Parallel Programming Practical

N Body Simulation in MPI

Trisha Anand (2596695)

1. Introduction

As part of this assignment a parallel N-Body algorithm has been implemented using C and the MPI (Message Passing Interface) communication library. The application has been run and benchmarked on the DAS-4 cluster at the VU.

2. Design and Implementation

Sequential Algorithm :

In a given step, the total force on a single body is calculated. This force is sum of all the forces of all the bodies on this particular single body. This is done for all the bodies in the world sequentially. Newton's third law of F(a,b) = -F(b,a) is utilized to cut the computations in half. The force on each body is used to calculate its velocity which in-turn is used to calculate it's new position. This is repeated for each step.

Parallel Algorithm :

In a given step, for each particle, first the force is calculated, then the velocity and then its location (x,y coordinates). For this reason, the work unit to be distributed amongst the processes should be particle (or bodies). Each process should be responsible for computing forces, velocities and locations for the bodies it is responsible for each step.

a. Naive Version

For the first parallel version, the baseline algorithm used in sequential algorithm was changed. Instead of utilizing the newton's third law of equal and opposite reaction, all the forces were calculated. This leads to equal work for all bodies and no communication requirement when it comes to calculating the total force on any given body. The total computations for calculation of forces are doubled. Everything else remains the same. The bodies are distributed to all the particles in a block distribution of a 1D array (of all the bodies).

In the initialization phase, MPI Broadcast is used to distribute the mass and radius of all bodies to all the processes. These two arrays are immutable during the entire execution and need not be communicated during execution. Also, initial locations and initial velocities are also broadcasted to all the processes. All the processes maintain the (x,y) positions of all the bodies in the world throughout the execution.

For each step, first the forces are computed for bodies belonging to the particular process. All the variables required for the computations are available on the local machine at this stage. No communications or dependencies exist. Once the forces have been

calculated, the velocities and new positions for the bodies are computed. This is done on all the processes leading to new forces, velocities and positions computed for all the bodies in the world. At the end of the step, the new locations should be updated across all the processes so that in the computation of forces in the next step, the processes have access to the x and y location of all the bodies. To achieve this **MPI_Allgatherv** collective is used which gathers from all the processes and distributes to all the processes. For each process, there is a **count_per_process[process_id]** which stores the number of bodies that the process is responsible for. The index "'low" corresponds to the first index belonging to the process. The MPI collective call used is as below:

```
MPI_Allgatherv(&world->bodies.x[low], count_per_process[process_id], MPI_DOUBLE,
world->bodies.x, count_per_process, displs_array, MPI_DOUBLE, MPI_COMM_WORLD);

MPI_Allgatherv(&world->bodies.y[low], count_per_process[process_id], MPI_DOUBLE,
world->bodies.y, count_per_process, displs_array, MPI_DOUBLE, MPI_COMM_WORLD);
```

The naive version doubles the computations and in single process version takes double the amount of time in comparison to the sequential version provided in the assignment. This is expected.

b. Optimized Version

i. Algorithm

For the optimized version, the first order of business is to revert back to the same algorithm as the sequential version to get an immediate 2x speedup from the naive parallel version covered in the last section by decreasing the computations for force by half. This requires communication between processes instead of computations. During the force calculations, whenever force is computed, the opposite force has to be saved and communicated to the process that owns the other body.

Load Balancing : In compute force, force is only calculated for a given body if its index is lower than the other body. Otherwise the force is derived from already calculated force using Newton's third law. This property of the algorithm makes the work vary depending on each body (Body 0 computes all the forces because all the other indices are greater than 0, and Body 'N-1' computes no forces because all the other indices are smaller than N-1 and hence the forces are derived.) So, this algorithm required cyclic distribution of bodies instead of block partitioning for load balancing on all the processes.

Cyclic Distribution is implemented by storing indices that each processor would be responsible for as below. 'particle_distribution[process_id]' is an array that stores all the indices that the process - 'rem' - would be responsible for.

```
for (i = 0; i < world->bodyCt; i++)
   {
       int rem = i % num_processes;
       int div = i / num_processes;
```

```
        particle_distribution[rem][div] = i;
    }
```

In the initializing phase, the root process (process id : 0) broadcasts the arrays that are required for all the processes. This includes the mass and the radius arrays. Mass and Radius arrays are static and don't change during the execution of the simulation. We utilize this fact by storing these two arrays in all the processors who require these values for force calculations.

In compute force, each process has access only to the bodies that belong to it. Instead of all processes passing their own bodies to the others for computation, ring communication is used as shown in Figure 1. In this, each process sends data to its right neighbour and receives data from its left neighbour. (Reference 1)

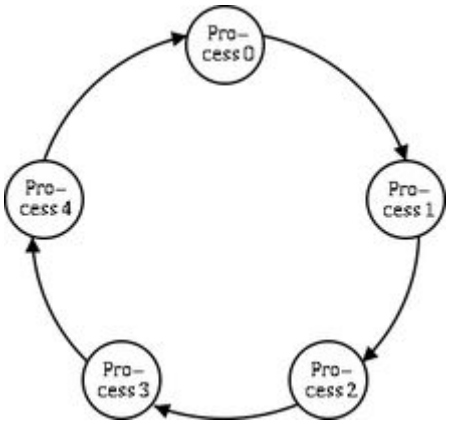

Figure 1 : Ring Communication

First step in compute forces is to create this temporary communication array which shall store all the forces for the particles owned by the process which are derived using the newton's third law. The forces that are computed (and not derived) are only computed by the process that owns the bodies. These computed forces are added directly to the force for the body. The derived ones can come from either the same process or from other processes. These derived forces are stored in the temporary communication array.

The forces are calculated as shown in the steps below :

```
Loop which runs 'number of processes' times :
{
    1. Calculate forces between the particles owned by the process and the
       particles in temporary communication array.
    2. Computed force is added to the force of the particle owned by the process.
       The derived forces are added to the forces in the temporary communication
       array.
    3. Send the temporary communication array to the right side neighbour.
    4. Receive the temporary communication array from the left side neighbour.
}
```

As the loop runs ‘number of processes’ times, the temporary communication array from each process is passed ‘number of processes’ times. This ensures that the force between all the particles is computed for all the processes.

At this point the temporary communication array contains the same bodies as that of the process. The derived forces saved in the temporary communication array are now added to the forces saved on the process. At this point all the forces have been calculated and we can move on to the next steps of computing velocities and positions.

ii. Implementation

The temporary communication array contains the indexes of the particles and the forces for these particles as well as the positions of these particles as calculated in the previous step. This array contains these multiple arrays so that when communicating only one array is sent and received instead of using multiple MPI communication for each of these arrays. This temporary communication array is first created at the beginning of compute forces step. It is filled up with indices, and positions. The space is allocated for forces and made zero.

The first implementation uses MPI_Sendrecv_replace for the temporary communication array. This requires no allocation of another buffer to receive the temporary communication array from the left side neighbour. It is used in the following manner :

```
MPI_Sendrecv_replace(tmp_comm_array, array_size, MPI_DOUBLE, neighbour_send, 0,
neighbour_recv, 0, MPI_COMM_WORLD, &status);
```

The rest of the computations (velocities, positions) are the same as sequential version. The particle distribution array stores the bodies that each process is responsible for. Using this array as a reference, the velocities and new positions are calculated for the bodies that the process is responsible for. These calculations are independent and do not require any special effort to parallelize.

iii. Optimization 1

MPI_Sendrecv_replace is a blocking method of sending and receiving data. The first optimization applied to reduce the time spent in compute forces function is to make this a non blocking call. To achieve this, MPI_Isend and MPI_Irecv are used instead. A new buffer is allocated to receive temporary communication array from left neighbour so that send and receive can happen independently of each other. A barrier is introduced after these two statements so that once the send and the receive is completed, the temporary communication array can be rewritten using the buffer from the receive.

iv. Optimization 2

The next optimization applied was to use persistent communication. The assumption is that since the same arguments are used for MPI send and receive throughout the computation of forces across all the steps, one can eliminate the cost of the overhead for communication between the process and communication controller.

Persistent send and receive are initialized in the following way :

```
MPI_Recv_init(receive_buffer, arr_size, MPI_DOUBLE, neighbour_recv, 0,
MPI_COMM_WORLD, &request[1]);

MPI_Send_init(tmp_comm_array, arr_size, MPI_DOUBLE, neighbour_send, 0,
MPI_COMM_WORLD, &request[0]);
```

In the loop for communicating the temporary communication array and computing forces, the sends and receives can be started repeatedly by calling MPI_Start function.

3. Experimentation and Measurements

The following experiments have been run on DAS4 with varying number of machines. The number of cores used on each machine though is limited to 1. The raw data used to plot the figures are available in Appendix in Section 7.

Each of these experiments measure execution time of *do_compute()* function on each node. The initial distribution of data and final gathering of final data is not included in the timing. After *do_compute()* finishes running on all nodes, the time elapsed is calculated on all nodes. Maximum time elapsed is found across all the nodes and that is treated as the total executing running time.

Each experiment is run from 1 node to 16 nodes and run time is recorded for all the number of nodes on which the experiment is run.

a. Experiment 1 : 32 bodies / 100,000 steps

The three versions (Implementation, Optimization 1 and Optimization 2) were run with varying number of nodes for 32 total bodies over 100,000 steps.

Maximum speedup observed with the final version = 8.559 seconds (sequential run) / 4.353 seconds (16 nodes) = 1.966

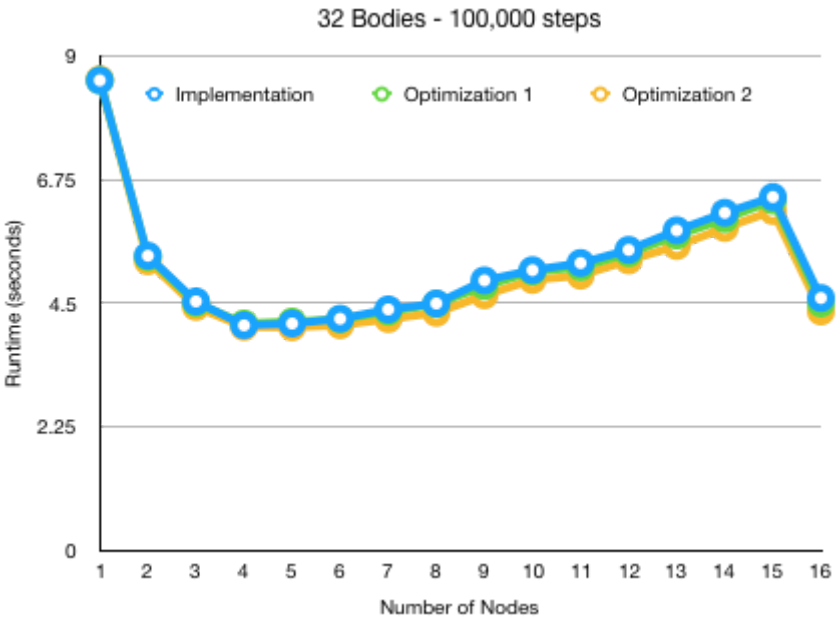

Figure 2 : 32 Bodies/100,000 steps

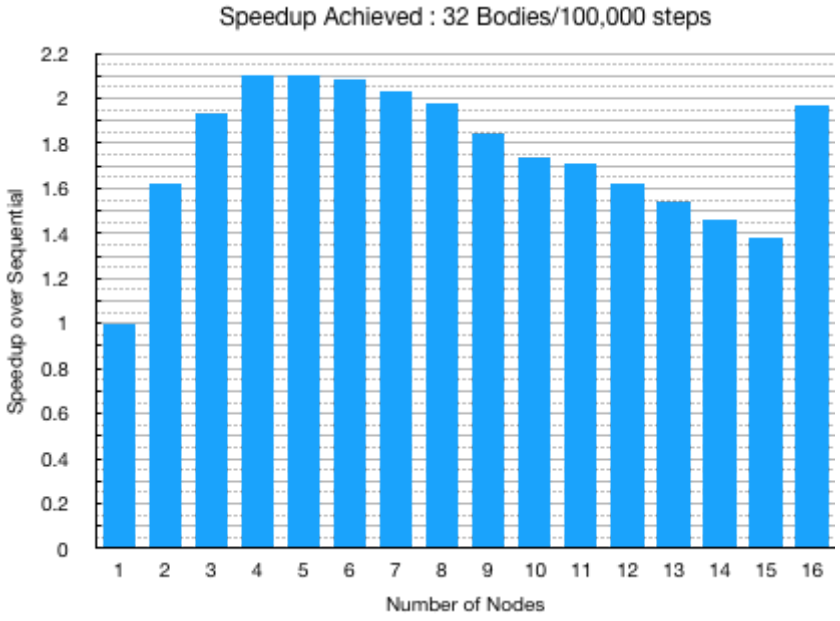

Figure 3 : Speedup over sequential nbody-par : 32 bodies/100,000 steps

As can be seen the run time decreases with increasing number of nodes till 5 nodes and then steadily increases with increasing number of nodes. This is because after 5 nodes the cost of communication is more than the benefit of parallelization as the size of the problem is too small. The sharp decline in run time for 16 nodes is theorized to be because of equal distribution of the problem leading to possible runtime across any node to be similar leading to very low run time overall.

In this experiment, benefits of optimization 2 (persistent communication) is visible. This is the only experiment where this optimization shines through. This is because the computation load of this problem is so low that the communication latencies are visible. This experiment shows that persistent communication offers slightly better performance in

repeated communication using the same arguments by pre-constructing the communication handler.

b. Experiment 2 : 256 bodies / 100,000 steps

The three versions (Implementation, Optimization 1 and Optimization 2) were run with varying number of nodes for 32 total bodies over 100,000 steps.

Maximum speedup observed with the final version = 564.191seconds (sequential run) / 43.871 seconds (16 nodes) = 12.86

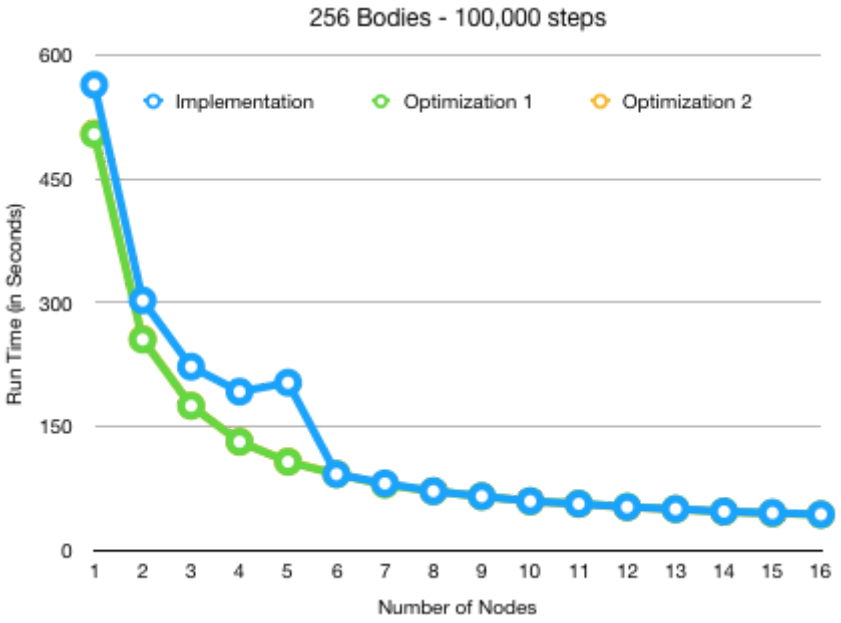

Figure 4 : 256 Bodies/ 100,000 steps

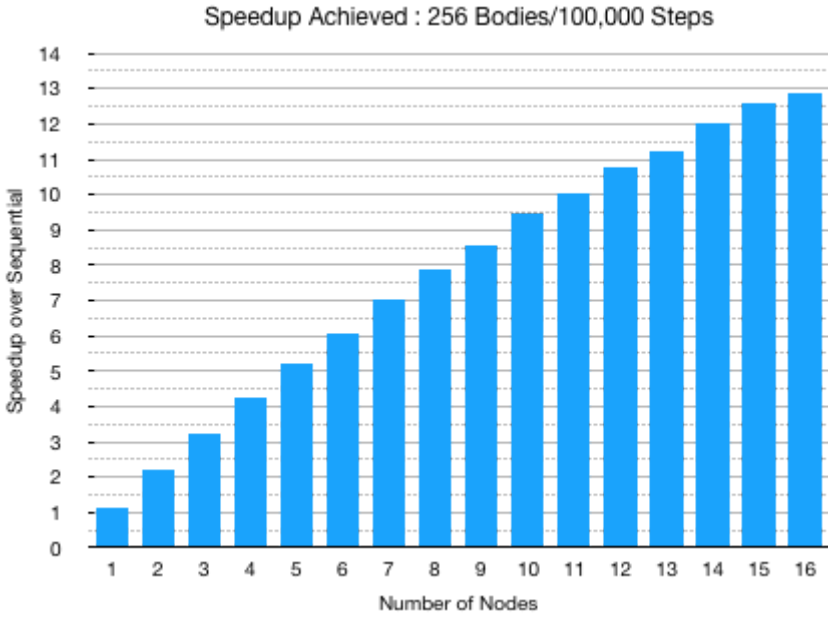

Figure 5: Speedup over Sequential nbody-par : 256 Bodies/100,000 steps

In this version, Optimization 2 and Optimization 3 have no difference in run time performance. In this version, the original implementation suffered from high run time for a few nodes as seen in Figure 4. This is hypothesized to be because of the use of *MPI_Sendrecv_replace()* function for communication which has a blocking property. This variance in performance during run time should be expected from blocking communication paradigms. Using isend, irecv and persistent communication, which in turn is non blocking, leads to the removal of random spikes in run time seen in experiment data whilst using *MPI_Sendrecv_replace().*

c. Experiment 3 : 1000 bodies / 10,000 steps

The three versions (Implementation, Optimization 1 and Optimization 2) were run with varying number of nodes for 32 total bodies over 100,000 steps.

Maximum speedup observed with the final version = 765.003 seconds (sequential run) / 51.072 seconds (16 nodes) = 14.98

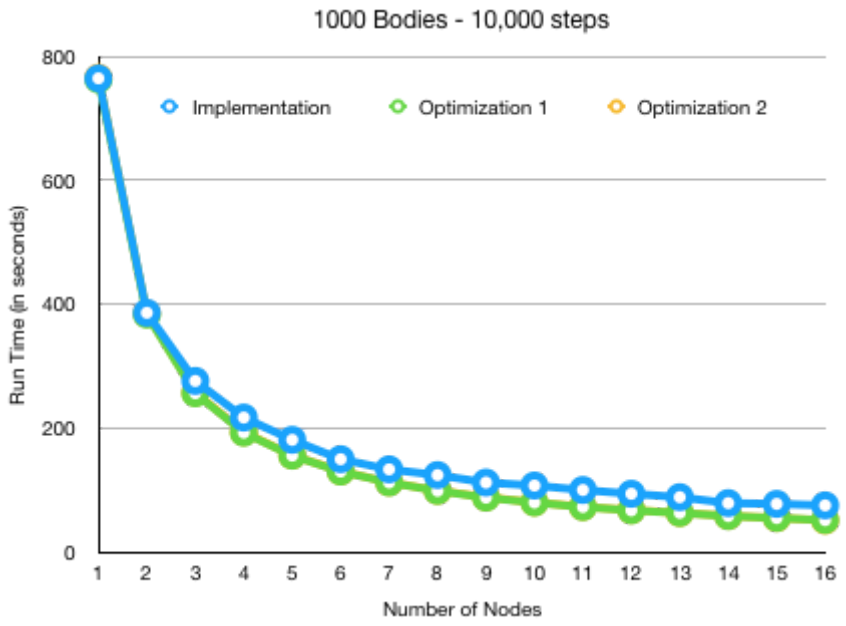

Figure 6 : 1000 Bodies/10,000 steps

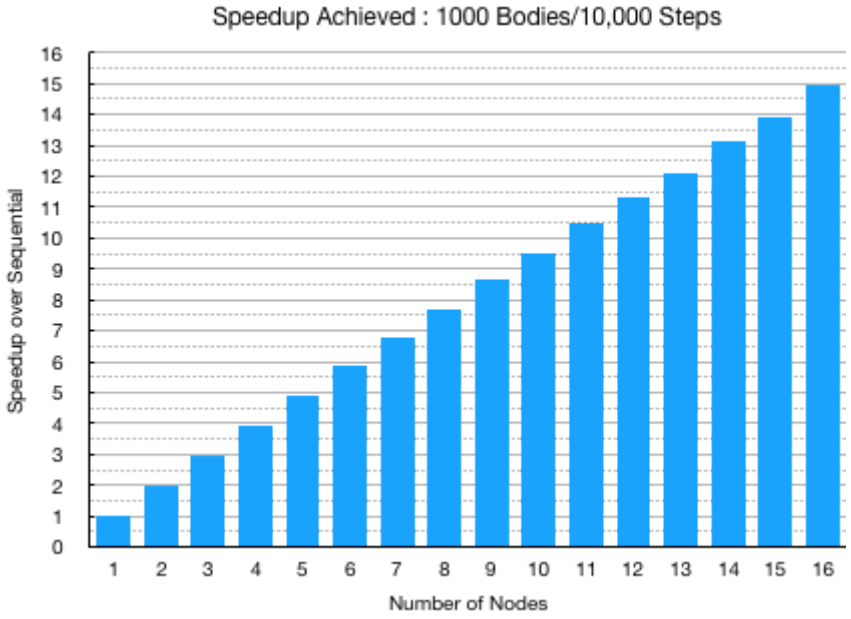

Figure 7 : Speedup over Sequential : nbody-par : 1000 Bodies/10,000 steps

As the size of problem is increased we are getting closer to the maximum possible speedup. Once again, same as Experiment 2, no surmisable difference exists between the performance of Optimization 1 and Optimization 2.
In the parallelization, even though there is no optimization to hide the latency of communication (like overlapping of computation and communication), the computations are large enough that communication latencies can be almost ignored for large problem statements like in this experiment of 1000 bodies.

4. Conclusion

As part of this assignment, n-body problem was parallelized using MPI. Using cyclic distribution of bodies across the MPI processes, and ring communication between these processes, speedup of 15 was achieved over 16 machines. Three different communication paradigms were used : Blocking, Non blocking, and Persistent.

5. References

1. https://www.srividyaengg.ac.in/coursematerial/CSE/104815.pdf

6. Appendix

In this section, the raw data is presented using which the figures (charts) in Section 3 were created.

1. Table 1 : 32 Bodies/100,000 steps

In the table 1 below, the runtime in seconds is recorded with different number of nodes used to run the problem 32 Bodies over 100,000 steps.

Number of Nodes	Original Implementation (seconds)	Optimization 1 (seconds)	Optimization 2 (seconds)
1	8.559	8.562	8.576
2	5.367	5.341	5.267
3	4.536	4.505	4.436
4	4.100	4.134	4.071
5	4.133	4.170	4.065
6	4.220	4.210	4.102
7	4.390	4.347	4.214
8	4.498	4.488	4.328
9	4.918	4.815	4.648
10	5.105	5.077	4.927
11	5.234	5.156	5.005
12	5.474	5.412	5.284
13	5.829	5.735	5.545
14	6.149	6.037	5.873
15	6.436	6.373	6.182
16	4.598	4.497	4.353

2. Table 2 : 256 Bodies/100,000 steps

In the table 2 below, the runtime in seconds is recorded with different number of nodes used to run the problem 256 Bodies over 100,000 steps.

Number of Nodes	Original Implementation (seconds)	Optimization 1 (seconds)	Optimization 2 (seconds)
1	564.191	504.317	505.385
2	302.900	255.568	256.376
3	222.674	175.273	175.523
4	192.416	131.780	131.904

5	203.327	107.511	107.698
6	92.492	92.423	92.830
7	81.282	79.907	79.933
8	71.896	71.825	71.700
9	65.692	65.652	65.680
10	59.792	59.773	59.404
11	56.572	56.692	56.373
12	52.657	53.024	52.244
13	50.288	50.262	50.147
14	47.241	47.100	46.836
15	45.544	45.323	44.887
16	43.777	43.493	43.871

3. Table 3 : 1000 Bodies/10,000 steps

In the table 3 below, the runtime in seconds is recorded with different number of nodes used to run the problem 1000 Bodies over 10,000 steps.

Number of Nodes	Original Implementation (seconds)	Optimization 1 (seconds)	Optimization 2 (seconds)
1	765.003	763.279	765.974
2	386.297	383.446	384.832
3	276.242	256.318	257.505
4	217.125	192.796	193.583
5	181.147	155.601	156.015
6	149.834	129.924	130.399
7	133.433	112.244	112.579
8	124.201	98.780	99.086
9	112.143	87.727	88.469
10	107.284	79.849	80.198
11	100.122	72.669	72.938
12	94.017	67.230	67.620
13	88.502	63.030	63.229

14	78.899	57.923	58.228
15	77.673	54.796	54.867
16	75.250	51.381	51.072