Lab 1 - Report

1. EXPLANATION OF PART 3 - HELLO, WORLD!

When we run *hello.c* on the AWS Server with *n* processes, the sentence "Hello, I am *y* of *n* processes." is printed for every *y* between 0 and n-1. The interesting part is the order in which the sentences are printed. After running the command `mpirun -np 4 -hostfile hosts hello` a few times, the results vary:

Hello, I am *1* of 4 processes	Hello, I am 3 of 4 processes	Hello, I am 0 of 4 processes
Hello, I am *2* of 4 processes	Hello, I am *2* of 4 processes	Hello, I am *2* of 4 processes
Hello, I am *3* of 4 processes	Hello, I am *1* of 4 processes	Hello, I am *1* of 4 processes
Hello, I am *0* of 4 processes	Hello, I am *0* of 4 processes	Hello, I am 3 of 4 processes

The order of the processes seems to be random. This is probably because the amount of time it takes for a process to finish is dependent on what else the processor its running on has to deal with, and that appears random to us. As we vary the number of processes from 4 all the way up to 100, the pattern is more or less consistent.

The only thing to note is that since our AWS Server has 8 processors on which to actually do work (as noted in the *hosts* file), when we ran the file with more than 8 processes, we actually expected that the first 8 would finish in random order, then the next 8 would finish in random order, etc. What we found, however, is that while lower number processes tend to finish earlier, the overall order is still mostly random. This is probably because a processor does not wait for other processors to finish their work before picking up another process.

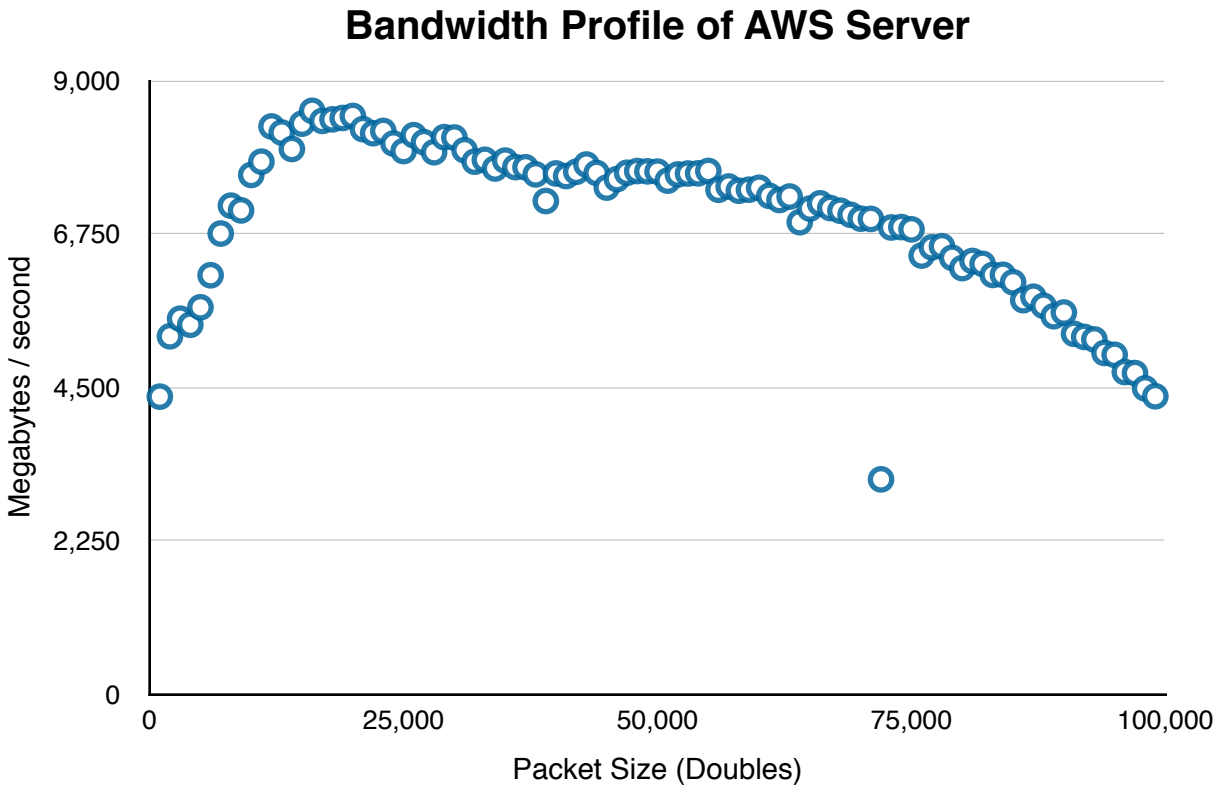
2. EXPLANATION OF PART 4 - MACHINE PROFILING

Profile Summary

Before proceeding to explain the process of finding each of the parameters, we present the profile of our AWS Server consisting of its message latency, its message bandwidth as a function of packet size, and its total floating point operations per second as a function of the number of processes employed.

- **Latency:** ~350-520 nanoseconds. The precise number varied every time we ran the code.
- **Bandwidth:** The bandwidth linearly increased to around 8100 Megabytes/second, when the packet size was around 30,000 doubles. It then decreased linearly all the way to 4500 Megabytes/second

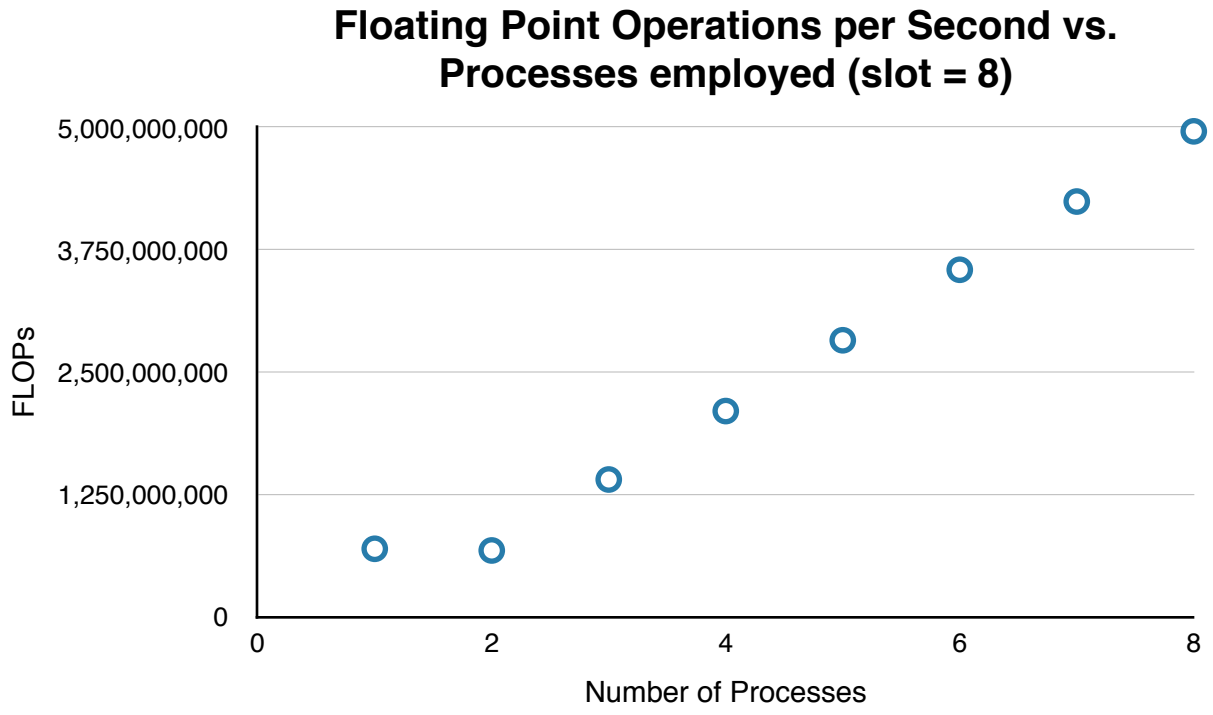
with a packet size of 990,000 doubles. We suspect this is because messages have a size limit of around 30,000 doubles. When we increased past that size, sending the payload required multiple messages, which increased overhead.

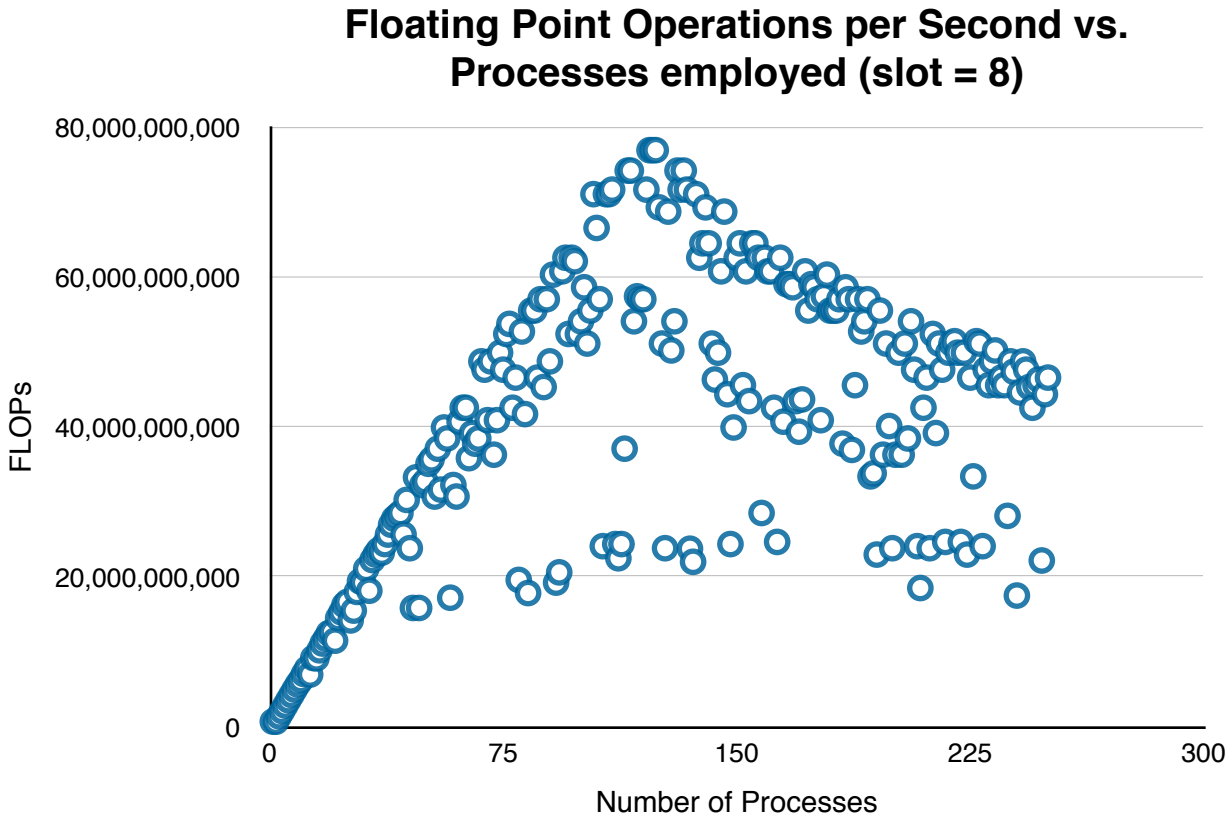


- **Floating Point Operations per Second**: The total number of FLOPs increased linearly as we varied the number of processes employed, until reaching a cap. Even though we indicated that the available slots (processors) to use were 8, we pushed the number of processes all the way to 250. We expected that after 8 processes running in parallel, the FLOPs would decrease (since certain operations would have to wait). However, to our surprise, the FlOPs kept increasing until around 145 processes were used. Looking at Open MPI FAQ, it mentions that this is referred to as *oversubscribing*, and that:

> If the default slot count is exhausted on all nodes while there are still processes to be scheduled, Open MPI will loop through the list of nodes again and try to schedule one more process to each node until all processes are scheduled.

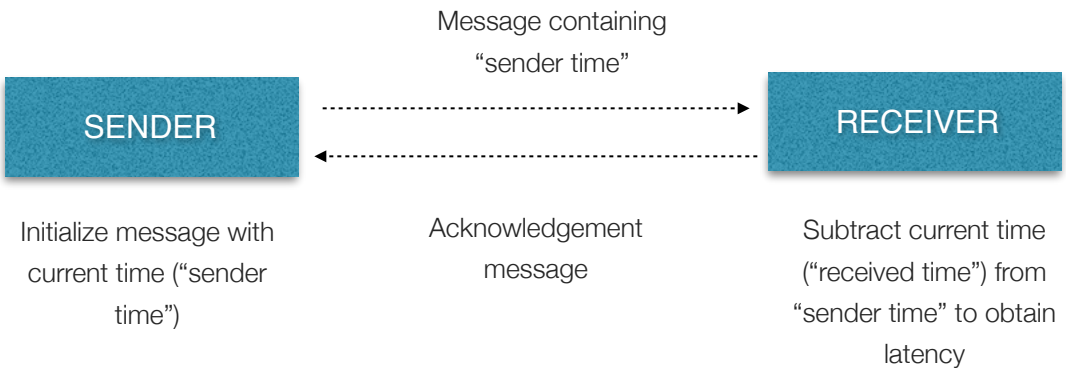
It is very surprising then that we still notice improvement after 8 processes, a possible reason for this to happen is simulated parallelism by the OS, though this is just a hypothesis.


Floating Point Operations per Second vs. Processes employed (slot = 8)
5,000,000,000
3,750,000,000
2,500,000,000
1,250,000,000
0
FLOPs
0
2
4
6
8
Number of Processes


Floating Point Operations per Second vs. Processes employed (slot = 8)
80,000,000,000
60,000,000,000
40,000,000,000
20,000,000,000
0
FLOPs
0
75
150
225
300
Number of Processes

Calculating Message Latency

The high level idea behind calculating the message latency between 2 processes is the following (please note that a lot more thought is involved, explained below):



A more detailed process follows:

- After initializing the MPI library, retrieving the global size and and retrieving the rank of each process, we first make sure that we have enough processes to calculate latency (2 processes minimum). After that, we set up a *SENDER* process and a *RECEIVER* process. We also set a determined number of latency tests as a macro, in order to get an average latency.
- The *SENDER* process behaves in the following way:
  - For a given number of tests, it:
    1. Logs the current time
    2. Sends the time to the *RECEIVER* process
    3. Waits for an acknowledgement from the *RECEIVER* process before starting over. This is important, otherwise the *SENDER* process would just keep sending messages while the *RECEIVER* is performing other previous operations.
- The *RECEIVER* process behaves in the following way:
  - It initializes a *message_latency* variable to 0, and then, for a given number of tests, it:
    1. Receives the message from *SENDER*
    2. Logs the time at which it received it
    3. Calculates the time elapsed between sending and receiving
    4. Adds the elapsed time to the latency variable

5. Sends an acknowledgement message to the *SENDER*.

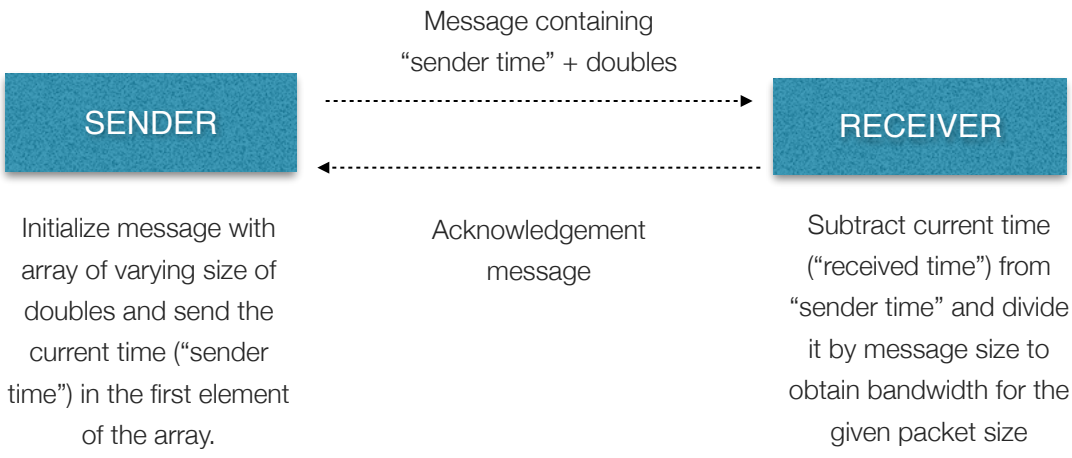
- At the end, the *RECEIVER* divides the total time elapsed by the number of tests, and then prints the value to the standard output. This is the latency.

Important notes:

- The latency we obtained was based on the average latency of 100,000 tests. And lies between ~350-520 nanoseconds.
- The number of tests can be modified accessing the *latency.c* file and changing the *NUM_TESTS* macro.
- In order to debug the program, we print in the output certain messages using a defined function called *debug_print*. This function is only called by compiling in *debugging mode*, which can be done by: `mpicc latency.c -o latency_check -D DEBUG`

Calculating Message Bandwidth

The high level idea behind calculating the message bandwidth between 2 processes is the following (please note that a lot more thought is involved, explained below):



A more detailed process follows:

- After initializing the MPI library, retrieving the global size and and retrieving the rank of each process, we first make sure that we have enough processes to calculate latency (2 processes minimum). After that, we set up a *SENDER* process and a *RECEIVER* process. We also set a determined number of bandwidth tests as a macro (*NUM_TESTS*), the value over which we increase the packet size in every test (*PACKET_STEP*), and the number of messages to send in every iteration of the test (*NUM_MGS*) in order to get an average bandwidth for every packet size.

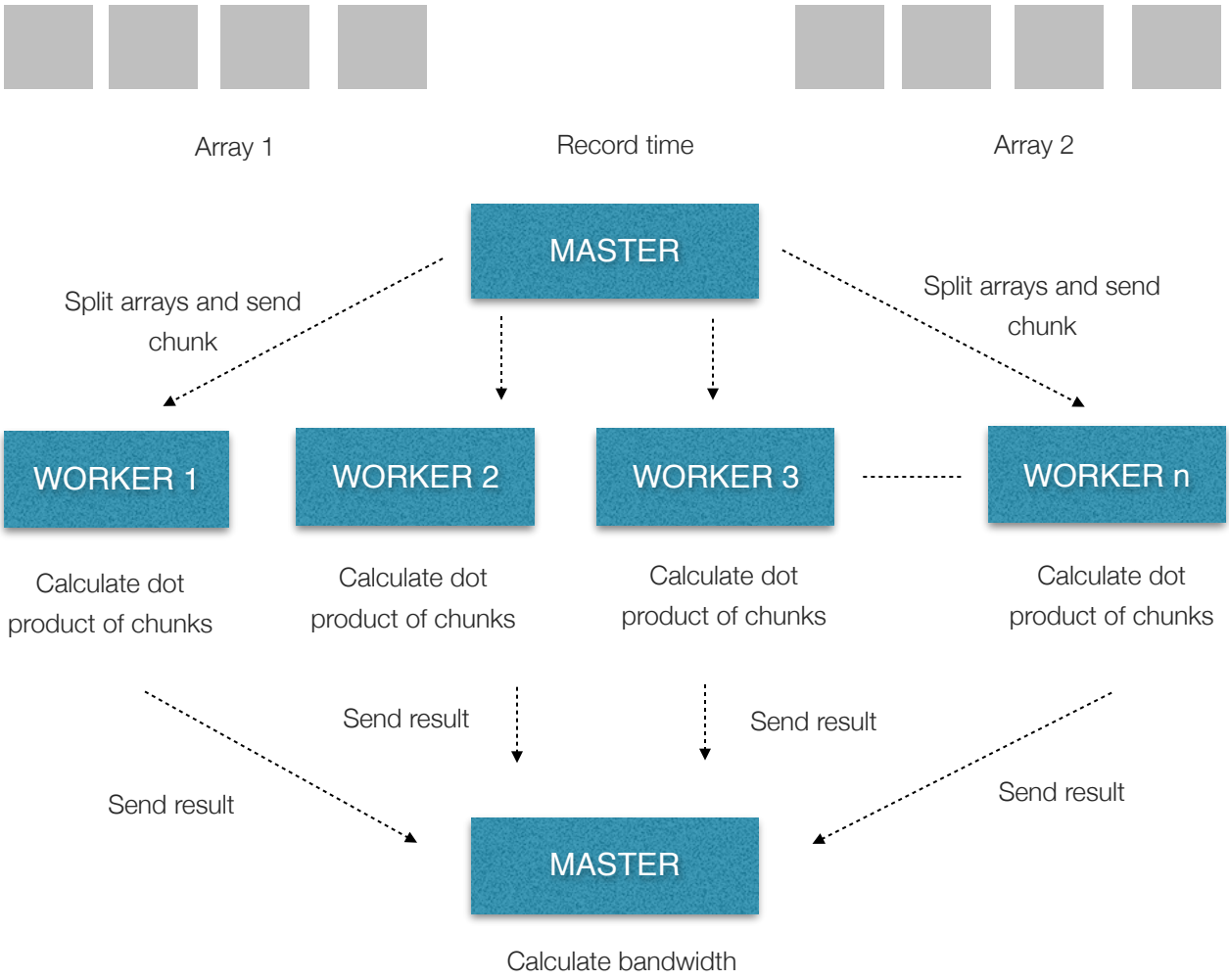
- The *SENDER* process behaves in the following way:
  - For a given packet size (number of doubles), it:
    - For a given number of tests, it:
      1. Creates a message with the given number of tests, initializing each to 0.0
      2. It logs the time at which it sends the message, and saves to it to 0th index of the message it is about to send
      3. Sends the message to the *RECEIVER*
      4. Waits for an acknowledgement from the *RECEIVER* before starting over
- The *RECEIVER* process behaves in the following way:
  - For a given packet size (number of doubles), it:
      1. For a given number of tests, it:
         a) Initializes a *message_latency* variable to 0
         b) It then calculates the time it takes to send a packet of the given size as follows:
            i) Receives the message from *SENDER*
            ii) Logs the time at which it received it
            iii) Calculates the time elapsed between sending and receiving
            iv) Adds the elapsed time to the latency variable
            v) Sends an acknowledgement message to the *SENDER*
      2. It calculates the average time to send a message with the given packet size by dividing the total time spent elapsed to send all messages (*message_latency*) by the number of messages sent (*NUM_MGS*).
      3. It updates the results array with a (packet size, average time to send a such a packet) duple.
  - After all the tests have been run for each packet size, it iterates over the results array and prints out (packet size, bandwidth) duples. Note that the bandwidth is calculated by a function called megabytesPerSecond.

Important notes:

- Both the *SENDER* and the *RECEIVER* increase their packet size from 1000 to (1000 * 100) in steps of 1000 in our specific implementation. However, the packet size step and the number of tests (in this case, 100), can be varied via macros.
- We obtained a maximum bandwidth of 8100 megabytes/second for 30,000 doubles, after which it started to decrease.
- In order to debug the program, we print in the output certain messages using a defined function called *debug_print*. This function is only called by compiling in *debugging mode*, which can be done by: `mpicc bandwidth.c -o bandwidth -D DEBUG`

Calculating FLOPS

The high level idea behind calculating the floating point operations per second using dot product for a given number of processes *n* is the following (please note that a lot more thought is involved, explained below).



A more detailed process follows:

- After initializing the MPI library, retrieving the global size and and retrieving the rank of each process, we set up a master worker architecture. The *MASTER* (process id = 0) initializes the vectors for which the *WORKERS* will calculate the dot product and then splits them into chunks that are sent to the *WORKERS*. The *WORKERS* calculate the dot product for their part and send it back to the *MASTER*, which then adds up all the results.
- In the case that we run the code on just 1 process, the code runs slightly differently. The *MASTER* does not chunk up the work or send any messages Instead, it simply initializes the arrays, calculates the dot product, and returns the FLOPS.
- To actually calculate the FLOPS, it logs the time at which it sends the first work message and then once it receives the last result message it again logs the time. In the 1 processor case, the start and end times are logged before and after the dot product calculation. We subtract the former from the latter to get the elapsed time in seconds. We calculate the total floating point operations per second as follows:

$$\text{FLOPS} = (\text{size of vector} * 2) / \text{elapsed time}$$

- We multiply by 2 because for each element in the arrays we do one multiplication and one addition.
- The *MASTER* then prints the total FLOPS to the standard output.

Important notes:

- The array size for the dot product can be indicated as a macro, by modifying the *ARRAY_SIZE* variable. In our case, we used an array of size 1,000,000.
- Without oversubscribing the processors, we obtained a 5,000,000,000 FLOPS for the 8 process case.
- It is worth noting that the 2 processor case is slightly worse than the 1 processor case. This is because one worker is actually doing all the work.
- Finally, it is also worth mentioning that the way we split the initial arrays into chunks might not be exactly accurate. Since we simply divide the array size by the number of available processors, in the case this number is not exact, we lose a few calculations. However, we believe that in the long run this does not affect our final results to the point where it is worth changing.
- In order to run the program for *x* processors, we created a bash script that calls `mpirun -np x -hostfile hosts flops`
- In order to debug the program, we print in the output certain messages using a defined function called *debug_print*. This function is only called by compiling in *debugging mode*, which can be done by: `mpicc flops.c -o flops -D DEBUG`