**Assignment 2(OpenMP Tasking) Documentation**

Md Rafiqul Islam - 12123971

Technology used:

C++, OpenMP

How to run program:

1. Compile .cpp files:

* g++ a2-sequential.cpp -o a2-sequential.exe (sequential code)
* g++ -fopenmp a2-openmp.cpp -o a2-openmp.exe (for OpenMP)

1. Run program:

* ./a2-sequential.exe
* ./a2-openmp.exe

Code Structure:

The program consists of three different C++ files: a2-helpers.hpp, a2-sequential.cpp and a2-openmp.cpp.

* a2-helpers.hpp – This is header file which contains the defined data structures of the program. It has Image data structure, gradient data structure, and some functions called interpolate\_rgb\_color, colorize and get\_2d\_kernel. Here, interpolate\_rgb\_color sets the image color where it needs. Colorize function takes care of gradient color of the picture. And get\_2d\_kernel() function gives the kernel which one used for image filtering.
* a2-sequential.cpp – With the help of this c plus plus file we generate our Mandelbrot set and, we print image of it and filtered it. First, a set of random gradient values adjusted for our Mandelbrot algorithm. Here we have Mandelbrot\_kernel() function which check if the given point is a member of the Mandelbrot set or not, using z = z \* z + c this formula. If the point was a member of Mandelbrot set, then we colored then point with a color otherwise we used different color to identify the point. In our case, we just used maximum 2048 iterations.

Another fuction mendelbrot() is takes as parameter an image and ratio then it takes image’s height, width, and channels then for every pixel of image it calls Mandelbrot\_kernel() function. Then Mandelbrot\_kernel() checks if the given pixel is a member of Mandelbrot set or not. If it was a member then it colored the pixel in black otherwise different gradient color.

There is another function in this file called convolution\_2d. This function is used for applying Gaussian filter for this Mandelbrot set image. Note that here all works had been done in a single processor. That’s why this approach is slower.

* a2-openmp.cpp – This c plus plus file contains parallel code of Mandelbrot and Convolution part. Here inside mandelbrot() function the parallelization has happened. First of all, the number of threads for used parallel execution has been set using omp\_set\_num\_threads(num\_of\_thread\_used) where num\_of\_thread\_used = 1,2,4,8,16. The performance of using different threads are given below in the Table-2.

Mandelbrot part: Inside Mandelbrot () function there are two nested for loops. One of them has maximum iteration image height and another one has image width. This part of the code can be ideal choice for parallelization. This part of code has been parallelized using parallel for in openMP like this:

**#pragma omp parallel for schedule(dynamic) default(none) private(i,j,pixel,c) shared(h, w, channels, ratio, image) reduction (+:pixels\_inside) collapse(2)**

Here the nested loop has been parallelized with #pragma omp parallel. OpenMP parallel loop is worksharing constructs that take an amount of work and distribute it over the available threads in a parallel region, created with the parallel pragma. Here one important thing is #pragma mop parallel for didn’t create team of threads, it takes the team of threads that is active and divide the loop iterations over them. This means that the omp for directive needs to be inside a parallel region. Here also the same things happened. #pragma omp parallel for divide the for loop depends on how much threads was using (1,2,4,8,16). After splitting the loop iteration, the work has been done in parallelly and then the split works also has been joined by parallel for. Parallel for takes care of splitting total loop iterations and joining them, this the one advantage of using parallel loop.

When the parallel loop was used, there were also some addition clauses used. First, the default variable data scope was stopped by using default(none). Then manually with the help of private() and shared() clause the variable’s data scope were defined. Those variables what needs to be updated during every iteration these were defined as private and those variables which don’t need to be updated are defined as shared.

Another important clause is reduction() which used here. This clause here because with building of mandelbrot here also the number of pixels were counting inside the loop. This is one kind of summation operation. The reduction() clause works good for this kind of operation. This clause also saves the code from the data race condition.

In general, the more work there is to divide over several threads, the more efficient the parallelization will be. In the context of parallel loops, it is possible to increase the amount of work by parallelizing all levels of loops instead of just the outer one. As it was told that the 2 nested loop were parallelized here that’s why the collapse(2) clause was used here. In this case all N\*N iterations are independent but generally omp for directive will only parallelize one level so with the help of collapse(2) the 2-level parallelization had been done which is more efficient.

When parallelization has started different thread start work independently and these works are not always same. Some works are bigger, and some are smaller. Sometimes some threads take long time, and some threads takes less time. That’s why for final output the main worker must wait for other threads result. This is one kind of latency of program. OpenMP has a solution for this kind of situation. There is a clause called schedule() which can take two types of parameters, one is static and another one is dynamic. OpenMP in default used static schedule. Static schedule is that kind of schedule that main worker must wait until all works ends. On the other hand, dynamic schedule is something else. It dynamically shares work. If one threads completed all his work, then that thread takes more from other busy threads. This way the work sharing has been done efficiently and program got speedup.

Convolution part: This part looks more complicated, but it is not. There are more than 5 nested loops inside the convolution\_2d() function. But interesting thing is we don’t need to parallelize all these loops. Here the most iterated loops were selected. Here we would like to parallelize the image height loop and image width loop which are inside nsteps and channels loop. Like before at first the number of threads were set for parallelizing using omp\_set\_num\_threads(num\_of\_thread\_used) where num\_of\_thread\_used = 1,2,4,8,16.

**#pragma omp parallel for default(none) shared(h,w,kernel,displ,ch,src,dst) collapse(2)**

By using #pragma omp parallel for here the image height and image width loop were parallelized. The default (none) clause was use because we want all the data scope access the closest memory location. This will increase performance of the code. After defining default(none) clause, the shared () clause was used for data scoping. As we can see form the code the loop variables and other variable which need to be updated in every iteration are all declared inside the loops. That’s why these variables are already in private data scope. So, we don’t need to define them again. So, we just need to define these variables which were pre declared outside the loop and used inside the loops as shared(). For convolution part h, w, kernel, displ, ch, src, dst etc. variables were defined as shared.

In this part we also interested to parallelize 2 loops that’s why the collapse() clause was used with parameter 2 like this collapse(2). Collapse clause parallelized height and width loop nicely. The performance is also good (see Table:2).

Tables:

Table-1 Sequential:

|  |  |  |  |
| --- | --- | --- | --- |
| Mandelbrot time (s) | Convolution time (s) | Total time (s) | Total Mandelbrot pixels |
| 20.9354 | 51.0527 | 71.9881 | 1478025 |

Table-2 Parallel OpenMP (parallel for):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Threads | Mandelbrot time (s) | MT. Speedup | Convolution time (s) | CT. Speedup | Total time (s) | Total Mandelbrot pixels |
| 1 | 20.8479 | 1.0042 | 52.612 | 0.970363 | 73.4599 | 1478025 |
| 2 | 10.651 | 1.96558 | 26.5727 | 1.92125 | 37.2237 | 1478025 |
| 4 | 5.39954 | 3.87726 | 13.6441 | 3.74175 | 19.0436 | 1478025 |
| 8 | 2.97233 | 7.04344 | 7.70936 | 6.62217 | 10.6817 | 1478025 |
| 16 | 1.63255 | 12.8237 | 4.22842 | 12.0737 | 5.86097 | 1478025 |

Table-3 Parallel OpenMP (omp task):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Threads | Mandelbrot time (s) | MT. Speedup | Convolution time (s) | CT. Speedup | Total time (s) | Total Mandelbrot pixels |
| 1 | 20.5927 | 1.01664 | 52.7829 | 0.967221 | 73.3756 | 1478025 |
| 2 | 10.4583 | 2.0018 | 26.6797 | 1.91354 | 37.138 | 1478025 |
| 4 | 5.38178 | 3.89005 | 13.6494 | 3.7403 | 19.0311 | 1478025 |
| 8 | 2.97795 | 7.03015 | 7.75953 | 6.57935 | 10.7375 | 1478025 |
| 16 | 1.79432 | 11.6676 | 4.24192 | 12.0353 | 6.03624 | 1478025 |

Speedup Graph and Description:

Fig-1: Mandelbrot speed up curve.

Fig-2: Convolution speed up curve.

Graph Description:

Two different version were tried to parallelize the code. First version was only using ‘omp task’ and another version is ‘parallel for loop’. Both version of code compiled and run in the ALMA. The result data for ‘parallel for loop’ version and ‘omp task’ version shown in Fig-1 and Fig-2. Both figures show the comparison between ‘parallel for loop’ version and the ‘omp task’ version of the code.

There are two figures which represents the speed up data of parallelize code. Fig-1 represents the Mandelbrot part speed up curves and Fig-2 represents Convolution part speed up curves. In the graph x-axis contains thread numbers and y-axis contains speedup values.

If we see Fig-1, for Mandelbrot part there are two different were tried to parallelize the code. One version is ‘parallel for’, and another version is ‘omp task’. From it is clearly seen that ‘parallel for’ version performs much better than ‘omp task’ version. For parallelizing different threads were used, they are 1, 2, 4, 8 and 16. When the threads number are increasing for parallel for version of code, the speedup value is also increasing, which means the parallelization is working. On the other hand, in ‘omp task’ version we can see that this performs bad, even it is worse than sequential version.

Similarly, For Convolution part in Fig-2 we also can see that when the threads number are increasing the speedup value of ‘parallel for’ version also increasing. For ‘parallel for’ version parallelization working fine. But for ‘omp task’ version the performance is poor. There is a huge difference between ‘parallel for’ version and ‘omp task’ version, ‘parallel for’ version always performs better.

Discussions:

* Performance differences between omp task and parallel for loop versions: There are huge performance difference between these two versions. ‘Parallel for loop’ version divide the total loop iteration into some chunks (depends on how much processor using), then these chunks of iterations execute parallelly. This way code executes faster than before which means it boost performance. But when we tried ‘omp task’ version it also created multiple tasks for parallelly execution but these during tasking we must take care of synchronization. During synchronization which task has finished the work it has to wait for other task which are not finished yet (in our case critical section does that). But using omp tasking we didn’t get better performance. It took long time to execute the program. Maybe there are also some better tasking methods, but we couldn’t find that. After all, what we got is that our ‘omp parallel loop’ version done really good job. Its performance is way better than the sequential and ‘omp task’ version. When we used 2 threads it gives almost 2x performance for both Mandelbrot and convolution part. This way for 16 threads it gives more than 12x performance.
* Task granularity (small vs big tasks): From working experience it has seen that yes task granularity matters. When tasks are small then the parallel execution can’t perform well because these small tasks divided into threads, but these tasks are too small and also there are a large number of task generated and all these task couldn’t finish in the same time. Then one thread has to wait for other. After all small tasing takes more time. But if we divide large work into some chunks of iterations then it gives better result. For example, we parallelize for loop using #pragma omp parallel for. This omp line divide the loop into some small chunks depends on thread number, like if we have total iteration 1024 and we use 2 threads for parallelization then chunks size will be 1024/2 = 512 iterations. The more thread we use then chunks size will be smaller and start executing at the same time parallelly. The schedule clause done this job more efficiently. It helps the threads to properly use time.
* Distributed the work:

Thread 3

Thread 2

Thread 1

Thread 0

Result

Fig-3: Distribution of work in parallel for

Here we discuss about work distribution for ‘parallel for’ version. In Fig-3 there is an example of work distribution for ‘parallel for loop’ using 4 threads. The work division depends on how many threads used for parallelization. If 2 threads were used, then loop would be divided by two parts. Similarly, if 16 threads were used then loop would be divided into 16 parts. For example, suppose we want to parallelize Mandelbrot part which has multiple for loops. The outer loop has total iteration is 1536, using 4 threads. So first, we just set thread number for parallel execution. This line ‘omp\_set\_num\_threads(4)’ ensure that out code will use 4 threads. Then ‘#pragma omp parallel for’ will divide the whole loop into 4 parts. Each part of the work will take 1536/4 iterations. And then it will start executing parallelly. After execution had finished the pragma will auto gathered the parallelly executed result and will give a final output.

* Differences in speedup: The speedup has been measured for both version ‘parallel for’ and ‘omp task’ version. For ‘parallel for’ version using 1 thread almost give same speedup as sequential code but for 2,4,8,16 threads the scenario is different. After using one thread when the thread number is increasing the speedup also increasing. The best speedup so far, we got is ~12.85 for 16 threads in ‘parallel for’ version. When using many threads, the work divided into these threads and execute code faster which means that the more threads we use for parallel execution the more speedup we will get. But on the other hand, the speedup for ‘omp task’ version was poor, sometimes it worse than sequential version. To avoid data racing condition, we had to use critical section, but that section drops the performance of the code.
* Differences in speedup I observed with different clauses: Yes, I observed different speed up in one case with different clauses. When I try to parallelize Mandelbrot part of the code using ‘#pragma omp parallel for’ I saw some differences in speedup when using different clauses. In first try I just used ‘#pragma omp parallel for’ here by default the schedule was static in that time I got some speed up but not much, like for 16 threads I got only 5x speedup. But after adding dynamic scheduling using ‘#pragma omp parallel for schedule(dynamic)’, I got more speed up. It was around 12.8x which is great.
* Interesting findings: There are some interesting things I found during parallelization. In first try when I used omp task inside #pragma omp parallel the code was superfast but that time I realize that the output of total pixel count is incorrect. After that I try to find the reason behind it. After many research I found some issues. Here, at first time I didn’t use any critical section protection. That’s why when the threads running, they may wish to use shared variables and data race condition happened. To resolve race condition, I used a critical section inside the nested loop. After adding critical section, the code was giving correct output as I expected but unfortunately, the performance was too poor, even worse than sequential code.

After that I was trying to find another good solution. Then I tried parallel for loop version like this #pragma omp parallel for, this time also the execution speed most likely as exception but the same problem arises. The pixel counts still giving wrong result. Then with ‘parallel for’ I used reduction () clause which helps me a lot to getting correct output. Now code is faster but not too fast. After some research I found that schedule() clause. Normally omp used static schedule which can’t use time efficiently because when the parallel work is running some threads completed their works early and some threads still working that time. These ways the time-consuming increase in the code. But fortunately, omp has dynamic schedule. By using dynamic schedule this problem can be resolved. When dynamic schedule was used the threads who finished his works, take extra work from other threads, and execute code faster. By using schedule() clause I got better performance.