CS402 Assignment 2: Parallelize the Karman code

jiyu yan 1851015

I. Introduction

The code in this assignment calculates the velocity and pressure of a 2D flow and writes a binary file that contains solution values. The purpose is to parallelize the code using MPI, OpenMP and analyze the performance with original sequential code.

There are two types of cell, fluid and obstacle, represented as a 2D grid, and the main job is to calculate one point in each cell. The properties for each cell include u(x velocity), v(y velocity), p(fluid pressure), flag(cell type). The communication pattern is using a five-point stencil to calculate a point, by checking whether the solution satisfies the governing equations and cyclically generate the new one based on current stencil if not met the condition.

II. Code analysis

To understand the background and for further performance improving, the original code are analyzed and tested in this section.

The successive over relaxation(SOR) method could speed up convergence as it is the method to generate new solution by solving the Posson equation, and the key point here is that it introduces a chain of dependencies for each cell.

With the help of red/black ordering calculation, the cells are divided into two populations and they could be updated in parallel since the stencil for any cell is the cell itself and cells of opposite color, work could be balanced across processors using less communications.

The main loop is located in *main* method of *karman.c* file, which is responsible for the most parts of computation. At first it calculates an approximate time-step size using *set_timestep_interval* method, then for each cell, a tentative new velocity(f,g) based on previous(u,v) values and flag matrices is updated using *compute_tentative_velocity* method, together with the RHS matrix by taking the f, g and flag matrices as input using *compute_rhs* method.

The most computationally intensive part is to use red/black SOR here to solve the Poisson equation in *poisson* method to update pressure matrix. Finally the (u,v) values are updated using *update_velocity* methods and the cell adjacent to edge is using *apply_boundary_conditions* method. Apart from *apply_boundary_conditions* which is defined in *boundary.c*, above methods are all defined in file *simulation.c*.

TABLE I
Default running time

function	self seconds	percent
poisson	21.39	97.01
compute_tentative_velocity	0.48	2.18
update_velocity	0.08	0.36
compute_rhs	0.07	0.32
set_timestep_interval	0.04	0.18

With the help of the profiler gprof, the time behaviour for main routines in the code could be recorded. Although many methods are added at first to compute the time for each specific method using MPI_Wtime(), they are commented out finally since the convenience and accuracy of gprof. The results are shown in tableI, by testing the original serial code with *test/initial.bin* as input. Around 97% time is spent on *poisson* function, which gives us a clear message about which part of code should be analyzed and paralleled.

III. Parallelize with MPI

A. Decomposition

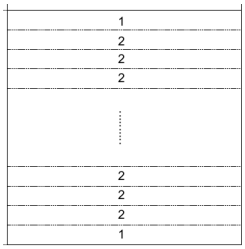

Fig. 1. 1D decomposition

Whether using 1D or 2D decomposition is the first problem. The difference between them are shown in Figure1 and Figure2. The number in figures denote the necessary communication frequency for each processor to transmit the boundary data for updating in SOR method.

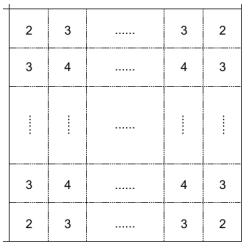

Fig. 2. 2D decomposition

Suppose the original data dimension is I*J and the I dimension is divided in 1D decomposition, the size of processors is p and it could separate the data evenly. Then the total data transmission for 1D decomposition is $(p-2)*2*(J+2)$, plus the processors at both ends: $2*(J+2)$, results in $(2p-2)*(J+2)$.

In 2D decomposition, as shown in Figure2, suppose the I dimension is divided by p and J dimension is divided by q, the processors labeled by 4 need $2*(I+2)/p+2*(J+2)/q$, which accounted for the vast majority of the calculations. Due to the complexity and scalability, I dimension 1D decomposition is used.

The visualization of solution for *initial.bin* using program *bin2ppm* is shown in Figure3, after code

Fig. 3. Solution for initial.bin by serial code

parallelism the picture need to be kept same and the files need to be identical by using *diffbin* program to check.

As shown in Listing1, Imax are divided by the number of processors. Although program should execute with any input size theoretically, something went wrong and may differ bin files when Imax couldn't be divided by the number of processors in integer. The size of the input(Imax) must match the number of processes to load balance and ensure the correct result.

```
1  MPI_Status status;
2  MPI_Init(&argc, &argv);
3  MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
       ;
4  MPI_Comm_rank(MPI_COMM_WORLD, &proc);
5  if (imax % nprocs != 0) {
6    if (proc == 0){
7      printf("Imax␣can't␣be␣divided␣by␣%d
           .\n", nprocs);
8      MPI_Abort( MPI_COMM_WORLD, 1 );
9    return -1;
10   }
11 }
12 each_part = imax / nprocs;
13 ileft = proc * each_part + 1;
14 iright = min(imax, each_part * (proc +
       1));
```

Listing 1. decompositoin

B. Parallize the Poisson method

As shown in TableI, *poisson* method cost most of time and thus is the foremost method we should focus on. There are three parts in this method could be parallelized:

1) Calculate sum of squares
2) Red/Black SOR-iteration
3) Partial computation of residual

The SOR part must be parallelized, which contains a four-layer for loop. After decomposition and using ileft, iright to denote the left and right bound of each processors, rather than the original serial

Fig. 4. wrong result by removing Listing2, although the bin files are same

code, it is important to pass the data across every bound correctly.

Listing2 shows how to use MPI_Send and MPI_Recv to finish the work. Only the leftmost and rightmost parts need to be passed in one direction, others in the middle all passes bidirectional. If missing this part of code, although the final bin file is same to the original serial code output by using diffbin to check, the bin2ppm would show the differnece as shown in Figure4 to prove this part of code is essential.

```
1 if (proc != nprocs - 1) {
2   // Pass only to the right
3   MPI_Send(&p[iright][0], jmax+2,
        MPI_FLOAT, proc+1, 0,
        MPI_COMM_WORLD);
4   MPI_Recv(&p[iright+1][0], jmax+2,
        MPI_FLOAT, proc+1, 0,
        MPI_COMM_WORLD, &status);
5 } if (proc != 0) {
6   // Pass only to the left
7   MPI_Recv(&p[ileft-1][0], jmax+2,
        MPI_FLOAT, proc-1, 0,
        MPI_COMM_WORLD, &status);
8   MPI_Send(&p[ileft][0], jmax+2,
        MPI_FLOAT, proc-1, 0,
        MPI_COMM_WORLD);
9 }
```

Listing 2. Passing data across the boundary between processors, second part of poisson

The first part in *poisson*, which used to compute the sum of squares, could be parallelized using the code shown in Listing3, after changing the bound of (1, imax) in for loop to (ileft, iright). The *p0* is reduced from each part to *p0_all* and then we need to put the results to each processor, as shown in Listing3.

```
1 MPI_Allreduce(&p0, &p0_all, 1,
      MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD
      );
```

Listing 3. Allreduce method in first part of poisson

The last part is used for partial computation of residual, which also could be parallelized using MPI_Allreduce, similar to the first part, to sum up the add*add part in each processor and put the add-up results back, shown in Listing4.

```
1 MPI_Allreduce(&add2, &add2_all, 1,
      MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD
      );
```

Listing 4. Allreduce method in third part of poisson

Finally, before return *iter* at each step, the *p* matrix need to be updated using *MPI_Allgather*[1] method. It is same to gather all parts from each processor to processor 0(root processor) and then broadcasting the total *p* matrix from root to all processors. Rather than using tedious way shown in Listing5, it is elegant to use *Allgather* method as Listing6 instead to gather *p* matrix at each processor and then put the updated *p* in each processor.

```
1   if (proc == 0) {
2   MPI_Gather(MPI_IN_PLACE, 0,
        MPI_DATATYPE_NULL, &p[1][0],
        each_part*(jmax+2), MPI_FLOAT, 0,
         MPI_COMM_WORLD);
3 } else {
4   MPI_Gather(&p[ileft][0], each_part*(
        jmax+2), MPI_FLOAT, NULL,
        each_part*(jmax+2), MPI_FLOAT, 0,
         MPI_COMM_WORLD);
5 }
6 MPI_Bcast(&p[0][0],(imax+2)*(jmax+2),
      MPI_FLOAT,0,MPI_COMM_WORLD);
```

Listing 5. tedious way to transmit data at the end of poisson method

```
1 MPI_Allgather(MPI_IN_PLACE, each_part
      *(jmax+2), MPI_FLOAT, &p[1][0],
      each_part*(jmax+2), MPI_FLOAT,
      MPI_COMM_WORLD);
```

Listing 6. Allgather method at each iter at the endof poisson

C. Performance

Are all three parts of poisson worth parallelizing? Intuitively and theoretically only the second part(SOR iteration) must be parallelized. To find out the answer, code with and without parallelization of the first and third parts in poisson was tested, with different number of processors 2,4,6 and the

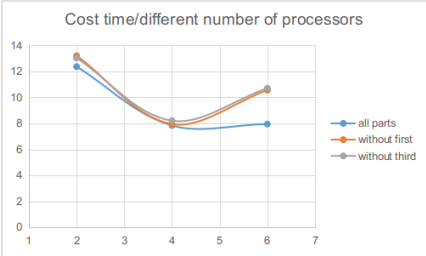

Fig. 5. three parts parallelizing in poisson

test/initial.bin as input. All results are the average number of 3 experiments by using gprof, to measure the total time(the last line of cumulative seconds in gprof) of the whole program in different situations.

TABLE II
THREE PARTS PARALLELIZING

Time cost	2	4	6
all parts parallelized	12.37	7.84	7.95
without first part	13.21	7.95	10.56
without third part	13.04	8.21	10.71

From TableII, all three parts of poisson deserve to be parallelized, as it took less time than two other situations, either without first part or third parallelization part.

Another finding is that, the running time decreases when increasing the number of processors from 2 to 4, and then increases from 4 to 6, with the same input, as shown in Figure5. To confirm this discovery, the influence of the number of processors and input size are tested deeper.

In order to using 2 to 8 processors, the 480.bin and 960.bin files are made since those number could be divided by 2,4,6,8. By running script below, the results shown in Table.

```
1 mpirun -np 8 ./karman -x 960 -i 960.
      bin -o out.bin
2 gprof karman > 960_8.gprof
```

Listing 7. mpirun example

This result is consistent with our expectation that parallelization leads to a reduction in overall time as the process begins to increase from 2 to 4,

TABLE III
WITH INPUT 960.BIN

Time cost	2	4	6	8
poisson	21.72	14.63	19.14	14.61
all	23.13	16.38	22	18

TABLE IV
WITH INPUT 480.BIN

Time cost	2	4	6	8
poisson	6.11	3.65	4.67	3.54
all	6.52	4.08	5.49	4.67

and as the number of processes increases further from 4 to 6, the overhead caused by parallelization becomes more apparent, same as the results above, indicating that the process number is not 'the more the better'. We can find out what the reasonable number of processes is for each input by this kind of experiments.

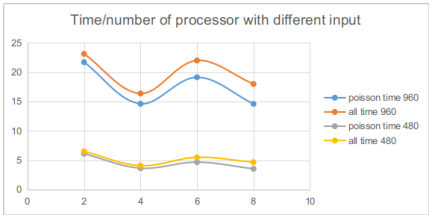

Fig. 6. time/processors with different input

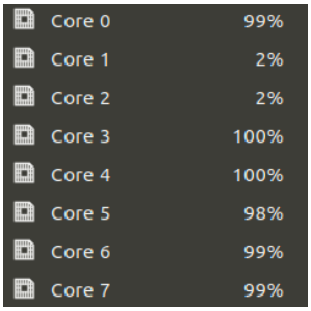

Fig. 7. cpu utilization, tested by 6 processors

However, unexpectedly, as the processors increases from 6 to 8, the time decreases again and the results are similar as using 4 processors, rather than increases as the overhead is more apparent

as we expected, as shown in Figure6. This is very interesting and it may related to the hardware, since I tested on my own laptop with Intel Core i5-8250U CPU @ 1.60GHz x 8 and MPICH implementation. In all eight core cases, compared to the six cores shown as Figure7, it may bring other benefits such as hardware symmetry, further improving parallel performance.

Still, it could be reasonably speculated that if other conditions are the same, if the number of processes can continue to increase from 8, the time consumed by the program would rise again and continue to increase as the number of processes increases, due to the inevitable parallelization overhead.

D. Futher parallelize

Apart from *poisson* method, other methods such as *compute_tentative_velocity*, *update_velocity* and *set_timestep_interval* could also be parallelized using MPI.

Since the second time-consuming method is *compute_tentative_velocity*, it is parallelized and the way to pass the data across bounds, and using MPI_Alltather to update f and g matrix is similiar to previous method. It named poisson plus in TableV as only *poisson* and *compute_tentative_velocity* is parallized.

Based on this, all possible methods in the main loop of karman.c are parallelized finally, means all possible (i=1, i¡=imax) is instead by (i=ileft, i¡=iright) and at the end of method *update_velocity*, all matrices are calculated correctly using MPI_Allgather. The advantage here is now we only need to broadcast the matrix once, and when Allgather is in the possion before, every iteration actually calls MPI_Allgather and broadcasts the p matrix.

After several tests(average of five tests for each version), with input test/initial.bin, the results shown in TableV meet our expectations. This all-parallelized version is the best one although it is just slightly better than the other when the number of processor is 2 and 4. When it is 6, the best one is poisson-only version since it brings the least overhead. The all-parallelized version is our final choose since the good performance under all three processor number conditions.

TABLE V
COMPARISON OF DIFFERENT VERSION CODE

Time	2	4	6
only poisson	12.04	7.84	8.19
poisson plus	12.04	7.40	8.94
all parallelized	11.16	7.05	7.87

IV. PARALLELIZE WITH A HYBRID APPROACH

Limited to the hardware environment(my own laptop), the appropriate number of threads and the number of processes needed to be chosen before the experiment. From practical experience, the sum of thread number in each processor number shouldn't over 8. If for example 4 processors are used and 4 threads are set in openMP environments, $4 * 4 = 16 > 8$, it could cause additional serious overhead and have a big loss in performance.

Before adding any code of openMP, interestingly I found just adding the head line below could bring some overhead, as shown in TableVII. With input test/initial.bin, number of processor and thread both equals to 2.

```
1 #INCLUDE <OMP.H>
```

After adding a little omp realated code and several tests, by setting the processor number to 2 and the omp threads from 2 to 4, the performance of (processor=2, thread=2) is clearly better than (processor=2, thread=4), since the latter has too many threads, introducing unnecessary overhead when there is already mpi parallelism. So the number of processes and the number of threads in the following experiments are set to 2.

TableVIII shown the results when omp related code are added in karman.c(first line of Listing 8) and *compute tentative velocity* method. After comparing (i,j),(du2dx,duvdy,laplu) and (i,j,du2dx,duvdy,laplu), the middle one is chosen and the code is shown in line 2,3 of Listing8.

```
1 #PRAGMA OMP PARALLEL FOR SCHEDULE(
      STATIC) PRIVATE(I,J)
```

TABLE VI
OMP OPTIMIZATION

total time	rhs	pos 1 reduction	ij ,beta mod	beta mod	pos 2 reduction	upd_v	set_time
test1	11.42	13.12	12.01	9.36	13.19	9.31	11.63
test2	11.18	13.12	10.05	9.57	13.01	14.35	13.57
test3	11.6	11.18	12.55	9.13	13.18	11.06	15.93
avg	11.4	12.473	11.536	9.353	13.127	11.573	13.71
whether use	no	no	no	yes	no	no	no

TABLE VII
OVERHEAD OF OMP.H

total time	original	only add omp.h
test1	11.52	12.83
test2	10.58	12.83
test3	9.6	13.29
test4	10.15	12.38
test5	9.87	12.49
avg	10.344	12.764

TABLE VIII
OMP COMPARISON

total time	ij	kar	kar+ij	kar+3	kar+5
test1	12.79	12.66	12.59	11.6	12.89
test2	13.14	12.27	12.47	14.47	11.18
test3	12.79	12.37	11.5	11.81	12.45
test4	11.57	12.27	11.63	10.44	13.05
test5	11.43	10.96	12.27	10.53	12.41
avg	12.344	12.106	12.092	11.77	12.396

```
2 #PRAGMA OMP PARALLEL FOR SCHEDULE(
      STATIC) PRIVATE(DU2DX,DUVDY,LAPLU)
3 #PRAGMA OMP PARALLEL FOR SCHEDULE(
      STATIC) PRIVATE(DUVDX,DV2DY,LAPLV)
4 #PRAGMA OMP PARALLEL FOR SCHEDULE(
      STATIC) PRIVATE(BETA_MOD)
```

Listing 8. decompositoin

Similarly, some other omp-related codes are added in and tested, the results shown in TableVI, only the beta_mod one is better to use, as shown in line 4 of Listing8, others are all commented out.

All openmp tests are based on the MPI code bofore. Multiple processors are connected in parallel with mpi and each processor uses openmp within some parts of the code for multi-thread shared memory parallelism. The advantage of this is that you can reduce the number of processes in the mpi parallel, thus reducing the number of messages (and possibly the size) to increase the speed; the openmp on each node can save memory overhead in parallel. So this hybrid parallel approach has certain advantages over a single parallel approach.

However, with same input test/initial.bin, the results 9.353s with (nprocs=2,threads=2) omp version, is just a little better than 10.344s(pure MPI version, nprocs=2) and much worse than pureMPI with nprocs=4 which is 7.05s. While openmp introduces certain overheads, the benefits are limited. In actual use, the use of this hybrid mode is not as widespread as pure MPI.

V. CONCLUSION

Through this assignment, the karman code was performed in pure MPI and MPI,openMP mixed version in parallel. By optimizing to the specific inputs and choosing the appropriate number of processors, and the number of threads in openMP, the execution time of the code can be greatly reduced by parallelism. Excessive number of threads and processes can lead to excessive overhead and reduce the benefits of parallelism. For specific inputs, which functions require MPI parallelism and where they can be optimized with openMP, actual testing is necessary and could lead to clear results. By comparing the results, the hybrid mode can bring a little benefit on the basis of MPI parallelism. In actual use, MPI is more commonly used based on scalability and flexibility.

REFERENCES

[1] D. L. He, “Message Passing Programming II,” *Computer Science*, p. 40.