given by Tokio.

So, why did we decide to use this framework ? Well-known in the Rust community and advised by our **Project Teacher**, using Tokio assured us good performances, a lot of useful tools but also the fact that our application will not become fully deprecated until Tokio becomes it too, and due to his popularity, it will certainly not happen in the near future.

Also, those useful tools that we've talked about were used by many parts of our project and not only the remote execution part. We also use it for the **Job Manager** part that we'll discuss below.

Comparison between Tokio and Mio :

Mio can be seen as an anterior low level version of Tokio, it has less functionalities than Tokio but is way **easier to apprehend**. It let the user have the tools to create a basic event loop using **tcp & udp connections**. It misses a lot of functionalities which could be required for the project though. Tokio on the contrary has more features which are related to the project, but is harder to apprehend because it uses a lot of concepts we had to understand before using the tools it offers.

C. Development

1. Parser

The parser.rs file is very **straight forward**, it only **contains a call** to the **function parse** from Pest that given a String, will return **a result containing the AST** described by the class Pairs (also from Pest library).

The specifics lines below allow the code to load (**at compilation time**) the different Rules from our grammar defined in the parallel.pest file :

```
#[derive(Parser)]
#[grammar = "core/parallel.pest"]
pub struct ParallelParser;
```

2. Interpreter

The Interpreter can be seen as the **intermediate class** between the parser and the job manager. Indeed, it uses the AST returned by the parser in order to **create jobs**, which are sent to the **job manager**. It will also tell the job manager if some **options** are specified.

The Interpreter does not have a structure, it offers only functions it has in its file to **interpret** so it can be seen as a **local library**.

The first function used is **interpret()**, which takes as arguments the Pairs<Rule> that describes the command of the user according to the grammar, it is the AST. The purpose of the function is to **detect the options, client/server addresses, separators and commands** using the grammar in order to create jobs. After that, these jobs are added to the job manager's list. If successful, it sets the environment using the job manager. Otherwise, it returns an error if there is an issue with the analysis of the elements of the command. This function uses other sub-functions, which will be covered in the following paragraphs.

Indeed, interpret() uses the sub-function **build_combinations()** in order to create all the jobs according to the options and the separators which have been detected. It uses as argument a vector containing the values between the different separators.

Finally, interpret() calls **create_all_jobs()**, which manages the opening/closing brace characters '{' and '}' in order to modify the jobs that have been created using build_combinations(). At the end of this function, the jobs are pushed inside the job manager.

In the next part, we will focus on the structure of the jobs which are created using the previous functions.

3. Job & Job Manager

Job

A job is the equivalent of a **shell script** to be executed. Here is its structure :

```
pub struct Job {
      cmd: String,
      parameter: Vec<String>,
}
```

The *cmd* attribute is the name of the **Unix command**, the *parameter* attribute is the list of **parameters** to give when executing the command. Typically, `cmd` will be "/bin/sh" or "/bin/bash", etc.

We have provided this structure with methods such as a constructor and an execution function. The **constructor** allows us to initialize an instance of the Job structure from a list of words.

The execution function allows to create a **Tokio Command** in order to execute the command in the **tokio runtime**. Finally it **waits** for the **pipe of the command output** to be created in order to return it. If it is not created then the command is not good and returns an error.

Job Manager

The job manager allows us to **manage the execution environment** and the **jobs**. Here is its structure **simplified**:

```
pub struct JobManager {
      pub shell: String, //the shell used to launch jobs
      cmds: Vec<Job>,
      nb_threads: Option<usize>,
      dry_run: bool,
      keep_order: bool,
      // -- snip --
}
```

The *cmds* attribute is the list of commands (jobs) to be executed, *nb_threads* is the number of threads to be used in the execution environment, *dry_run* is true if the dry run option has been specified by the user, as for *keep_order.*

As for Job, JobManager has a **constructor** to initialize the attributes with default values. The structure also has a function to **add a command** to the list, a function to **change the execution options** and functions to **execute all the jobs** in the set environment and wait for the results.

4. Remote execution

For the **remote execution** functionality, we decided to use a standard **Client/Server architecture**.

Let’s quickly review how it is designed.

Firstly, we have the **Client**, when an user wants to use the remote execution option (by typing a command like “`parallel --client 127.0.0.1 8080 echo {1} {2} ::: a b c ::: 1 2 3`”.

By doing this, the **Client** will parse the command, and analyze the request. If a remote execution is asked, we’ll launch the **Client-side** of the remote execution. We decided to fully **parse** the command before **sending** the request to the **Server** in order to not allow the possibility to send incorrect requests and to disable a possibility of Denial Of Service.

Then, we have the **Server side**, it’ll use the following principle: **When the server receives a request sent by a Client, it will create a Worker which will be connected to the Client and will handle his request**. By doing so we can handle several Clients at the same time.

Those **Workers** and the **Client** that they are handling, will communicate using a **Channel** designed and implemented by us.

This **Channel** will work using **Tokio's asynchronous API, and the principle of interest.** Each phase of the communication will have an associated interest, for example when we want the **Client's channel** to read some data, we'll have to give it the **"READABLE" interest**. Using this technique allows us to only read or/and write when it's necessary, and we can also use the **asynchronous I/O functions** so **non-blocking functions** which usually improve the performance.

Each Channel (on the Client and the Server side) will have a **Channel Listener** (which will be the Client object for the Client side, and the Worker object for the Server side) on which we'll call the functions **sent** when a message has been fully sent and **received** when a message has been fully received. By following this pattern, we can let the **Channel** handle the communication itself and the **Channel Listener** will use the data exchanged to do his job.

Currently our application exchanges very explicit messages (String) in order to know in which state is each side of the application. But we could easily use Integer to do so, we decided to use those explicit messages in the beginning in order to facilitate us the debugging of the Remote execution (it's easier to analyze the exchange when the packet contains information understable by the human).

5. Entry of the tool

The entry point of our tool is via the Parallel structure that follows :

```
pub struct Parallel {
      job_manager: JobManager,
      command: String,
}
```

This structure is composed of 2 attributes: *job_manager* which is an instance of JobManager (explained above), *command* is the string entered by the user.

This structure allows to **parse**, **interpret** the command and **launch the execution** by the JobManager.

6. Other

In this section, we will add some additional information about the development of our tool.

The Job and JobManager structures inherit the **std::fmt::Display trait**. This trait allows us to define the description of a structure instance. This is particularly useful when you want to display the description of an instance on the standard output.

We used the **env_logger and log frameworks**. These frameworks are logger frameworks, they allow to display one or more information on one standard output (stdout, stderr, ..) in developer mode. So that frameworks macros can be used, it is necessary to add an environment variable during the compilation or the execution of the program.

Example :

```
RUST_LOG=info ./main       for displaying lines with the macro info!(...)

RUST_LOG=debug ./main      for displaying lines with the macro debug!(...)
```

To test our functions or to make simple tests, we have set up many **tests** in the files concerned by them.

To facilitate the development and manage the executions via cargo we used the cargo package named **cargo-make**. Like GNU Make, the cargo-make **task runner** enables to define and configure sets of tasks and run them as a flow.

D. Organization

For this project we needed to organize ourselves in an efficient way.

We were asked to split the **documentation** part of the project (**tracking sheet**, **technological watch**, **conventions**, **specifications**, etc…) from the **source code**, so we created two repositories, one called **docs** containing all the documentation and the other called **rust-parallel** containing the project itself (generated by **cargo**).

Then, during the learning part of our project, we used a document shared using google drive to note some remarks, indications, and advice about the **Rust** language.

This helps us a lot because it allows each one of us to learn when he wants it, and still to have access to some advice given by the others.

We also used **gitlab issues** to keep an eye on issues we encountered and that we didn't succeed to fix alone or that we didn't have the time to fix after the discovery of the **issue**.

E. Distribution of tasks

At the beginning of the project, everyone had the same task, learning **Rust** using the **Rustlings** application and the **Rust Book**.

After we've finished with this task, we split into two groups, one needed to learn about **Frameworks** we could use for our project, and the other had to learn what does **GNU Parallel** and how it works, then they would have to recap briefly to the other group what they learnt about **GNU Parallel**.

Then came the development part of the project. For this phase of the project we decided to give each one of us a specific task. We split it into 4 main tasks; **Job/Job Manager**, **Parser**, **Interpreter** and **Remote Execution**.

This kind of organization needed us to be very careful on the main architecture of our code, and how each of the 4 previous parts will communicate between them.

IV. Statement

A. Done

During this project, we achieved to implement **all main features** (parsing interpretation, values injection, keep-order/dry-run/job options). Moreover we began to implement the secondary features and especially, we achieved the **remote option**.

We have set up **many tests** of our code and performance tests, the results of which you can see below.

We managed to implement the basics of the project, which can be enhanced with other features.

B. Upgrade

Many upgrades can be done on our project, compared to the **"real" GNU Parallel,** we haven't implemented a lot of functionalities. Our main goal was to produce an application that could be the core of a real GNU Parallel written entirely in **Rust**.

But, if at the moment, we could have a few weeks more, we would focus on the upgrades below.

Remote exec part:

The implementation of the **Remote Execution** could be upgraded in a few ways.

Firstly, the remote execution can't be used with the **--jobs** option for the moment. Because we have to create the **Tokio's runtime** at the Server creation, we cannot decide a maximal number of threads running for the execution of the request. This could be an interesting feature to add.

Secondly, we could (and should) modify the messages exchanged during a remote execution. At the moment, to exchange information about the current state of each side of the remote execution we send very explicit messages (String) despite the fact that we could send integers with a linked meaning. If we modify that, we could decrease the amount of data sent on the network.

And finally, we could also modify the Channel object in order to make it more efficient and usable more easily.

C. Performance

So what about performances, Parallel is used to reduce the time taken by the execution of a task usually done in a sequential way.

In order for us to have a better idea of how much this tool can be powerful, we've designed a simple test.

We wrote this script (named PasswordCrack.sh) :

```
#!/bin/bash
if [ "$1" = "534246" ]; then
    echo "You succeed to find $1"
fi
```

This script prints a message telling the user that the argument he has given to the program as an argument is equal to the value “534246”. We chose this kind of test because it is possible with this to simulate a Brute Force task.

Our goal will be to measure the time took by the two following commands :

- `./rust_parallel --jobs 6 bash PasswordCrack.sh {1}{2}{3}{4}{5}{6} ::: 1 2 3 4 5 6 ::: 1 2 3 4 5 6 ::: 1 2 3 4 5 6 ::: 1 2 3 4 5 6 ::: 1 2 3 4 5 6 ::: 1 2 3 4 5 6`
- ``for i in `seq 111111 666666`; do ./PasswordCrack.sh $i; done``

Those two commands will do the same operations but the first command uses our **Rust Parallel** in order to parallelize this task using 6 jobs (so 6 threads), contrary to the second one which will try each possible passcode one by one.

For the commands above we get the following results using the “**time**” tool :
Rust Parallel :
```
real   0m16.199s
user   1m25.678s
sys    0m54.129s
```
Sequential *(simple bash script)* **:**
```
real   28m56.836s
user  10m51.210s
sys    18m34.264s
```
GNU Parallel :
```
real  1m31.476s
user  1m50.453s
sys   1m33.664s
```

We also did it with a 5 digit code, and we got the following result :
Rust Parallel :
```
real   0m1.044s
user   0m5.461s
sys    0m3.385s
```
Sequential :
```
real   2m22.396s
user   1m15.335s
sys    1m11.196s
```
GNU Parallel :
```
real  0m5.652s
user  0m7.585s
sys   0m5.334s
```

As we can see the difference of time taken by the task for each method is incredibly significant.

V. Conclusion

To conclude, this project allowed us to discover a new language which is more and more used by big companies like Firefox, Google, Amazon and Microsoft. This language also allowed us to discover new concepts like ownership.

We were able to accomplish everything we had planned in time but there are still many features present in GNU Parallel that could be implemented in this project.