Report High-Perfomance-Computing - 2

U2291111

March 2023

1 Initial Profiling Result

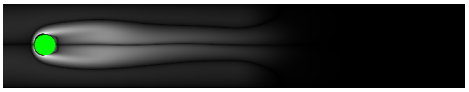


Figure 1: Original Karman

Function	Self runtime (s)	Cumulative runtime (s)	Total running time (%)
poisson	25.50	25.50	95.28
compute_tentative_velocity	1.00	26.50	3.74
compute_rhs	0.12	26.62	0.45
update_velocity	0.11	26.73	0.41

Table 1: Initial Measurement

1.1 Decomposition Strategy

From the table above, the function poisson has the highest runtime compared to other functions. Therefore, parallelization of this function will have a significant impact on overall runtime. The input variables imax and jmax are, by default, 660 and 120, respectively.

1D decomposition : The idea is to split the number of tasks (imax) to be distributed evenly to each processor. The number of tasks for each processor can be obtained by dividing the total number of iterations (imax) by the size of the processors. From the rank number, the initial value and final value for each processor can be calculated. The jmax value is not necessarily to be split, as it will create more complex memory communication between processors.

Send and receive data: Since the poisson function is calculated twice, starting with red and then black cells in 2D iterations, the list of arrays from the first iteration on the edge must be sent to the edge of neighbouring processors before the second iteration takes place to keep the data updated. Here is where MPI call send and receive will be allocated to send and receive with adjacent processors.

Gather and broadcast data : Before exiting the function, an MPI call must be made to update the pressure matrix on all processors. Therefore, MPI's all gather function will be used to

collect and update the pressure matrix in all processors. The all-gather function must also be flexible on how to gather from uneven workload distribution. Since the pressure matrices will be used in the next step process.

Barrier handling : A barrier must be put in place to avoid any race-condition happening in every step of the process.

Comparison : Comparing the parallelize with the original karman is crucial to ensuring the generated optimization karman is accurate.

Different number of processors : After an accurate result is obtained, the code will be run on different processors for analysis purposes.

2 Optimization

2.1 MPI syntax

2.1.1 poisson

```
/* Calculate sum of squares */
for (i = initial_imax; i <= end_imax; i++) {
    for (j=1; j<=jmax; j++) {…
}

MPI_Allreduce( &p0, &p0, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD );

p0 = sqrt(p0/ifull);
if (p0 < 0.0001) { p0 = 1.0; }

/* Red/Black SOR-iteration */
for (iter = 0; iter < itermax; iter++) {
    for (rb = 0; rb <= 1; rb++) {
        //#pragma omp parallel for
        for (i = initial_imax; i <= end_imax; i++) {
            for (j = 1; j <= jmax; j++) { …
        } /* end of i */

        //All boundaries p will be send to adjacent processors
        // Use the rank of the sending processor as the tag
        MPI_Sendrecv(&p[initial_imax][1], jmax, MPI_FLOAT, rankL, 0,             // sending to left
                     &p[initial_imax-1][1], jmax, MPI_FLOAT, rankL, 0,           // receiving from left
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        MPI_Sendrecv(&p[end_imax][1], jmax, MPI_FLOAT, rankR, 0,            // sending to right
                     &p[end_imax+1][1], jmax, MPI_FLOAT, rankR, 0,          // receiving from right
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } /* end of rb */

    /* Partial computation of residual */
    *res = 0.0;
    for (i = initial_imax; i <= end_imax; i++) {
        for (j = 1; j <= jmax; j++) {…
    }

    MPI_Allreduce(&*res,&*res, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD );

    *res = sqrt((*res)/ifull)/p0;
    /* convergence? */
    if (*res<eps) break;
} /* end of iter */

MPI_Allgatherv(&p[initial_imax][1], total_imax_iteration_by_rank*(jmax+2), MPI_FLOAT, &p[1][1],counts,displacements, MPI_FLOAT, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD );
return iter;
```

Figure 2: MPI syntax

Overview : The first MPI Allreduce will be used to sum all squares in every other processor. Inside Red/Black SOR operation, 4 MPI call functions will be made: 2 send functions are needed to send data to the right and left of current processors, and the other 2 are needed to receive data from the right and left of current processors. Another MPI Allreduce function in the computation of residual is called to sum all residual values in each processor. Before exit function, MPI Allgatherv is called to update all pressure matrices in each processor. Here all offsets and receive count values must be the same for all processors. An MPI Barrier function is called to ensure all processors reach this barrier before the next step process.

2.1.2 compute_tentative_velocity

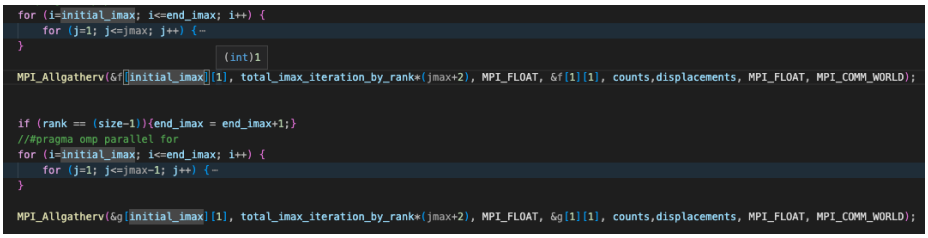


Figure 3: MPI syntax

Overview : 2 MPI Allgather functions will be used to collect f and g matrices for each processor.

2.1.3 compute_rhs

Figure 4: MPI syntax

Overview : A single MPI Allgather function will be used to collect RHS matrix for each processor. An MPI Barrier before the exit.

2.1.4 update_velocity

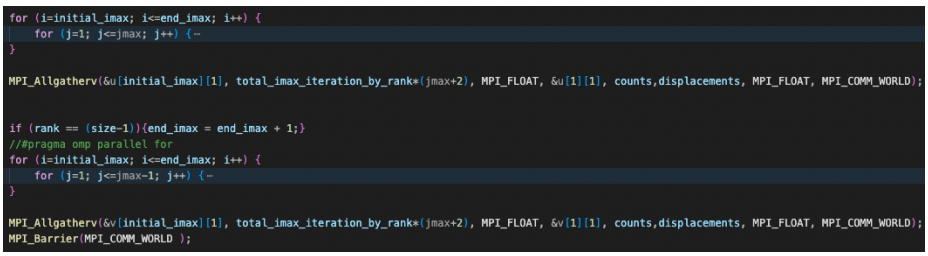


Figure 5: MPI syntax

Overview : 2 MPI Allgather functions will be used to collect u and v matrices for each processor. An MPI Barrier before the exit.

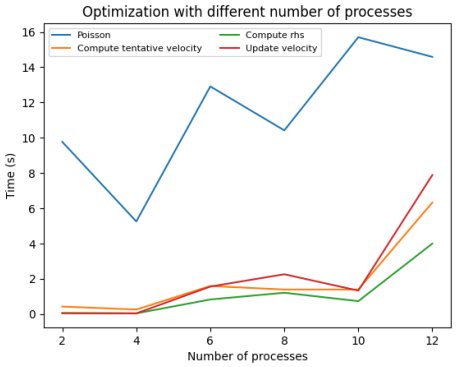
2.2 MPI Parallelization Result

2.2.1 MPI Parallelization Profile (small input size)

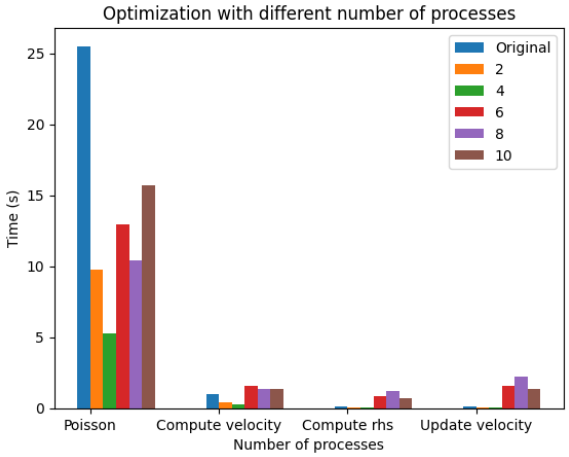
Below is the performance result for the first 12 ranks with default input value 660 (imax)x 120 (jmax).

Function	Cumulative runtime (s)					
	2	4	6	8	10	12
poisson	9.7668	5.2519	12.9128	10.4211	15.7057	14.5900
compute_tentative_velocity	0.4228	0.2534	1.5878	1.3892	1.3892	6.3246
compute_rhs	0.0501	0.0360	0.8240	1.2073	0.7291	3.9940
update_velocity	0.0463	0.0346	1.5567	2.2551	1.3270	7.8848

Table 2: MPI Optimization



Parallel Runtime (s)



Parallel Comparison (s)

Figure 6: MPI Parallelization

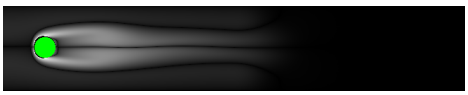


Figure 7: Parallelize Karman

Overview : As the number of processors increases, the runtime for all functions varies. And the generated karman.bin file is identical to the original file.

2.2.2 MPI Parallelization Profile (larger input size)

Below is the performance result for the first 12 ranks with larger input value 1320 (imax)x 240 (jmax).

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 94.92    198.00   198.00                             poisson
  4.10    206.55     8.54                             compute_tentative_velocity
  0.54    207.67     1.12                             compute_rhs
  0.32    208.34     0.67                             update_velocity
  0.15    208.65     0.31                             set_timestep_interval
  0.05    208.75     0.10                             apply_boundary_conditions
```

Figure 8: Profile of serial larger input size Karman

Function	Cumulative runtime (s)					
	2	4	6	8	10	12
poisson	77.7595	41.1267	51.0656	40.7118	49.8497	42.3785
compute_tentative_velocity	3.4308	1.9358	9.6534	13.9359	15.3447	18.1194
compute_rhs	0.4174	0.0360	4.6952	6.4702	7.7490	8.6407
update_velocity	0.3633	0.4280	9.9865	13.5902	15.2810	17.3948

Table 3: MPI Optimization

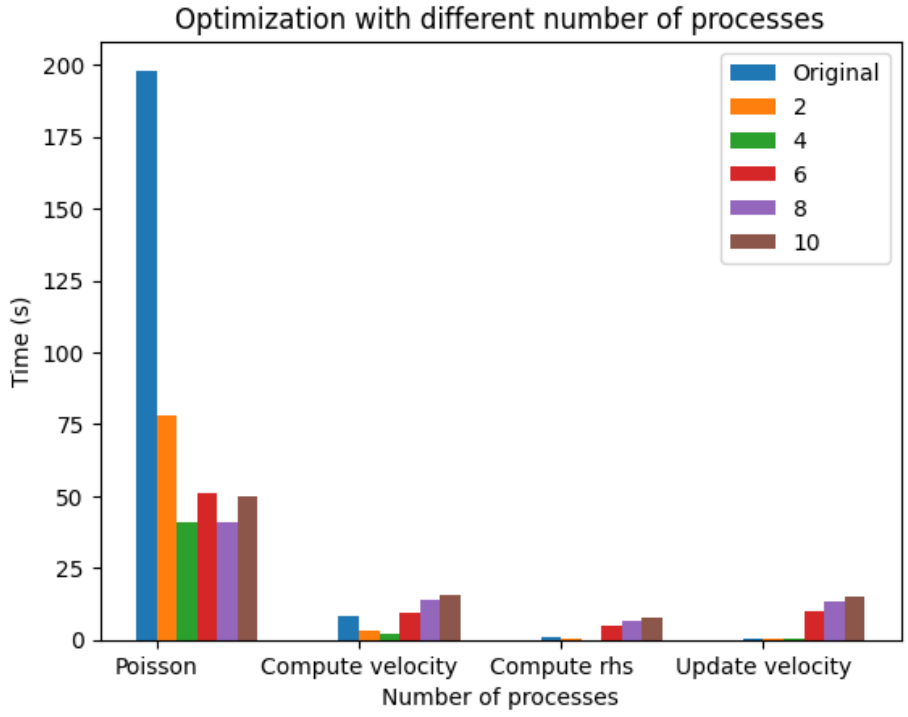


Figure 9: Bar chart of larger input size Karman

Overview : As the number of processors increases, the runtime for all functions varies. The best performance for larger-size input is with four processors.

2.3 OpenMP - MPI Parallelization

2.3.1 OpenMP - MPI Parallelization Profile

For the implementation of OpenMP, the usage of big problems is used to see the difference in performance. Below is the table to see the comparison of the poisson function when OpenMP is used with different numbers of nodes and ranks. Because the subtask generated between ranks is still large at higher imax values, OpenMP will be used to split the task between red and black iteration.

MPI	OpenMP	Nodes	Tasks/node	Cpus/task	time (s)
✓	✗	1	4	1	41.1068
✓	✗	4	1	1	69.7531
✓	✗	4	1	4	73.7585
✓	✓	4	1	4	71.4573
✓	✗	4	4	1	48.0445
✓	✗	2	2	2	59.5283
✓	✓	2	2	2	59.9589
✓	✗	2	4	1	40.4368

Table 4: OpenMP - MPI Optimization

Discussion : Under 1 node, 4 tasks-per-node, 4 ranks are communicating within a single node, which involves sending data between different processes or threads running within the same computer system. This communication is typically faster and has a lower latency than communication between different nodes, as the data does not need to travel over a network. As can be seen from the first row in the table.

On the other hand, communicating between different nodes involves sending data between different computer systems connected by a network. This type of communication is typically slower and has a higher latency than communication within a single node, as the data needs to travel over the network and may encounter congestion or other issues that can affect its delivery. From the second row, the time is increased due to communication between four different nodes. Tuning for the fastest communication between nodes can be seen in the last row, where the total ranks are 8, and the time is relatively lower than communication within a single node.

OpenMP does not significantly improve performance in this case, but it does show some decrease in time at higher node numbers, as can be seen from the 3rd to the 4th row in the table above. Therefore, in this case, openMP applications will not be applicable, as the time of execution has already been sped up to almost five times with MPI alone.

3 Overall

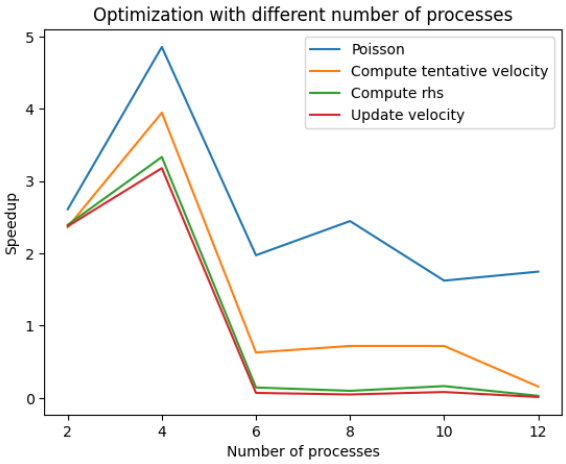


Figure 10: Overall speedup

Overview : From the line chart, the speedup increases until it reaches a maximum at 4 processors. Possible reasons include communication overhead, which can arise when parallel tasks need to exchange data or synchronise their actions in order to coordinate their computations. For example, in a distributed-memory parallel system, tasks running on different nodes need to communicate and exchange data using message-passing interfaces (such as MPI) in order to coordinate their computations. This communication involves sending messages, waiting for messages to arrive, and coordinating the order in which messages are sent and received, all of which can introduce significant overhead.