CS402 Assignment 1: OpenMP for deqn code

Jiyu Yan
Student ID:1851015
High Performance Computing

I. Introduction

The deqn program is used to solve the diffusion equation in two-dimensions given specific time point and space location, mainly by iterating over the u0 array and updating the value in the u1 array with correct formula.

The purpose of this assignment is to parallelize appropriate loops in the code with OpenMP functions, and analysis the corresponding results such as runtime and overhead with different parameters, try to achieve good speedup.

II. Code analysis

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

The *main* method accepts the input file and creates a new *Driver* object, calls its *run* method. The *driver* then creates the *mesh* and *diffusion* object. *Mesh* holds the information about grid or array storing data specifically, *diffusion* sets inital data to *mesh* and calls *doAdvance* in *scheme* class. Majority of calculation including *diffuse*, *reset* and *updateBoundaries* are encapsulated in *doAdvance* method of *ExplicitScheme* class.

Which part of original code cost the most of time is the first question. 3 square input files(square1.in, square2.in, square10.in) are tested and the results getting from gprof command are shown in tableI.

TABLE I
Time% taken of all functions

Function/input file size	1000^2	2000^2	10000^2
VtkWriter::writeVtk	38.71	25.01	32.94
ExplicitScheme::diffuse	25.81	20.59	15
Mesh::getNx	8.07	17.65	17.07
ExplicitScheme::reset	12.9	14.71	12.28
Mesh::getTotalTemperature	3.23	10.3	6.96
Mesh::getU0	1.61	5.88	3.7
Mesh::mesh	1.61	2.94	1.74
Mesh::getDim	4.84	1.47	6.2
Diffusion::init	0	1.47	4.13

The most time is spent on *VtkWriter* which is responsible for writing running output data to some files, which couldn't be parallelized since the output must be consecutive. However, the default vis_frequency parameter with 1 of input file could be changed to -1, to speedup whole program running time to help us focus on the calculation part.

The two methods of *ExplicitScheme* class in *doAdvance*: *diffuse* and *reset*, cost the most of time. Since all 3 methods of *doAdvance* are the main part for diffusion calculation and are executed at each step. The *updateBoundaries* is not as time-expensive as these 2 methods because after every heavy calculation above, it only reflects data of boundaries and the complexity is small compared to *diffuse* and *reset*.

TABLE II
Number of calls for of each function

Function/input file size	1000^2	2000^2	10000^2
getNx	422084401	96096230	1200240116
getTotalTemperature	20	11	5
getU0	21000125	48000071	600000035
mesh	1	1	1
getDim	21	12	6

In *mesh* class, from tableII, *getU0* and *getNx* are time-expensive overall due to the high number of its calls rather than its own complexity. Observing the original code, those methods with the addition of *getDim* are all one-line code simply for return value. So in tableI now the primary time-consuming method of mesh class which could be improved with openMp is getTotalTemperature.

The last one in tableI, Diffusion::init, would cost more time when the grid gets bigger. That make sense since it calculates all values for u0 array at the beginning of program running.

III. Parallelize with OpenMP

No loop is included in *main*, and the *run* method of *Driver* class couldn't be parallelized since there is a dependency for *dt* at each step. The *Mesh*, *ExplicitScheme*, *Diffusion* class are the left 3 classes which could be parallelized for improving performance in this program.

The gprof is not suitable for multi-thread application in some situations due to its output could be incorrect and the instrumentation overhead could be high[1]. So in this section the change(shown in appendix) of makefile are changed back, and apart from the total program running time measurement method, three methods shown in tableIII are added into code located at class *ExplicitScheme*, *Mesh* and *Diffusion*, targeting for loop time measurement in these methods.

Based on input file omp.in, which is 10000^2 grid with 20 steps, to give enough running time to each loop and with parameter vis_frequency -1 to reduce unnecessary time, all original loop and total time cost are tested and the results shown in tableIV. The result is an average of three experiments.

All loop time measurement is calculated using the sum of those loop time taken at each step divided by step number. To prove these loop measurement methods are correct, the sum of all loop mean time taken is calculated.

$0.126 + 0.296 + 0.118 + 0.001 = 0.514$

It's very near the each step time 0.544, means most time are spent by these loops and the time measurement method is accurate enough.

TABLE III
LOOP TIME MEASUREMENT METHOD

Time measurement function	Target
getExplicitSchemeLoopTime	Diffuse loop
getExplicitSchemeLoopTime	reset loop
getExplicitSchemeLoopTime	update loop
getMeshLoopTime	get totalTemperature loop
getDiffusionLoopTime	Diffusion init time

TABLE IV
ORIGINAL CODE TIME TAKEN WITH OMP.IN

Each step mean time	0.544
Total tempature loop mean time	0.126
Diffuse loop mean time	0.296
Reset loop mean time	0.118
Update Boundary loop mean time	0.001
Diffusion init loop mean time	0.601
Total time	10.880

A. Mesh::getTotalTemperature

After the above analysis, only *getTotalTemperature* method need to be improved in mesh class. This method executes at each step and it is suitable to use reduction method of OpenMP for temperature addition here since all threads need to add the computation result together to get correct total temperature. The following line are added before the for loop in this method, then the static way is also tested of this parallelization and the results are shown in tableV.

TABLE V
TOTAL TEMPERATURE LOOP TIME TAKEN WITH OMP.IN

Temperature loop	static	dynamic	original
test1	0.038	0.032	0.126
test2	0.036	0.030	0.122
test3	0.035	0.031	0.129
mean	0.036	0.031	0.125

```
pragma omp parallel for schedule(dynamic)
     reduction(+:temperature)
```

Compared to the original 0.125, the dynamic way speeds up this loop 71.4% around. Dynamic scheduling introduces some additional overhead, but in this case it can lead to a better workload allocation.

B. Diffusion::init

Although the Diffusion init method only runs once before all computation steps, the time taken here is non negligible when the grid is huge. Since there is a 10000^2 grid in the omp.in file, the loop could be parallelized to try to reduce time taken.

TABLE VI
DIFFUSION INIT TIME TAKEN WITH OMP.IN

Diff. init	static	dynamic	original
test1	0.154	0.17	0.58
test2	0.135	0.183	0.63
test3	0.131	0.179	0.589
mean	0.14	0.177	0.599

As the tableVI shown, parallelization could reduce some time with input omp.in and the static way here performs better than dynamic, since the problem size at each loop is around equal and the benefit of latter could not surpass its overhead.

C. ExplicitScheme

Three loops in this class are analyzed above, *diffuse*, *reset* and *updateBoundaries*. As the *updateBoundaries* costs a little bit time compared to other methods at every situation so it could just be ignored.

1) reset: Both static and dynamic commands perform a little better than original code and the dynamic way is slightly preferable as shown in tableVII, implies the calculation amount is not stable as the loop executes from beginning to end.

TABLE VII
RESET LOOP TIME TAKEN WITH OMP.IN

reset loop	static	dynamic	original
test1	0.1018	0.0974	0.1187
test2	0.102	0.09685	0.1155
test3	0.1004	0.09735	0.11905
mean	0.1014	0.0972	0.11775

2) diffuse: Some other methods are tried such as collapse in OpenMP here, however, it didn't bring any gain. As shown in tableVIII, the dynamic way performs best among all methods with input omp.in.

TABLE VIII
DIFFUSE LOOP TIME TAKEN WITH OMP.IN

diffuse loop	static	dynamic	collapse	original
test1	0.1152	0.10815	0.1833	0.30705
test2	0.11175	0.1059	0.1738	0.28285
test3	0.109	0.1064	0.17525	0.2993
mean	0.112	0.107	0.178	0.2964

D. *Overall speedup*

After all changes, the comparison of original and parallelized code are made with 5 different files:square.in, square1.in, square4.in, square8,in, omp.in(equal to 1 to 5 on x axis in figure1). As the grid getting bigger, the performance of parallelized code getting more better than the original code.

Only the reset method is an exception, shown in lower left corner in figure1, as the changed code only slightly better when the input grid is big enough(omp.in). And only for smallest grid with input square.in, the original code performs better because of overhead, As shown in tableIX.

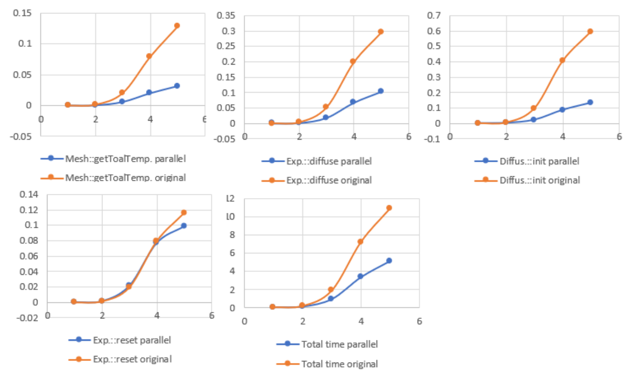

Fig. 1. Time comparison with different input

TABLE IX
SPEED UP

Input File	square	square1	square4	square8	omp
speedup	-25.80%	27.10%	49.50%	53.70%	53.20%

E. *Overhead*

In consideration of overhead, the input square.in with much smaller grid 100^2 is tested. Unsurprsingly, the original code perform a lot better than parallelized code. That make sense because the grid now is rather small so there is no need to introduce OpenMP as the overhead now is very huge. However, time for overhead still is rather small and acceptable considering the benefit of parallelization for normal or huge grid as shown in figure1, so the code change is kept in file.

TABLE X
TIME TAKEN WITH SQUARE.IN

square.in	parallel	original
Mesh::getToalTemp.	0.00015	1.39E-08
Diffus.::init	0.001	1.00E-10
Exp.::reset	0.0003	5.00E-05
Exp.::diffuse	0.0002	5.01E-05
Total time	0.073	0.058

F. *Thread number*

As shown in figure2, the Diffusion::init method gets the most benefits(0.594-0.133=0.461s) when the input is omp.in, so the effects of thread number change is tested at this loop since it would give clearer results.

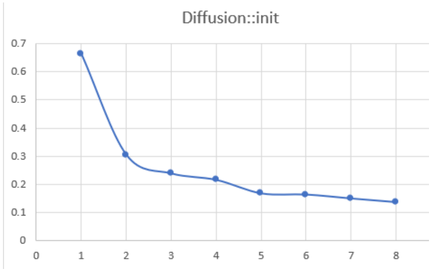

Fig. 2. Thread number effects with omp.in

The result is shown in figure2, as the thread number increases from 1 to 2, the time taken decreases around to half of original. From thread number increases from 2 to 8, the benefits get smaller and smaller, and there is no more advantage of 16 threads compared to normal max 8 threads. Since the overall overhead including the thread creation, termination and communication cost a lot time so the normal 8 max threads is suitable for this situation.

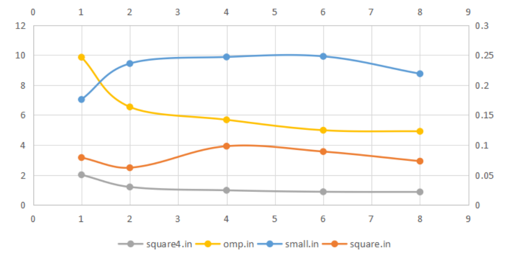

Fig. 3. Different file running time with variant thread num.

At last the impact of threads number on total run time is tested with different input files, shown in figure3. For lager grid files such as square4.in and omp.in, the time spent decreases as the number of threads increases. However, for smaller gird files such as square.in and small.in(both use right side y axis),multi-threading has no obvious help for total running time, due to overhead compared to the problem size.

IV. Conclusion

After code analyzing and adding time measurement methods, those loops spent more time are found, and parallized with suitable openMP command after comparing static and dynamic parameter in different situations.Parallelization in Mesh::getTotalTemperature, Diffusion::init, ExplicitScheme::diffuse, contributes the most benefits after code revised when the input file has a normal or huge grid. The overhead could make results worse when the grid is small enough such as 100^2, and the increase in the number of threads will, in general, make the performance better as long as the grid size is not too small.

V. Appendix

Running cpu: Inter i5-8250U CPU@1.80 GHz

A. gprof command

Gprof: add -pg to makefile at line 18, and add $(CXXFLAGS_DEBUG) to line37.

```
CXXFLAGS_DEBUG := -g -DDEBUG -pg
```

```
  $(LD) $(CXXFLAGS_DEBUG) $(LDFLAGS) -o $@ $^
```

After compile and run with input file, a gmon.out file would be in build directory. Using the command below(linux/windows) to see the simple version profiles(with -p).

```
gprof deqn gmon.out -p
gprof deqn.exe gmon.out -p
```

B. Test file

square.in: 100^2 with 20 steps
square1.in: 1000^2 with 20 steps
square2.in: 2000^2 with 10 steps
square4.in: 4000^2 with 20 steps
square8.in: 8000^2 with 20 steps
square10.in: 10000^2 with 5 steps
omp.in: 10000^2 with 20 steps
small.in:10^2 with 80 steps

References

[1] "Gprof," https://en.wikipedia.org/wiki/Gprof/.