Parallel Computing for Simulation Sciences: Project 3

Jaeyong Jung (359804)

July 12, 2016

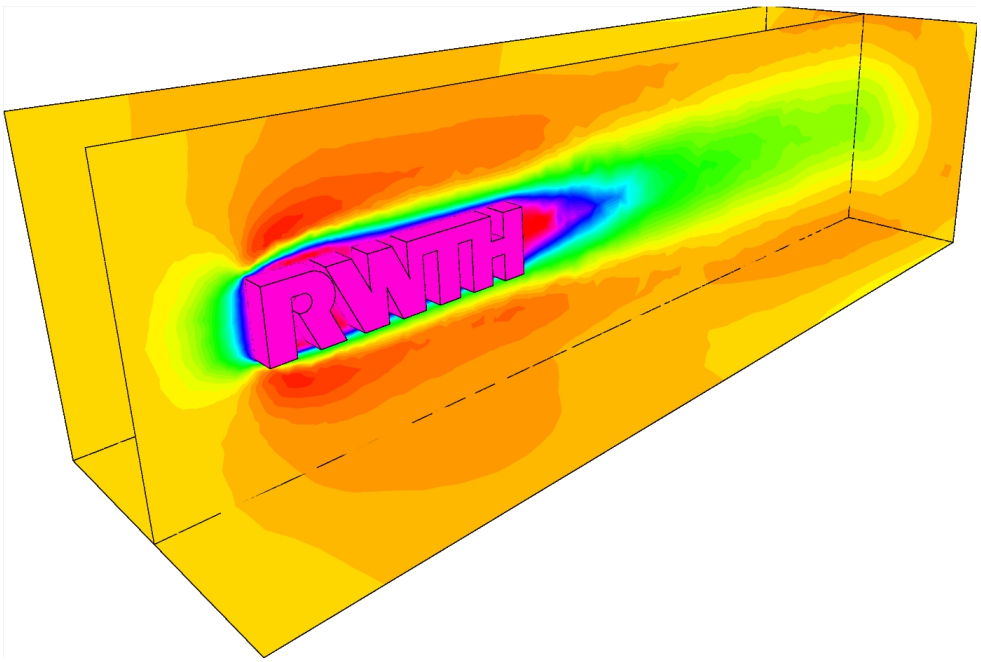

Figure 1: Figure made by 32 processors

1. For the round-robin algorithm use the MPI_Isend version, such that communication and computation can be performed to a certain extent concurrently.

The communication algorithm is conceived to allow communication to be done during process of serial computation. It is compulsory to have MPI_Wait commands because sendbuffer should not be modified before MPI_Isend is finished. Consequentially, the location of MPI_Wait affects the concurrency of communication and computation. To realize the concurrency, MPI_Wait can be placed just before "address_swaper". If this is set before serial computation, it will block computation until MPI_Isend is completely done. Therefore, the concurrency can be realized by allowing MPI_Wait to be just before sendbuffer's modification code as close as possible.

```
// Wait unitil previous Isend is finished so that addresses can be
    swaped.
if ((ipes+rounds) > 0)
{
        MPI_Wait(&request[0], MPI_STATUS_IGNORE);
        MPI_Wait(&request[1], MPI_STATUS_IGNORE);
        MPI_Wait(&request[2], MPI_STATUS_IGNORE);
}

// Isend communication
// npes -1 + round(=1) = npes is the last step
if ((ipes + rounds) < npes) {
        // wait until previous sending (mxyz, data, node_found) is
            finished
        address_swaper(&mxyz_fine_send, &mxyz_fine);
        address_swaper(&data_fine_send, &data_fine);
        address_swaper(&node_found_send, &node_found);

        MPI_Isend(&(mxyz_fine_send[0][0]), nnc_fine*nsd,
           MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &request[0]);

        // calculated data stored in data_fine. Send what each
           processor has now

        MPI_Isend(&(data_fine_send[0][0]), nnc_fine*ndf,
           MPI_DOUBLE, next, 1, MPI_COMM_WORLD, &request[1]);

        // calculated data stored in data_fine. Send what each
           processor has now


        MPI_Isend(&(node_found_send[0]), nnc_fine, MPI_INT, next,
           2, MPI_COMM_WORLD, &request[2]);
```

```
        // wait until previous sending (nnc_fine, offset_nn) is
           finished
        if ((ipes+rounds) > 0)
        {
                MPI_Wait(&request[3], MPI_STATUS_IGNORE);
                MPI_Wait(&request[4], MPI_STATUS_IGNORE);
        }
        nnc_fine_send = nnc_fine;
        offset_nn_fine_send = offset_nn_fine;
        MPI_Isend(&(nnc_fine_send), 1, MPI_INT, next, 3,
           MPI_COMM_WORLD, &request[3]);
        MPI_Isend(&(offset_nn_fine_send), 1, MPI_INT, next, 4,
           MPI_COMM_WORLD, &request[4]);
}
```

2. For distributing the fine mesh nodes and coarse mesh elements, conceive an algorithm that splits them as evenly as possible across the different processing units.

Yes, they will behave differently if size of buffer is too much big. MPI_Send and MPI_Rsend do not work with large buffer size. That is because temporary buffer in MPI_Send communication has limited size. If send buffer's size exceeds a certain amount of memory, temporary buffer cannot save send buffer's data. Thus, MPI_Send and MPI_Rsend should be used with small amount of send buffer. On computer cluster, it is measured that MPI_Send and MPI_Rsend will work with buffer size of 505 at most.

On the other side, MPI_Isend works with large buffer size. The reason is that MPI_Isend simply sends buffer's data when there is a matching MPI_Recv. In conclusion, it is recommendable to implement MPI communication with MPI_Isend on Round-Robin algorithm.

3. Comment on possible differences between the OpenMP- and MPI-parallelized codes with respect to consistency and uniqueness of the generated results. Is the generated file data.fine the same irrespective of the number of cores used for execution? We do not require your code to be consistent in that regard.

MPI-parallelized codes will not have uniqueness of generated results, depending on number of processors. If the number of processors is changed, it will yield different results. when there is a fine node that does not get into any coarse mesh with 0 tolerance, it will be interpolated with different coarse elements. That is because that kind of node can be interpolated with arbitrary element which allows 0.2 tolerance. Changing the number of processors will yield a change in offsets. Consequentially, processors will have different set of coarse elements, and try to do interpolation based on what they have. This nature of MPI-parallelization engenders randomness of result. However, difference between results will be not so big since tolerance is small.

On the contrary, the result of OpenMP is independent of the number of threads. The reason is that every thread has shared memory so that will try to have a look on the same coarse elements. To verify this in this project. Output file "data.fine"'s values are compared using MATLAB.

omp 8	omp 6	MPI 4 proc	MPI 8 proc	MPI 32 proc
8.75e-4	8.75e-4	9.38e-6	8.84e-4	9.39e-4

Table 1: Relative norm of error with respect to norm of data, made using 1 processor

Table 1 shows relative error of results. Reference is result made by one processor. 2-norms of errors are calculated using MATLAB. After that, 2-norms of errors were divided by the norm of the reference result. Regardless of using different number of threads to run OpenMP, the value of relative error is $8.75e-4$. Still, relative error of MPI depends on the number of processors. Therefore, the result of MPI depends on the number of processors. Nevertheless, the difference is negligible since tolerance is small. Also, the result of computation using MPI will be consistent if the number of processors is constant. That is because the algorithm to split computational load is only dependent on number of processors.

4. At the end of our round-robin process, if sending fine mesh data, the data may not reside on the same PE as it started on; this must be accounted by proper offset before writing the data.

In general, each processors will have different set of nodes other than it used to have. Therefore, it is important to keep track on offset of each set of nodes. By tracking the offset, each processor will know where their final data will fit in and write them on proper point. Below code includes MPI communication to pass offset value to next processor and receiving one from previous processor.

```
// To make it sure that sending nnc, offset are complete.
if (ipes + rounds >0)
{
        // previous step's sending nnc_fine complete
        MPI_Wait(&request[3], MPI_STATUS_IGNORE);
        nnc_fine_send = nnc_fine;
        MPI_Isend(&(nnc_fine_send), 1, MPI_INT, next, 3,
           MPI_COMM_WORLD, &request[3]);

        // sending offset_nn_fine complete
        MPI_Wait(&request[4], MPI_STATUS_IGNORE);
        offset_nn_fine_send = offset_nn_fine;
        MPI_Isend(&(offset_nn_fine_send), 1, MPI_INT, next, 4,
           MPI_COMM_WORLD, &request[4]);
}
else
```

```
{
        nnc_fine_send = nnc_fine;
        offset_nn_fine_send = offset_nn_fine;
        MPI_Isend(&(nnc_fine_send), 1, MPI_INT, next, 3,
          MPI_COMM_WORLD, &request[3]);
        MPI_Isend(&(offset_nn_fine_send), 1, MPI_INT, next, 4,
          MPI_COMM_WORLD, &request[4]);
}
```

5. Tracking analysis of MPI communication

It is necessary to check how processors have communicated by visualizing it. Red bar means time interval for MPI commands. In figure 2, it is seen that processors do communication with neighbour processors. In the first time interval, from 2.5 to 7.5 sec, several processors finished communication very early and the others took more time. It means that computational load, dependent on structure of mesh, is biased. If a processor has more computational load, it will finish serial computation later. It lets next processor to wait until the slow processor finishes. Also, it is seen that long red MPI_Recv bar is propagating from 0-th processor to 11-th processor. It shows that computational bias can be amplified as communication goes on.

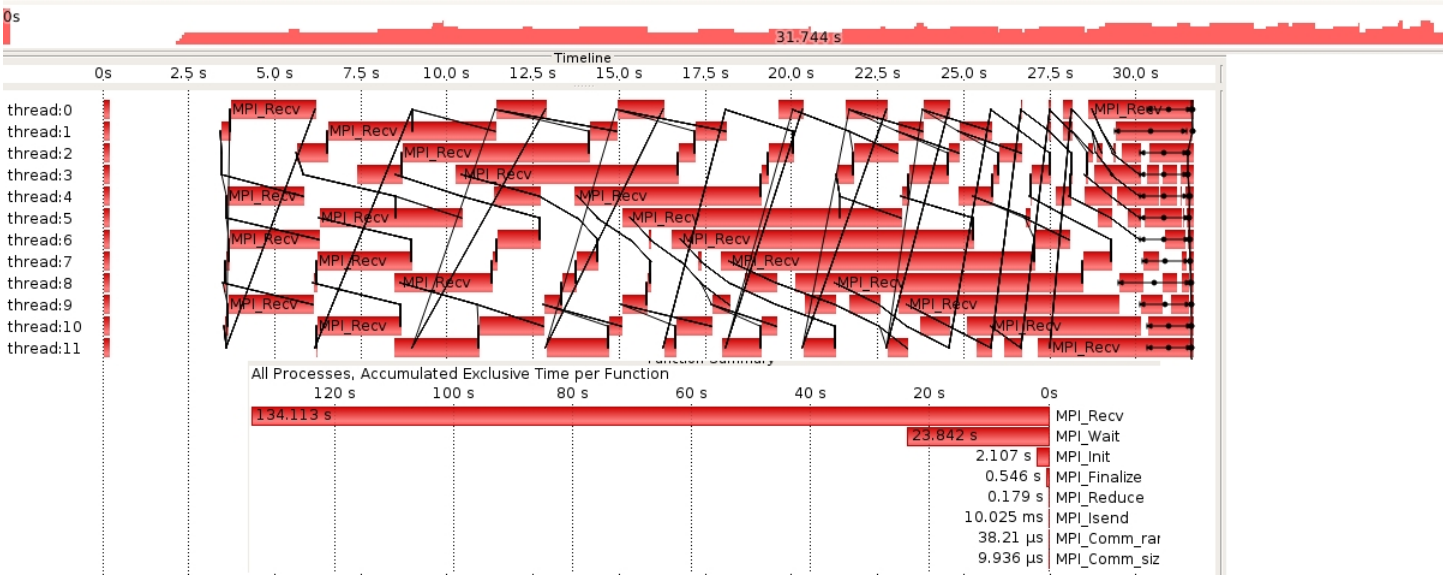

Figure 2: Overall communication graph

In summary, this tracking analysis insists that evenly distributed computational load is crucial to shorten total runtime.

6. Considerations about scalability

In order to study how much MPI makes the program faster, runtime is measured. For

each number of processors, runtime is measured three times and average value is calculated. Runtime, scalability, and parallel efficiency is visualized below.

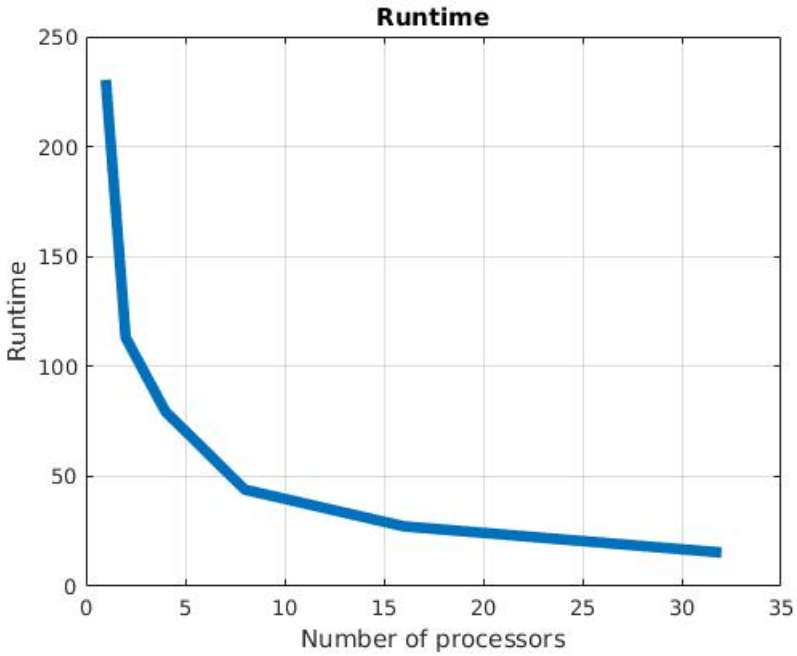

Figure 3: MPI Runtime

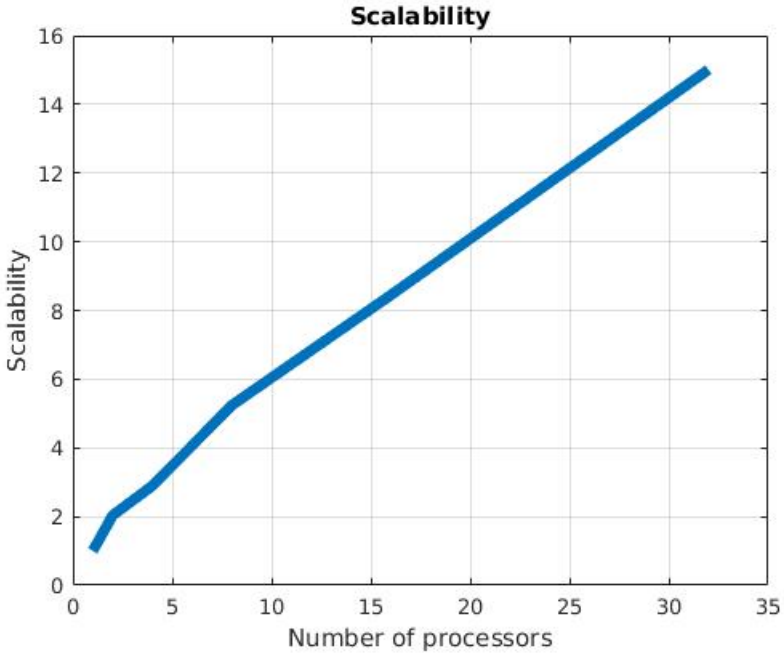

Figure 4: MPI Scalability

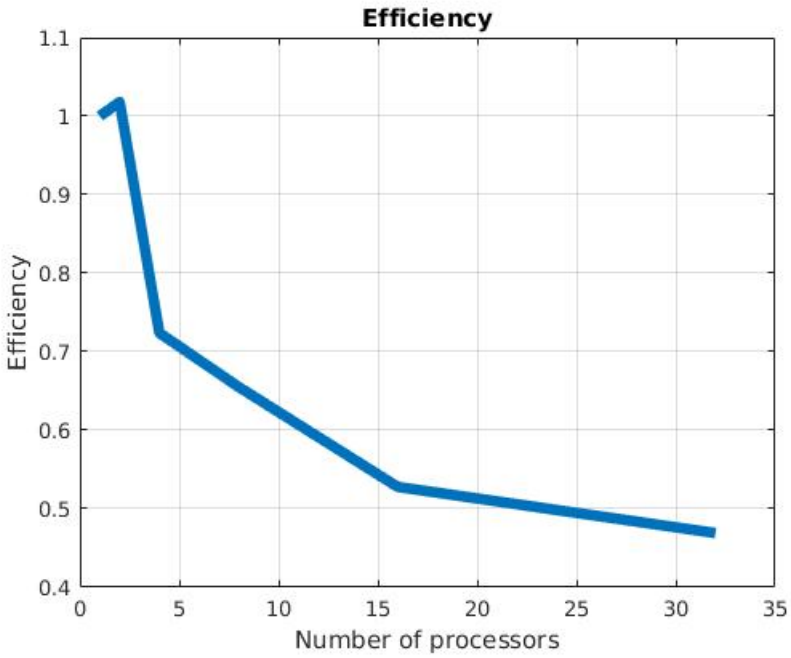

Figure 5: MPI Efficiency

The scalability is linearly increasing with respect to number of processors, according to Figure 4. At the same time, parallel Efficiency decreases inverse proportionally. It coincides with Figure 2 that larger number of MPI communication will waste time on MPI_Recv. It can be inferred from Figure 2 that as communication step goes on, it is more likely to waste time on MPI_Recv. MPI communication with more processor is accompanied by more communication steps, and it leads to bad parallel efficiency.

7. Another algorithm to improve scalability

As it is seen from previous Figures, the program is made to allow the concurrency but wastes much time on MPI_Recv. I changed the position of MPI_Wait to be right after MPI_Recv. Runtime is measured in the same way as it has been done. The alternative algorithm's MPI_Wait looks like below code.

```
// Round-robin loop
for (int ipes=0; ipes <= npes-1; ipes++) {

    // Irecv communication to get new data
    if ((ipes + rounds) > 0) {

        MPI_Recv(&(nnc_fine), 1, MPI_INT, prev, 3,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        MPI_Recv(&(mxyz_fine[0][0]), nnc_fine*nsd,
          MPI_DOUBLE, prev, 0, MPI_COMM_WORLD,
```

```
                MPI_STATUS_IGNORE);
            MPI_Recv(&(data_fine[0][0]), nnc_fine*ndf,
                MPI_DOUBLE, prev, 1, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
            MPI_Recv(&(node_found[0]), nnc_fine, MPI_INT, prev
                , 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(&(offset_nn_fine), 1, MPI_INT, prev, 4,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // wait until sending (mxyz, data, node_found) is finished
            MPI_Wait(&request[0], MPI_STATUS_IGNORE);
            MPI_Wait(&request[1], MPI_STATUS_IGNORE);
            MPI_Wait(&request[2], MPI_STATUS_IGNORE);
            printf("[%d] MPI_Recv done, message size %d \n",
                mype, nnc_fine);
        }
```

This code's runtime and pattern of MPI communication prove that it is more fast and efficient compared to the previous algorithm.

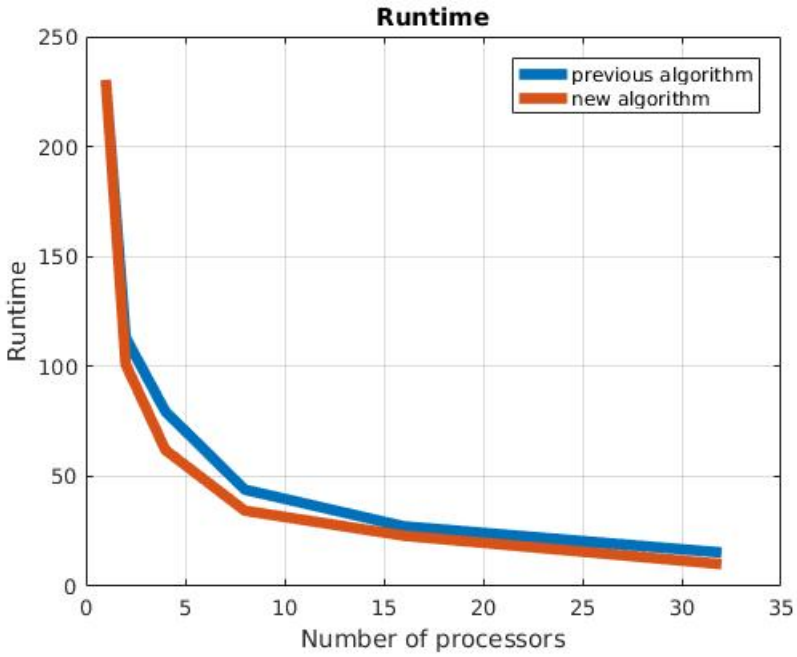

Figure 6: MPI Runtime Comparison

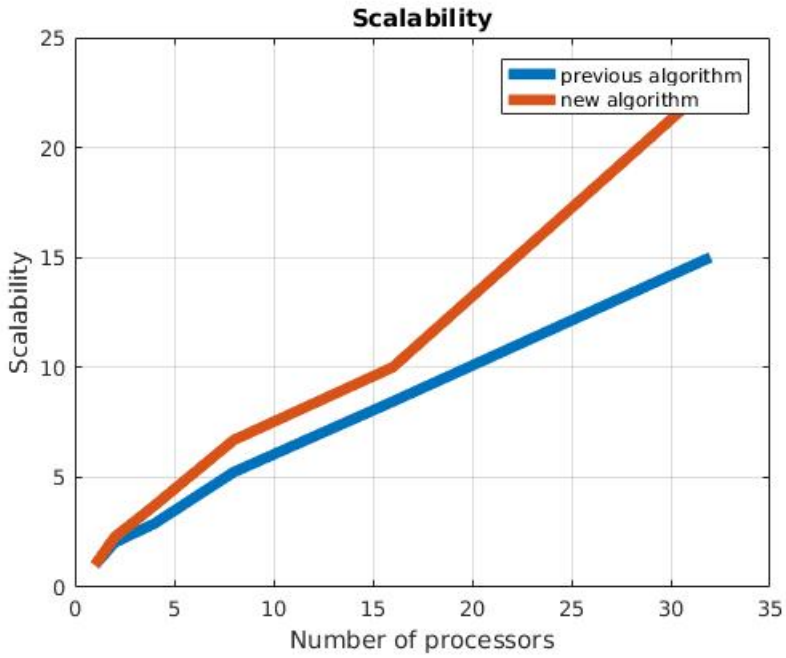

Figure 7: MPI Scalability Comparison

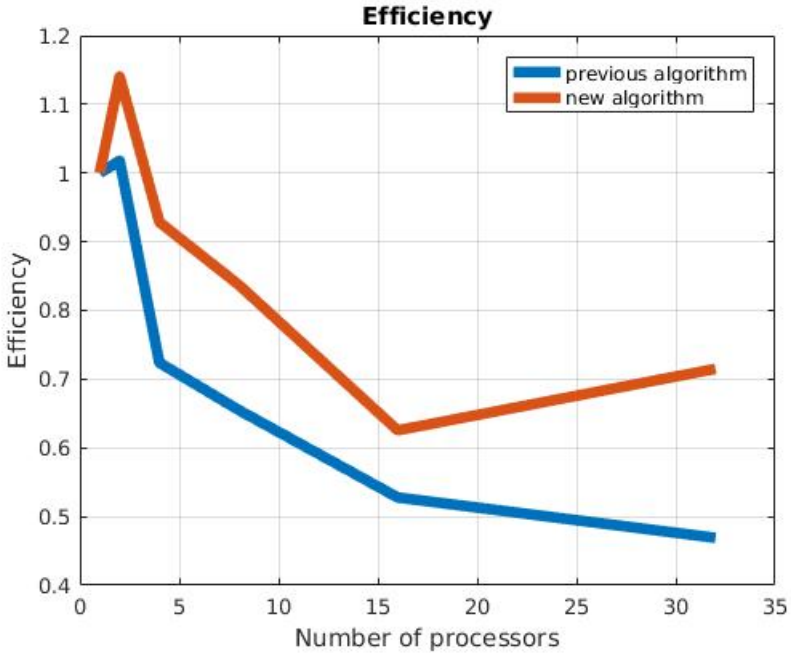

Figure 8: MPI Efficiency Comparison

This result is unexpected at the first time since the first algorithm looks faster. The computational is undertaken to compare two MPI communications. Figure 9 represents the alternative MPI communication.

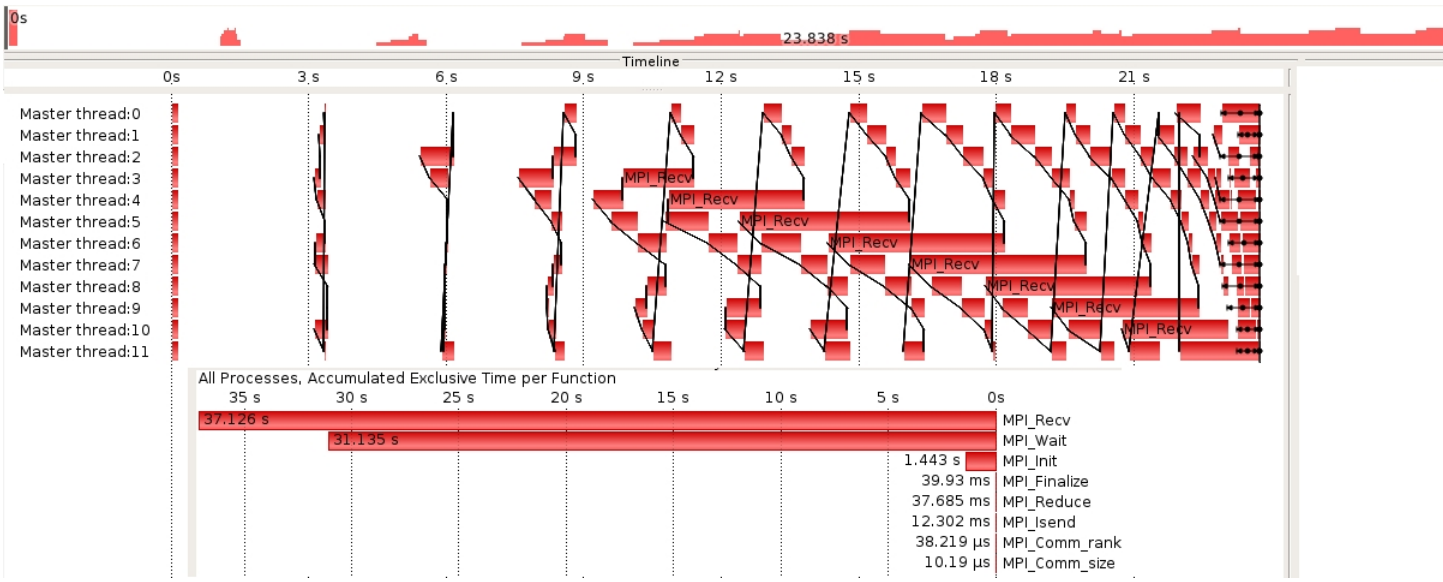

Figure 9: Overall communication graph

By comparing Figure 9 with Figure 2, it is clear that the alternative algorithm consumes less time for MPI communication. In Figure 2, MPI_Recv takes 134 seconds but the alternative one takes only 37 seconds. Nevertheless, the alternative needs about 8 seconds more to undertake MPI_Wait. This drawback is negligible since the alternative algorithm saves 97 seconds for MPI_Recv. Visually, Figure 9 looks that it has more structured and short communication pattern compared to Figure 2. Although, the alternative algorithm does not allow the concurrency between serial computation and MPI communication, it is much faster.

It is evident that there should be another factor that affects MPI communication performance other than the concurrency. Depending on structure of serial computational program or interactions of MPI communication, sometimes, trying another method that opposites to intuitive idea will be better.