Distributed Allocator
Using Message Passing Interface

Thibaut Barroyer, Hamza Senhaji-Rhazi, Axel Mendoza

December 21, 2017

1 Introduction

Nowadays, hundreds of millions of people are constantly taking photos, making videos and sending text messages all across the world. On the other hand, public and private company are collecting those data to build consumer preference systems, purchase and trend analysis. The government regularly collect every kind of data from census data to incident reports from police departments. This data deluge is growing exponentially. The total amount of data in the world was 4.4 zettabytes in 2013. That is set to rise steeply to 44 zettabytes by 2020. To put it in perspective, one zettabyte is equivalent to 44 trillion gigabytes. Even if we are improving the hardware performance, we can't treat efficiently this amount of data just by using standard algorithms. This is where distributed algorithms comes in. It tries to figure out how efficiently several autonomous computational entities (computers or nodes) could work together to achieve a goal than are unable to complete alone. All these entities have their own local memory, and communicate with each other by message passing in order to achieve a this common goal.

2 Message Passing Interface

The distributed allocator discussed in this paper implements the "Message Passing" communication protocol. This protocol is the basis of our model of concurrency. Indeed, MP is a modern way for a program to have several different processes working together. The processes send messages to each others to have specific triggers in order to execute a specific code associated with some tag filled in those messages. Within this communication environment, there is an important distinction among synchronous and asynchronous message passing. Two processes are synchronous when they are running at the same time and waiting each other for a response. While asynchronous interaction allow processes to be busy or not running when they receive a request. Both type of operations are used in this project, and will be discussed later.

This communication protocol allows parallel programming and multi-user handling. It can be very useful for heavy computational execution that can't be performed by a single machine. It also speeds up performance, by using all available threads on a single machine, which is a big difference comparing to non-parallel algorithms. Unfortunately, message passing is very network dependent. If the network is slow the synchronous communications will slow down the process.

Concerning the distributed allocator in this communication protocol, the processes have to act as if they have a shared memory. More in details, they have to work together to perform $alloc$, $read$, $write$ and $free$ requests sent by the user through the API.

Theses rules are mandatory to have a well working distributed allocator:

⇒ Each process can perform up to one send and one receive per loop to avoid chaotic behaviors

⇒ Every process must know where a variable is stored

⇒ A process can ask another one to read a variable

⇒ Same operation with write

⇒ Same operation with free

To complete this mandatory we used the architecture explained in the section below.

3 Network Architecture

This diagram summarize the chosen architecture. Each process have:

⇒ a thread receiver

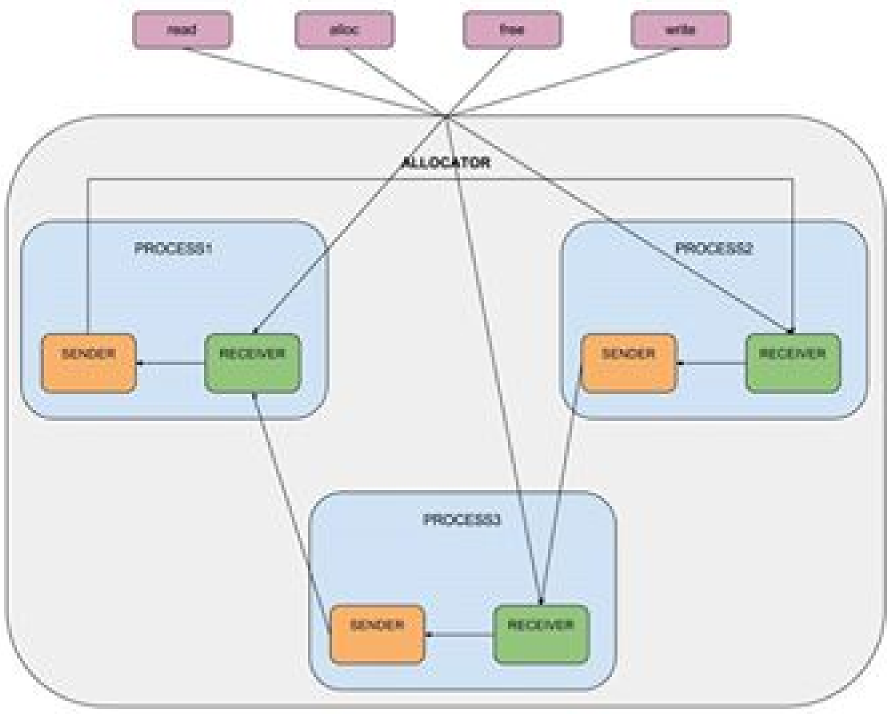
⇒ a thread sender

⇒ a local collection of data

⇒ a local list of the freed data ids

The user can communicate with all the receivers. We use the threading approach to have the processes asynchronous compared to the receiver. The processes have two queue, one to store the received messages and another to store the message the process want to send. A queue is a good approach for handling numerous messages arriving at the same time, so that we can perform each request sequentially. The topology of this network avoid simultaneous forbidden execution, like two writes at the same time.

We will now present the Open MPI API that was used in the implementation of the allocator.



4 Open MPI

Open MPI is a C, C++ and Fortran message passing interface that became the standard nowadays. It is a protable and scalable API for high-performance computing. Open MPI works with many configurations. The user only need to specify how many processes he wants to use and the code will have the same behavior. MPI implementations also include a run-time environment, the users need to start processes on multiple servers simultaneously.

We will now discuss the basic methods to have a deeper understanding of how MPI works:

⇒ MPI_init call initialize all the processes

⇒ MPI_close terminate all the processes

⇒ MPI_Comm_size returns the number of process

⇒ MPI_COMM_WORLD encloses all of the processes in a specified job.

⇒ MPI_Comm_rank return the unique id of the current process

⇒ MPI_Isend send a message to a process

⇒ MPI_Recv gets a incoming message

5 Message Passing Rules Implementation

5.1 Each process can perform up to one send and one receive per loop

Because each process have a queue of messages and is sending or responding to the other processes sequentially one time per loop. This rules is respected.

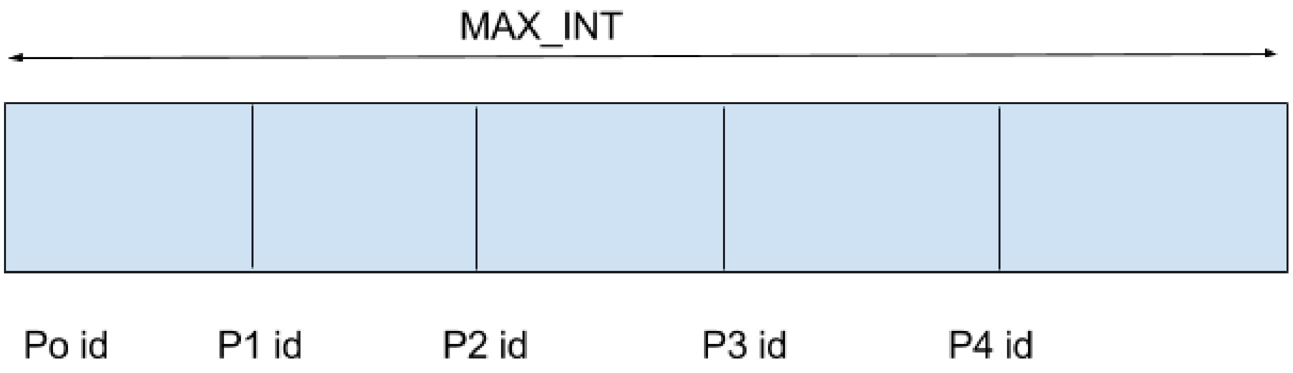
5.2 A process can ask another one to read, write, free, a variable

When we initialize the processes, we divide MAX_INT by the number of processes.

The example below illustrates an example with four processes. We implemented this solution because each process can know from which process a variable belong simply by computing the inverse calculation:

$process\_id = world\_rank * (MAX\_INT/nb\_process)$

So when a process need to *read*, *write* or *free* an id that did not belong to him, he just send a message to process_id and wait for the response.



6 Allocator methods and number of messages

6.1 Allocate a single variable

This method ask a process to allocate a single int. When *alloc*() is called by a user in the main program, the current process check if he has enough space, if it is the case, an *alloc*() is performed in the current process. The concerned process first check if there is a freed item in his local free list, he allocates in this free space or at the end of his local collection if no variable have been freed. If there's no space in the current process, a message is sequentially sent with the TAG=66 to all the others processes. If the asking process receives an acknowledge, he breaks the loop and wait for the returned *id*.

The number of messages is:

- ⇒ 0 if there's space to allocate on the concerned process
- ⇒ $nb\_process * 2$ if no process have space available, because each asked process returns a message to the requesting process

6.2 Write method

This process who received the *write* request check if the input *id* is in his local memory. If it is, he locks the value and *write* to it, if not, he sent a message to the process holding the variable to perform the *write* operation.

The number of messages is:

- ⇒ 0 if the id doesn't exist or the variable is in the local memory of the process
- ⇒ 2 if another process is holding the variable

6.3 Free a single value

The concerned process check if the *id* is stored in his local memory. If it is, he locks the variable and add it to his local free list.

The number of messages is:

- ⇒ 0 if the *id* doesn't exist or if the *id* is in the current process
- ⇒ 2 if another process is holding the variable

6.4 Allocate a collection of variables

Our local collection on each process has the following map structure: $\{id\{value, next\}\}$. When we want to allocate a collection, we call *alloc*() in a loop "size" times, and we chunk the different values with the next value.

During the loop on the size that we are trying to allocate, if the max-size for the current process is exceeded, a call is made for the next process: $alloc(size - allocated\_size)$ to continue the allocation. The current process switch his state to synchrone with the requested process and wait for the returned id to link it to the end of his allocation.

The number of messages is:

- ⇒ 0 if the asked process has enough space on his local memory

- ⇒ The number of messages is $2 * nb\_process$ in the worth case: if the allocated size have to be divided between all processes.

6.5 Loop through a collection

The *next* method is used to get the next id of a collection. When a process is asked the next of an *id*, he checks if the asked *id* is in his memory. If it is, he returns his next. If it is not, he asks the concerned process for the next value.

- ⇒ 0 if the collection is fully in the local memory
- ⇒ If each link of *ids* is between different processes we call $2 * size$

6.6 Free a collection

Given the head of a collection, we loop to the item of the collection, and add the ids to the local free list of the concerned processes.

- ⇒ 0 if the collection is fully in the local memory
- ⇒ As we call $2 * size$, plus $size * 2$ if every link of the collection is between process. So the number of messages is $4 * size$

7 Use case

- ⇒ MAX INT = 10
- ⇒ World size = 3

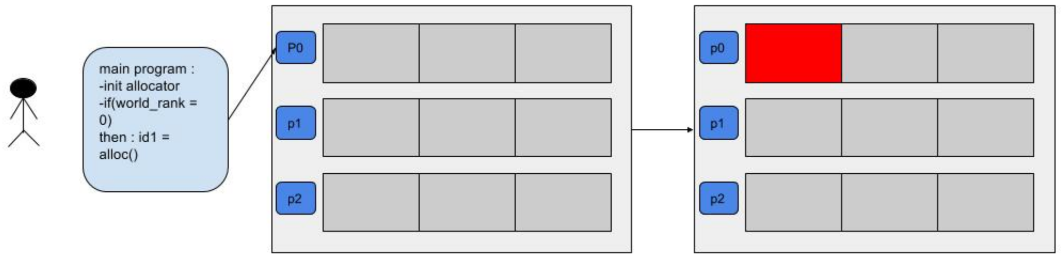


Figure 1: The user call alloc()

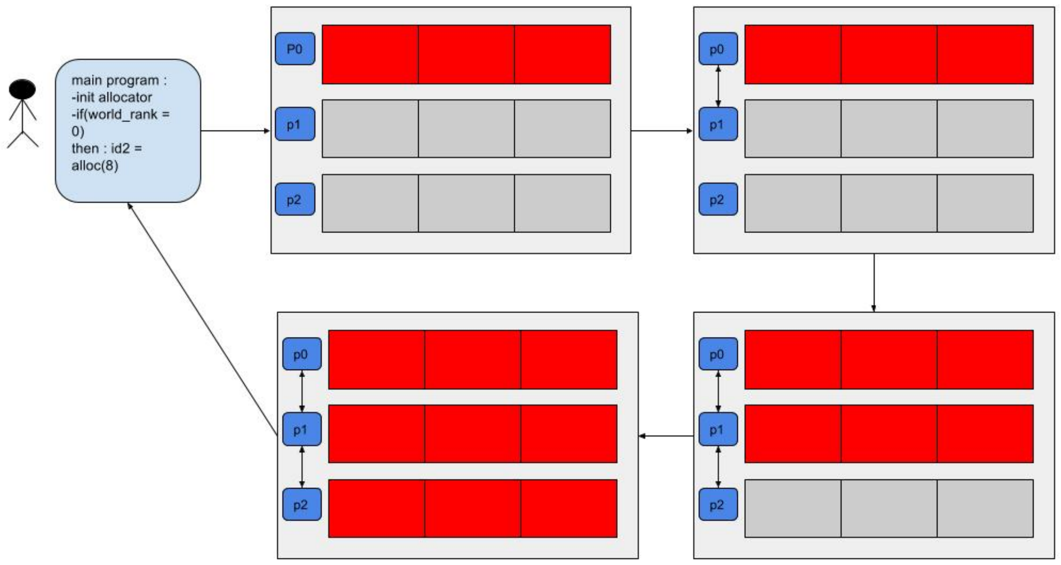


Figure 2: The user call alloc(8)

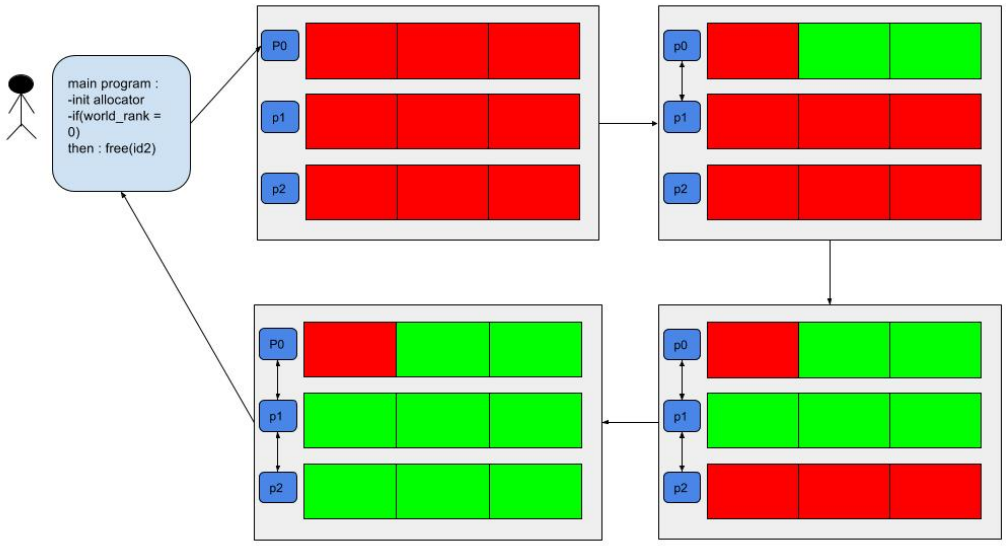


Figure 3: the user call free(id2)

8 Test Experiment

To test our API, we have implemented a quick sort of an array of size 3200.
The array is shared between 4 processes and each process allocates 800 cells of the array.
The process 0 is the master of the application and made the sort by accessing and writing on other process memory.
The application took 1.9 seconds to execute on an i7 3770K.

9 Conclusion

Implementing a distributed allocator is a very interesting project to have a deeper understanding of parallel programming using Message Passing Interface. Thinking about the network architecture and the way processes have to work with each other to process we a challenging problem. Unfortunately our allocator is not strong enough to handle big numbers. We also didn't have the time to implement an optimized version. But still it was a exciting subject about how to shared memory between numerous processes.