Parallel and distributed computing
Lab 2 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

Part 1:

We started by creating three files: api.c, api.h, and old_sample.c. api.h was the header file for api.c. old_sample.c contains the main file.

api.h : The header file contains the general struct declaration for work and result. It also contains the headers for the create function, compile function, and compute function. The create function creates work. The compile function compiles the results together to form a final result. The compute function finds the result of the inputted work. The work size and result size are also declared. Additionally, the MW_Run and testing functions are declared.

api.c : We define the MW_Run function. The MPI_Comm_size command returns the number of processes in the communication group. The MPI_Comm_rank command returns the rank of the current process in the communication group. If it is the master process, it obtains the entire work pool and distributes the work in round robin fashion to the rest of the processes. The last item in the work pool is null, so the master knows that there is no more work to send.

The master uses MPI_Send to send work to each worker. Upon receiving the work, the slave uses the user defined compute function to obtain the result, which is then sent back to the master process. Upon receiving the all the results from all the work that was sent, the master terminates the workers by sending a message with "TAG_TERMINATE" and then compiles all the results into a final result using the user defined compile function. Since we are doing round robin, the worker needs to keep running after it executes the task and terminates only when the master asks it to.

old_sample.c was our test file. It uses a very simple work structure that just contains a float. The result also just contains a float. The do_work function just copies the values from work to result and the process_result function computes the sum of all values received as result. The user can control the number of work structures by changing the value of size variable.

run.sh is a script file that we used to run these files.

Part 2:

We started by creating three files: mw_api.c, mw_api.h, and sample_q2.c. mw_api.h was the header file for mw_api.c. mw_api.c and mw_api.h are built on top of api.c and api.h. For this we made changes to the API. We added the serialization and deserialization functions for work and results. Prior to sending each chunk of work, master serializes the work, and then uses MPI_Send to each worker.

The primary reason to add serialization was that we did not know the number of factors. So result will contain a variable size array. To be able to pass that structure around we had to serialize it. For the work structure we tried to serialize mpz_t and it didn't work because sizeof(mpz_t) isn't consistent. For this part we just ended up using unsigned long. Though now our work structure doesn't need serialization but we still implement it so that we can continue trying serializing mpz_t.

Upon receiving the work, each slave process deserializes the work. It then uses the user defined compute function to obtain the result. The result is then serialized and sent back to the master process. Upon

receiving the results, the master process deserializes the results. After all the workers are freed and have sent all the results to the master, it compiles all the results into a final result using the user defined compile function.

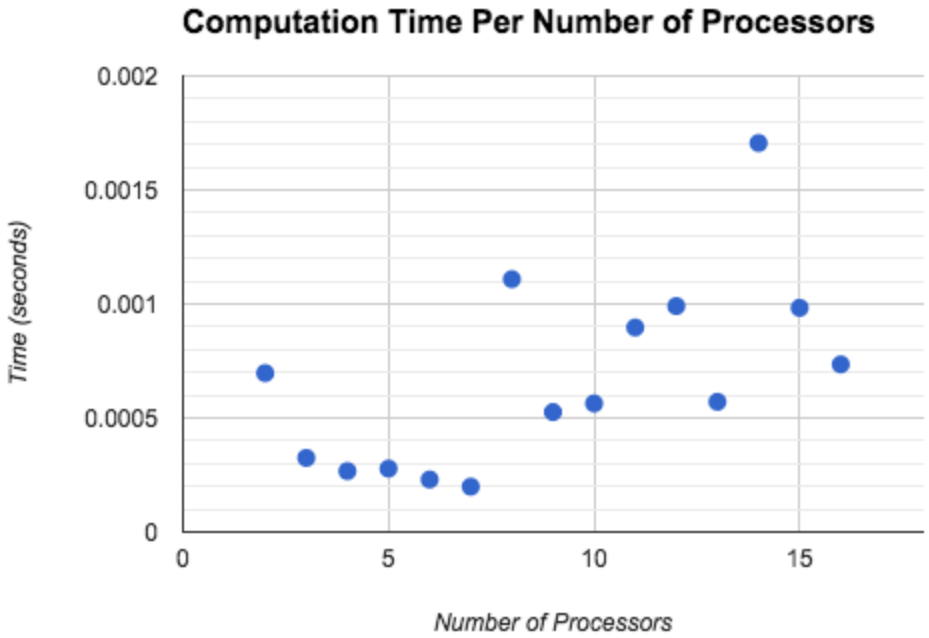
sample_q2.c was our user defined file. It contained the methods required to obtain the list of factors of a large number. We first defined the work and result structures. The work structure included the large number and pointers to an array of the numbers to check. The result structure included the pointer to an array of numbers that were factors.

The serialize_result function converted the result array into a byte stream. The serialize_work function converted the work array into a byte stream. The deserialize_result function converted the result byte stream back into the result array. The deserialize_work result converted the work byte stream back into the work array.

The create_work function converted the large number input from a long to a mpz_t so that the square root function could be applied to the number. The result was then converted back into a long. An array of integers from 1 to the square root of the large input was created.

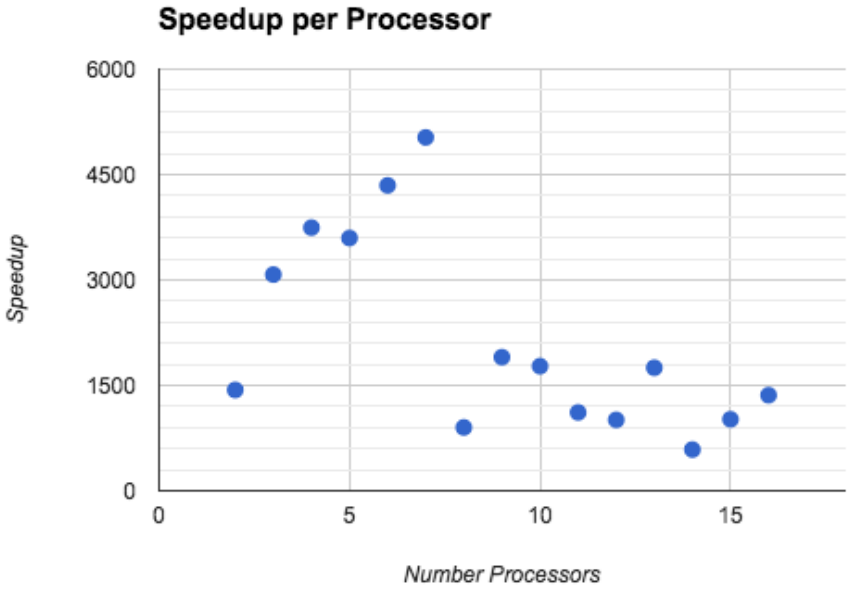
The process_results function compiled the list of result factors into one array. The do_work function checks if each value given in work is a factor of the input large number. It retains the numbers that are factors. The main function instantiates the API specification structure and passes it to the MW_Run function to run it.

fact.sh is a script file that we used to run these files.



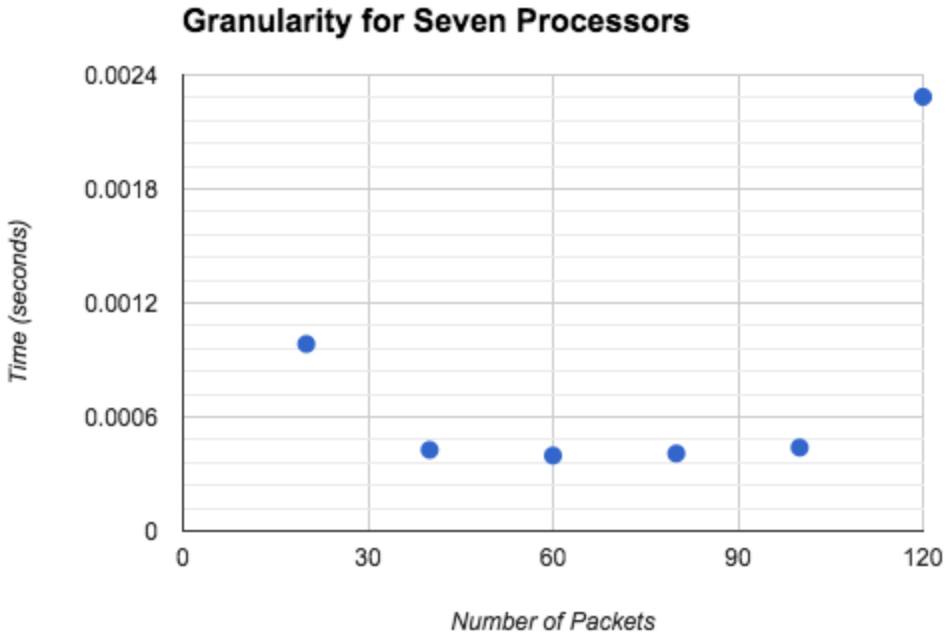
Parallel and distributed computing
Lab 2 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

The speedup is defined as the total work divided by the time on processors. We are assuming that the total work is going to be constant, so for our calculations, we made the total work to be 1.



At eight processors, the speedup is the least. The speedup increases from two to seven processors. After this point, the speedup decreased. Since the AWS instance is split into eight units, it was expected that the speedup would be highest at eight. However, there is quite a lot of overhead from the sending and receiving functions between the processes. Also probably it's because we are not using mpz_t and that doesn't generate enough work.

Parallel and distributed computing
Lab 2 Report
Yukti Abrol(ya246), Rohit Jain(rj288)



Granularity is the ratio of computation to communication. It appears as though larger packet sizes takes a longer time to transfer. As we increase the number of packets, the computation remains same but the communication is much more because we send only packet of work to each processor. So the communication overhead increases a lot as we keep increasing the number of packets.

For the pre-assigning work case, we made the number of packets equal to the number of processors. To check the dynamic allocation, we increase the number of packets such that each worker will request for work once it is finished with its present work. The benefits of pre-assigning work to each worker includes less work for the master in terms of assigning work, and less communication between the master and the workers. However, some workers can finish their tasks faster than other workers. This will result in some workers remaining idle. For dynamic allocation, there will be more communication overhead between the master and the workers as the master assigns more work to each worker when it has finished its previous work. The benefit is that the workers will not remain idle.