COM6521: Parallel Computing with GPUs (2019/20)
Assignment: Part 1

User Name: elt18sx

Student Registration Number: 190186546

Implement

The structure of the program consists of standardized preprocessing and simulation. Pre-processing is mainly to read parameters, read or generate data. The simulation part is handled by the step function, which mainly includes compute_volocity and updata_location, and decides whether to update the heat map according to whether the GUI window is displayed. As shown in the pseudo code below,

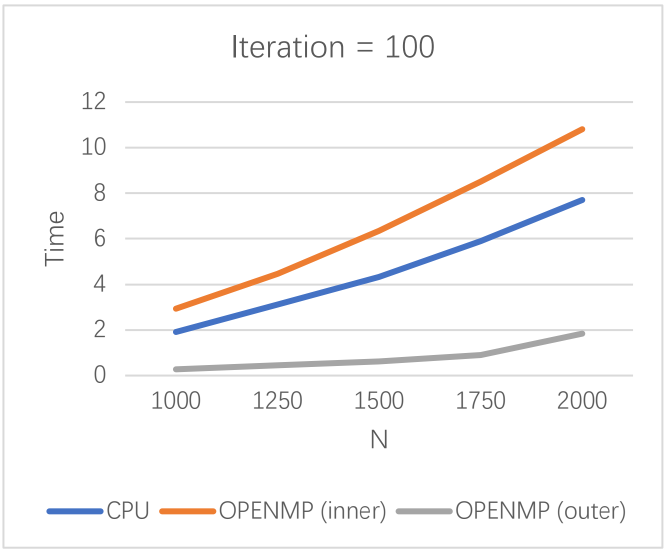
```
function main(argc, argv)
   args <- load_args(argc, argv)

   if args.input_file then
      bodies <- read_file(args)
   else
      bodies <- generate_data(args)
   end if

   if args.visualisation then
      initViewer(args,step)
      setNBodyPositions(bodies)
      setHistogramData(heat_map)
      startVisualisationLoop()
   else
      tic <- clock()
      step()
      toc <- clock()
      print(toc-tic)
   end if

   free(bodies)
   free(heat_map)
   return 0
end function

function step()
   for i = 0 to args.iter do
      compute_volocity(bodies,dt,args)
      updata_location(bodies,dt,args)
      if args.visualisation then
         update_heat_map(heat_map,bodies,args)
      end if
   end for
end function

function compute_volocity(bodies,dt,args)
   if args.m == OPENMP then
      omp parallel for
   end if
   for i = 0 to args.n do
      acceleration <- calculate_single_body_acceleration(bodies,i,args)
      bodies[i].vx <- acceleration.x * dt
      bodies[i].vy <- acceleration.y * dt
   end for
end function
```

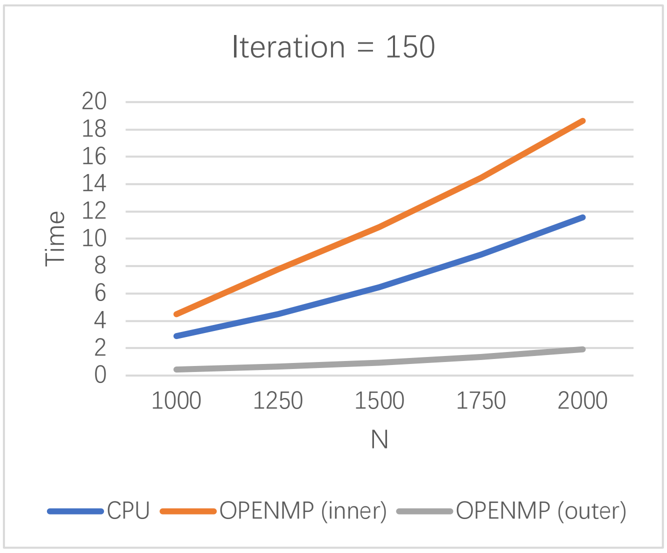
```
function compute_volocity(bodies,dt,args)
    if args.m == OPENMP then
        omp parallel for
    end if
    for i = 0 to args.n do
        acceleration <- calculate_single_body_acceleration(bodies,i,args)
        bodies[i].vx <- acceleration.x * dt
        bodies[i].vy <- acceleration.y * dt
    end for
end function

function calculate_single_body_acceleration(bodies,index,args)
    acceleration <- {0,0}
    target_body <- bodies[index]
    for i = 0 to args.n do
        external_body <- bodies[i]
        acceleration.x <- acceleration.x + acceleration from external_body
        acceleration.y <- acceleration.x + acceleration from external_body
    end for
end function

function updata_location(bodies,dt,args)
    if args.m == OPENMP then
        omp parallel for
    end if
    for i = 0 to args.n do
        bodies[i].x <- bodies[i].x + bodies[i].vx * dt
        bodies[i].y <- bodies[i].y + bodies[i].vy * dt
    end for
end function
```

Whenever OPENMP is encountered, parallel processing will be turned on. And the race condition is also specially processed, such as using critical, atomic, or allocating additional memory for each thread and merging data at the end (approaches will be discussed later).

Experiment Environment

The experimental code has been tested on Instance Hub and is able to run. The experimental data to be discussed below was generated by my local computer with an i7-9750H CPU of 6 cores and 12 threads. And when it involves OPENMP, the code is set to open the max thread, which is 12.





Parallelizing over Inner and outer loop

Observing the experimental effects of CPU, OPENMP inner and OPENMP outer, we can find that OPENMP outer is the best. If we take N as a variable and fix the remaining parameters, such as D and the number of iterations.

In the polyline above, the X-axis is the N variable from 1000 to 2000, increasing by 250 each time, and the Y-axis is the time spent in the simulation. The graph on the left is generated when the parameter D is 50 and the number of iterations is 100. The figure on the right keeps the same parameter D at 50, while the number of iterations is 150. (The above chart data is generated in debug mode, but release mode will also be discussed).

We can observe that when increasing N, the simulation time of CPU, OPENMP inner and OPENMP outer operations almost all have a linear increase. And OPENMP outer has the shortest time, followed by CPU, and finally OPENMP inner.

N	D	Iteration	Mode	Time
2000	50	200	CPU	15.382
			OPENMP (outer) Debug	2.625
			OPENMP (outer) Release	1.22

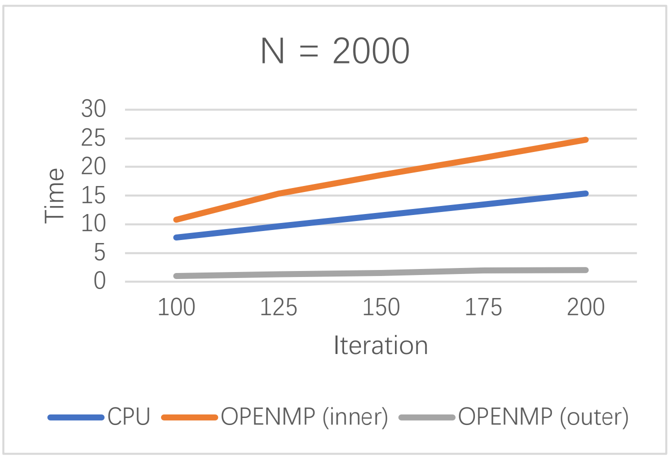
By comparing the above picture with the table, in debug mode, the speed of OPENMP outer is about 6 times faster than that in CPU mode. And in release mode, it can reach an amazing 12 times. This is entirely reasonable, because the number of threads running the simulation is 1 in CPU mode and 12 in OPENMP mode.

When we observe OPENMP inner, we will find that it will be slightly slower than CPU mode. This is mainly because, in order to solve the race condition, I added extra memory for each thread in the code, which led to an increase in the amount of code that needs to be executed. Despite OPENMP inner use of multi-threading, its execution efficiency is not high due to the waiting barrier and also the need for extra code execution.

The inner loop is mainly implemented in the calculate_single_body_acceleration function, and its pseudo code is roughly as follows,

```
function calculate_single_body_acceleration(bodies,index,args)
    target_body <- bodies[index]

    thread_num <- omp_get_max_threads()
    #pragma omp parallel for firstprivate(thread_num)
    for i = 0 to thread_num do
        local_acc[i] = {0,0}
    end for

    #pragma omp parallel for shared(index)
    for i = 0 to args.n do
        num <- omp_get_thread_num()
        external_body <- bodies[i]
        local_acc[num].x <- local_acc[num].x + acceleration from external_body
        local_acc[num].y <- local_acc[num].y + acceleration from external_body
    end for

    acceleration <- {0.0}
    #pragma omp master
    for i = 0 to thread_num do
        acceleration.x <- acceleration.x + local_acc[num].x
        acceleration.y <- acceleration.y + local_acc[num].y
    end for
end function
```

The above discussion was observed under the condition that N is a variable. By observing the left and right figures in the above figure, when the number of iterations increased from 100 to 150, the time consumed in CPU mode also increased by 50%, while OPENMP outer The data is basically unchanged. For a more detailed observation, we will set the fixed N value to 2000, D to 50, and set the number of iterations as a variable, from 100 to 200, each increment of 25.

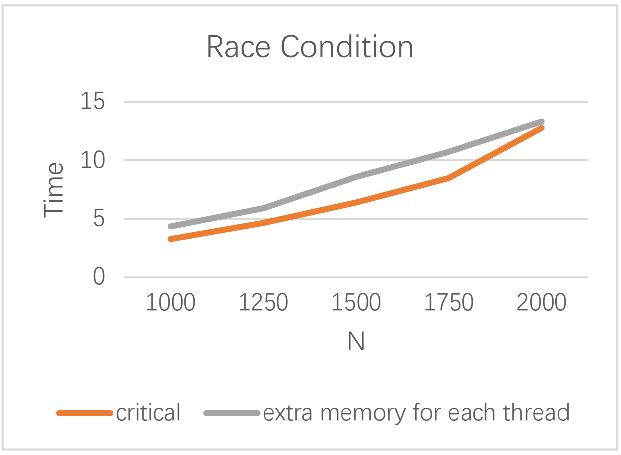


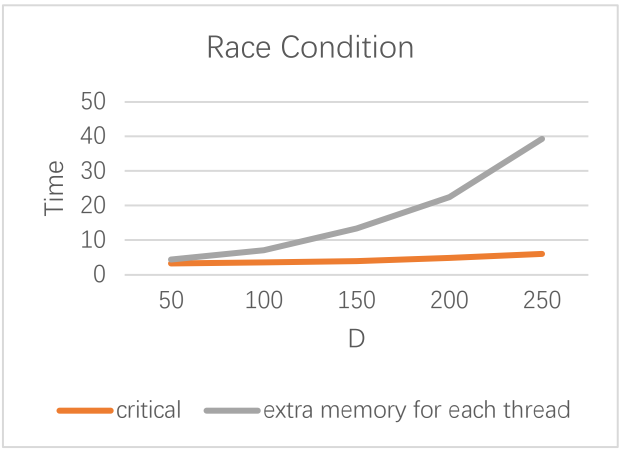
We can observe that whether increasing N or the number of iterations has a great influence on the CPU mode. However, for OPENMP outer, unlike increasing N, increasing the number of iterations has little effect on its time consumption. This is mainly because increasing N can lead to a greater increase in calculation.

Race Conditions

There are three common methods to avoid Race Conditions. Two of them do not need to change the code structure, they are atomic and critical. Whenever the specified area needs to be written, it will wait until no other thread is executing this data. Another way to avoid Race Conditions requires changes to the code structure. It is mainly done by allocating additional memory to each thread and finally assemble the data from all threads.

In my code, for faster code execution, the heat-map is only updated when the GUI window is opened. When the -i parameter is present, the heat-map will not be updated. But in order to quantify various methods to solve race condition, the code has been modified. At the same time, each iteration will update the heat-map. In the following figure, the N and D values will be changed separately to observe the performance of different methods.





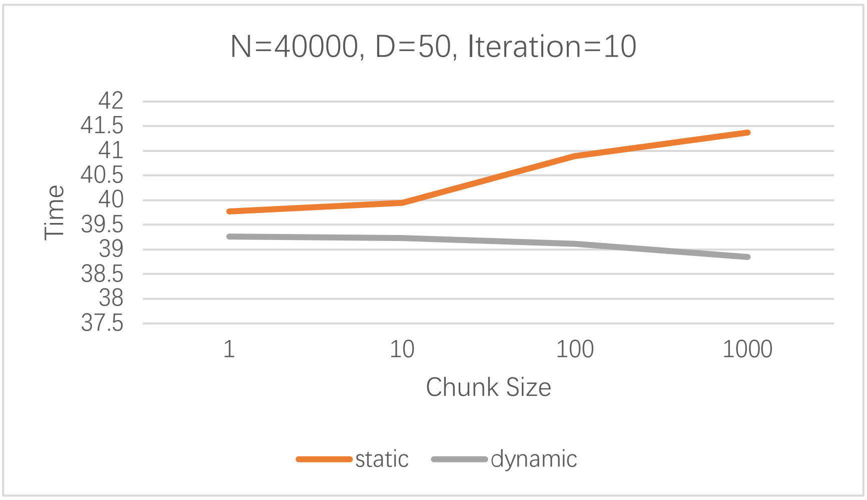
According to the left figure, when the value of D is fixed at 50, and the value of N is continuously increased, the time-consuming increases linearly. This is because when calculating the heat map, more bodies need to be traversed, so it takes more time. In comparison, the critical method is slightly better than the method of allocating additional memory to each thread, because the method of allocating additional memory also needs to sort the data of each thread together at the end, and this part of the time cannot be ignored.

According to the figure on the right, when the value of D is fixed at 50, and the value of N is continuously increased. For the critical method, the time consumption increases linearly. This is because when calculating the heat map, the heat map to be traversed is larger, so it takes more time. In comparison, for the method of allocating additional memory to each thread, the additionally allocated memory is n_thread * D * D. Therefore, when D increases, the memory consumption also increases by a power of two.

All in all, critical may be more suitable for updating the heat map.

Improvement in Stimulation

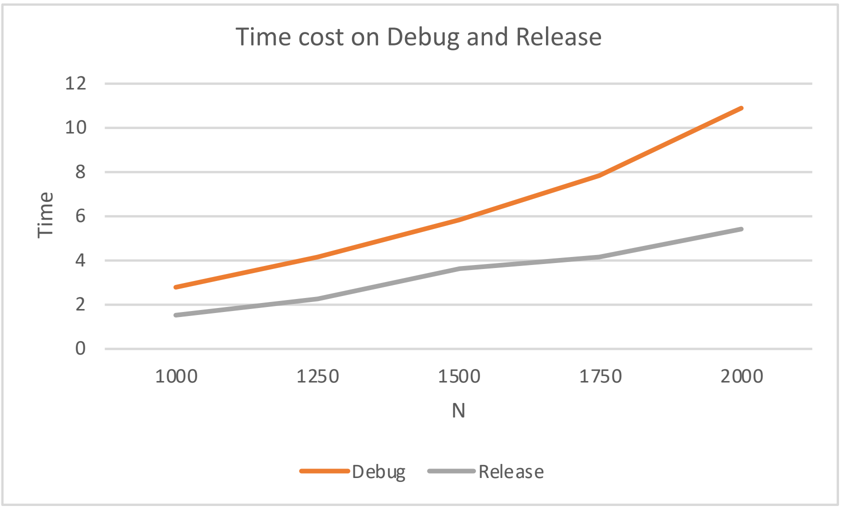
1. Schedule: Different schedules have different effects on the execution time of tasks. However, this effect is limited.



As shown in the figure above, under the same chunk size, the use of dynamic can accelerate the efficiency of task execution. At the same time, if you continue to increase the chunk size, static performance will become worse and worse, while dynamic performance will become better.

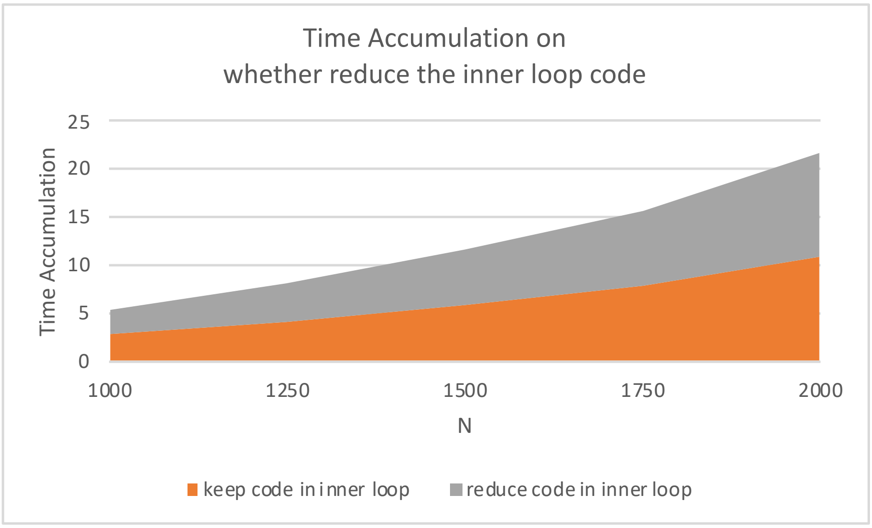
This situation is entirely reasonable. For static, different threads may have different execution speeds. In the case of a larger chunk size, some threads will be executed in advance and are idle. Therefore, the execution efficiency of the code becomes lower. However, for dynamic, the smaller chunk size can ensure the full use of threads and reduce the idle state of threads. But this will also cause the computer to detect the status of each thread too frequently. Such frequent operations will make the execution efficiency of the code lower.

2. Release and Debug: By changing the code from debug to release, the execution efficiency can also be greatly improved.



It can be seen from the above figure that the execution efficiency of Release is about twice that of Debug.

3. Reduce code in the inner loop: When trying to reduce the code in the inner loop, such as trying to move the part to the outer loop, the execution efficiency of the code can also be improved.



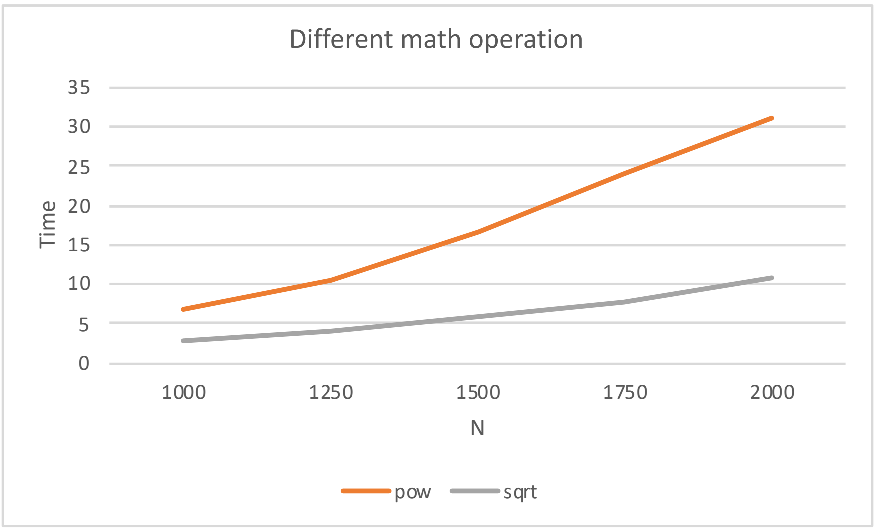
Since this method is more limited in improving performance, it is more convenient to observe the time accumulation graph here. It can be observed from the above figure that in the case of OPENMP, moving a line of code in the inner loop to the outer loop can improve efficiency. This is because the code in the inner loop needs to be executed n_outer * n_inner times. However, when a line of code is reasonably moved to the outer loop, the program executes n_outer * (n_inner -1) times less. Thus efficiency will become faster.

Other interesting Aspects

1. sqrt is faster than pow: Part of the code needs to calculate the acceleration between the stars. And this code needs to be solved according to the formula:

$$\vec{F_i} = Gm_i \sum_{1 \le j \le N} \frac{m_j(\vec{x_j} - \vec{x_i})}{\left(\left\|\vec{x_j} - \vec{x_i}\right\|^2 + \varepsilon^2\right)^{\frac{3}{2}}}$$

The denominator can be calculated simply by pow (x, 3/2), but when sqrt (x) * x is used instead, the speed becomes surprisingly fast. Compared with the method using pow, the formula calculated by sqrt can increase the code speed by 3 times.



2. Run is faster than debugging: By chance, I found that using the terminal to execute code is faster than F5 in Visual Studio 2019, which is about 2 seconds faster. Later, I noticed that F5 is a shortcut key for debugging. However, in a terminal, the code is directly run. Although this is not an improvement in code efficiency, it also made me realize that it is important to distinguish between debug and run.

Experimental Data Table

Benchmark Results table

N	D	Mode	Iteration	Time
1000	50	CPU	100	1.912
			125	2.406
			150	2.886
			175	3.336
			200	3.857
		OPENMP (inner)	100	2.934
			125	3.776
			150	4.485
			175	5.243
			200	5.852
		OPENMP (outer)	100	0.274
			125	0.352
			150	0.441
			175	0.52
			200	0.555
1250	50	CPU	100	3.12
			125	3.765
			150	4.484
			175	5.25
			200	5.993
		OPENMP (inner)	100	4.477
			125	5.841
			150	7.784
			175	9.43
			200	10.245
		OPENMP (outer)	100	0.445
			125	0.555
			150	0.672
			175	0.756
			200	0.849
1500	50	CPU	100	4.322
			125	5.383
			150	6.477
			175	7.569
			200	8.651
		OPENMP (inner)	100	6.351
			125	8.522
			150	10.876
			175	13.183
			200	14.503
		OPENMP (outer)	100	0.611
			125	0.774
			150	0.955
			175	1.136
			200	1.313

1750	50	CPU	100	5.909
			125	7.357
			150	8.853
			175	10.336
			200	11.747
		OPENMP (inner)	100	8.512
			125	11.698
			150	14.451
			175	16.88
			200	19.264
		OPENMP (outer)	100	0.894
			125	1.121
			150	1.344
			175	1.537
			200	1.695
2000	50	CPU	100	7.704
			125	9.667
			150	11.568
			175	13.495
			200	15.382
		OPENMP (inner)	100	10.809
			125	15.357
			150	18.627
			175	21.636
			200	24.757
		OPENMP (outer)	100	0.987
			125	1.26
			150	1.547
			175	1.907
			200	2.02

Schedule Results table

N	D	Mode	Iteration	Schedule	chunk	Time
40000	50	OPENMP	10	static	1	39.771
					10	39.948
					100	40.894
					1000	41.37
				dynamic	1	39.262
					10	39.235
					100	39.121
					1000	38.848

Debug and Release Results table

D	Mode	Iteration	Configure	N	Time
50	OPENMP	1000	Debug	1000	2.718
				1250	4.137
				1500	5.842
				1750	7.815
				2000	10.832
			Release	1000	1.46
				1250	2.21
				1500	3.56
				1750	4.173
				2000	5.365

Race Condition Results table

D	Mode	Iteration	Race	N	Time
50	OPENMP	1000	critical	1000	3.277
				1250	4.672
				1500	6.43
				1750	8.519
				2000	12.783
			extra memory for each thread	1000	4.356
				1250	5.946
				1500	8.583
				1750	10.713
				2000	13.333
N	**Mode**	**Iteration**	**Race**	**D**	**Time**
1000	OPENMP	1000	critical	50	3.308
				100	3.577
				150	3.912
				200	4.967
				250	6.071
			extra memory for each thread	50	4.447
				100	7.217
				150	13.401
				200	22.445
				250	39.32

Math: sqrt and pow Results table

D	Mode	Iteration	Function	N	Time
50	OPENMP	1000	pow	1000	6.747
				1250	10.562
				1500	16.52
				1750	24.004
				2000	31.206
			sqrt	1000	2.718
				1250	4.137
				1500	5.842
				1750	7.815
				2000	10.832