# 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)`

2. 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 etc. Here, interpolate_rgb_color sets the image color where we need. colorize() function takes care of gradient color of the picture. And get_2d_kernel() function gives the kernel which one is used for image filtering.

- a2-sequential.cpp – With the help of this c plus plus file we generate our Mandelbrot set sequentially 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 that point with black color otherwise used different gradient color to separate the point. In our case, we just used maximum 2048 iterations.

  Another function 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() function do his job as discussed before. At the same time, it do its job for 3 channels (r, g, b) which is last part of the nested loop.

  There is another function in this file called convolution_2d. This function is used for applying Gaussian filter into this Mandelbrot 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. First, a global integer variable 'num_of_thread_used' was declared with default value 1. Then the number of threads to use for parallel execution has been set using omp_set_num_threads(num_of_thread_used) where omp_set_num_threads () is a omp function which sets the number of threads in upcoming parallel regions, unless overridden by a num_threads clause and num_of_thread_used = 1, 2, 4, 8, 16. The performance of using different threads for different versions are given below in the Table-2, 3, 4. There are many versions of parallel code has been tested on ALMA but not all of them were performed well. Here among them only three versions are described here. Note that, every version of the code has tested multiple times in ALMA and every time it was giving almost same results.

  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. We choose this part of code to parallel in openMP. The two different versions of Mandelbrot part parallelization described below:

  - Version 1 (omp task): Our first version was 'omp task' version. We tried to parallelize the code using only 'omp task'. There is a sample structure of this version of code shown below:

```
omp_set_num_threads(num_of_thread_used);
#pragma omp parallel default(none)private(i,j,pixel,c)
shared(h, w, channels, ratio, image, pixels_inside)
#pragma omp single
for (j = 0; j < h; j++)
      #pragma omp task
      for (i = 0; i < w; i++)
            …….
            #pragma omp atomic
            pixels_inside++;
      ……
      ……
```

    First, we set the number of threads to use for parallel execution. Then by using '#pagma omp parallel' fork additional threads to carry out the work enclosed in the construct in parallel. The original thread has denoted as master thread. When the '#pagma omp parallel' was used, there were also some additional clauses were used. First, the default variable data scope was desabled by using default(none) clause. Then manually with the help of private() and shared() clause the variable's data scope were defined.

    '#pragma omp single' is single construct. With the help of single construct one thread generates only the tasks and all other threads execute the tasks as they become available. Those all generated tasks runs parallelly that's why if we don't use any synchronization method it will fall in data race condition. So, for synchronization we used '#pragma omp atomic'. The omp atomic directive allows access of a specific memory location atomically. It ensures that race conditions are

avoided through direct control of concurrent threads that might read or write to or from the particular memory location. We used omp atomic directive only for pixels_count.

- Version 2 (omp taskloop): After trying first version, we tried to parallelize the code using 'omp taskloop'. There is a sample structure of this version of code shown below:

```
omp_set_num_threads(num_of_thread_used);
#pragma omp parallel default(none) private(i, j, pixel, c)
shared(h, w, channels, ratio, image, pixels_inside)
#pragma omp single
#pragma omp taskloop collapse(2)
num_tasks(omp_get_num_threads()*40) grainsize(h*40) nogroup
    for (j = 0; j < h; j++)
          for (i = 0; i < w; i++)
                …….
                #pragma omp atomic
                pixels_inside++;
          ……
          ……
```

Taskloop is not a worksharing construct (like OpenMP for) that's we need to run it inside a single region unless we want to perform the loop multiple times. So that we have done this like below:

Like first version, we set the number of threads to use for parallel execution. Then by using '#pagma omp parallel' fork additional threads to carry out the work enclosed in the construct in parallel. The original thread has denoted as master thread. When the '#pagma omp parallel' was used, there were also some additional clauses were used. First, the default variable data scope was desabled by using default(none) clause. Then manually with the help of private() and shared() clause the variable's data scope were defined.

'#pragma omp single' is single construct. With the help of single construct one thread generates only the tasks and all other threads execute the tasks as they become available. Then we used '#pragma omp taskloop' construct for parallelization. The taskloop construct a task generating construct. When a thread encounters a taskloop construct, the construct partitions the iterations of the associated loops into explicit tasks for parallel execution. Here also to avoid data race condition we used omp atomic directive. With taskloop we also used some additional clauses like 'collapse()', 'num_tasks()', grainsize(), nogroup etc. we used collapse(2) because we want to parallelize two loops, num_task() for total number of task generated during parallelization, grainsize() for the number of logical loop iterations assigned to each generated task.

- Version 3 (omp parallel for): After 2nd version we also tried 'omp parallel for'. Here we tried to parallelize the code using only 'omp parallel for'. There is a sample structure of this version of code shown below:

```
omp_set_num_threads(num_of_thread_used);
#pragma omp parallel for schedule(dynamic) default(none)
private(i, j, pixel, c) shared(h, w, channels, ratio,
image) reduction (+:pixels_inside) collapse(2)
for (j = 0; j < h; j++)
      for (i = 0; i < w; i++)
            ……
            pixels_inside++;
      ……
      ……
```

Here the two nested loop has been parallelized with '#pragma omp parallel for'. OpenMP 'parallel for' is work-sharing 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 thing to note that is '#pragma omp parallel for' didn't create team of threads, it takes the team of threads that is active and divide the loop iterations over them. That means that the omp for directive needs to be inside a parallel region. In our case also the same things happened. '#pragma omp parallel for' divide the for loop depends on how much threads were using (1, 2, 4, 8, 16).

After splitting the loop iteration, the work was start executing in parallel and then the split works also had been joined after all thread finished their job. Parallel for takes care of splitting total loop iterations and joining them, this the one advantage of using parallel loop for programmer.

When the 'omp parallel for' was used, there were also some addition clauses used. First, the default variable data scope was disabled by using default(none) clause. 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 but used inside the loop for other reason are defined as shared. For private variables the different threads make their own copy of private variables for use and at last all threads add them up.

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

In general, the more work split 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. In this case all N*N iterations are independent but generally 'omp parallel 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 and faster. 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 which are not finished their work. This is time consuming which affect the parallel program performance. 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 wok from other busy threads. This way the work sharing has been done efficiently and program performs better.

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. We are interested to parallelize the large seize loops. Here we would like to parallelize the image height loop and image width loop which are inside nsteps and channels loop. The two different versions of Convolution part parallelization described below:

- Version 1 (omp task): Our first version was 'omp task' version. We tried to parallelize the code using only 'omp task'. There is a sample structure of this version of code for convolution part shown below:

```
for (int step = 0; step < nsteps; step++)
      for (int ch = 0; ch < channels; ch++)
            omp_set_num_threads(num_of_thread_used);
            #pragma omp parallel default(none) shared(h, w,
            kernel, displ, ch, src, dst)
            #pragma omp single
            for (int i = 0; i < h; i++)
                  #pragma omp task
                  for (int j = 0; j < w; j++)
                        …….
                  ……
                  ……
```

Tasking process is similar as described in the Mandelbrot part. First using default(none) clause default data scope sharing were disabled then with help of shared() clause the data scope of variables were defined. Here we didn't need any private clause because those variables which need to be update inside loop are already private because they were declared inside pragma command. '#pragma omp single' creates all the task using single threads and other threads are executing

these created tasks. Parallelized loops were run independently so we didn't need any synchronization section. The performance of code for different threads shown in Table:2.

- Version 2 (omp taskloop): We have tried another approach to parallelize Mandelbrot part. We tried 'omp taskloop'. There is a sample structure of this version of code for convolution part shown below:

```
for (int step = 0; step < nsteps; step++)
      for (int ch = 0; ch < channels; ch++)
            omp_set_num_threads(num_of_thread_used);
            #pragma omp parallel default(none) shared(h, w,
            kernel, displ, ch, src, dst)
            #pragma omp single
            #pragma omp taskloop collapse(2)
            num_tasks(omp_get_num_threads()*40)
            grainsize(h*40) nogroup
            for (int i = 0; i < h; i++)
                  for (int j = 0; j < w; j++)
                        …….
                  ……..
                  ……..
```

Taskloop process is similar as described in the Mandelbrot part. First using default(none) clause default data scope sharing were disabled then with help of shared() clause the data scope of variables were defined. Here we didn't need any private clause because those variables which need to be update inside loop are already private because they were declared inside pragma command. '#pragma omp single' creates all the task using single threads and other threads are executing these created tasks. Parallelized loops were run independently so we didn't need any synchronization section. Similarly we used collapse(2) clause for parallelize 2 loops, num_task() for total number of task generation and grainsize() for the logical iteration for every task. Code performance shown in Table:3.

- Version 3 (omp parallel for): After version 2, we tried to parallelize the code using only 'omp parallel for'. There is a sample structure of this version of code of convolution part shown below:

```
for (int step = 0; step < nsteps; step++)
      for (int ch = 0; ch < channels; ch++)
            omp_set_num_threads(num_of_thread_used);
            #pragma omp parallel for default(none)
            shared(h, w, kernel, displ, ch, src, dst)
            collapse(2)
            for (int i = 0; i < h; i++)
                  for (int j = 0; j < w; j++)
                        …….
                  ……..
                  ……..
```

Here also like before at first the number of threads were set for parallelizing using runtime function called omp_set_num_threads(num_of_thread_used) where num_of_thread_used = 1, 2, 4, 8, 16.

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:4).

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 (omp task):

| Threads | Mandelbrot time (s) | MT. Speedup | Convolution time (s) | CT. Speedup | Total time (s) | Total Mandelbrot pixels |
|---|---|---|---|---|---|---|
| **1** | 20.6711 | 1.01279 | 52.6656 | 0.969376 | 73.3366 | 1478025 |
| **2** | 10.42 | 2.00916 | 26.5992 | 1.91933 | 37.0191 | 1478025 |
| **4** | 5.31542 | 3.93862 | 13.5537 | 3.76669 | 18.8692 | 1478025 |
| **8** | 2.95793 | 7.07771 | 7.69397 | 6.63542 | 10.6519 | 1478025 |
| **16** | 1.67388 | 12.5071 | 4.15648 | 12.2827 | 5.83036 | 1478025 |

Table-3 Parallel OpenMP (omp taskloop):

| Threads | Mandelbrot time (s) | MT. Speedup | Convolution time (s) | CT. Speedup | Total time (s) | Total Mandelbrot pixels |
|---|---|---|---|---|---|---|
| **1** | 20.8106 | 1.006 | 52.937 | 0.964404 | 73.7476 | 1478025 |
| **2** | 10.7501 | 1.94745 | 26.8192 | 1.90359 | 37.5693 | 1478025 |
| **4** | 5.46529 | 3.83061 | 13.6204 | 3.74826 | 19.0857 | 1478025 |
| **8** | 3.08331 | 6.78991 | 7.72632 | 6.60763 | 10.8096 | 1478025 |
| **16** | 1.73551 | 12.063 | 4.18567 | 12.197 | 5.92118 | 1478025 |

Table-4 Parallel OpenMP (omp 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 |

Speedup Graph and Description:

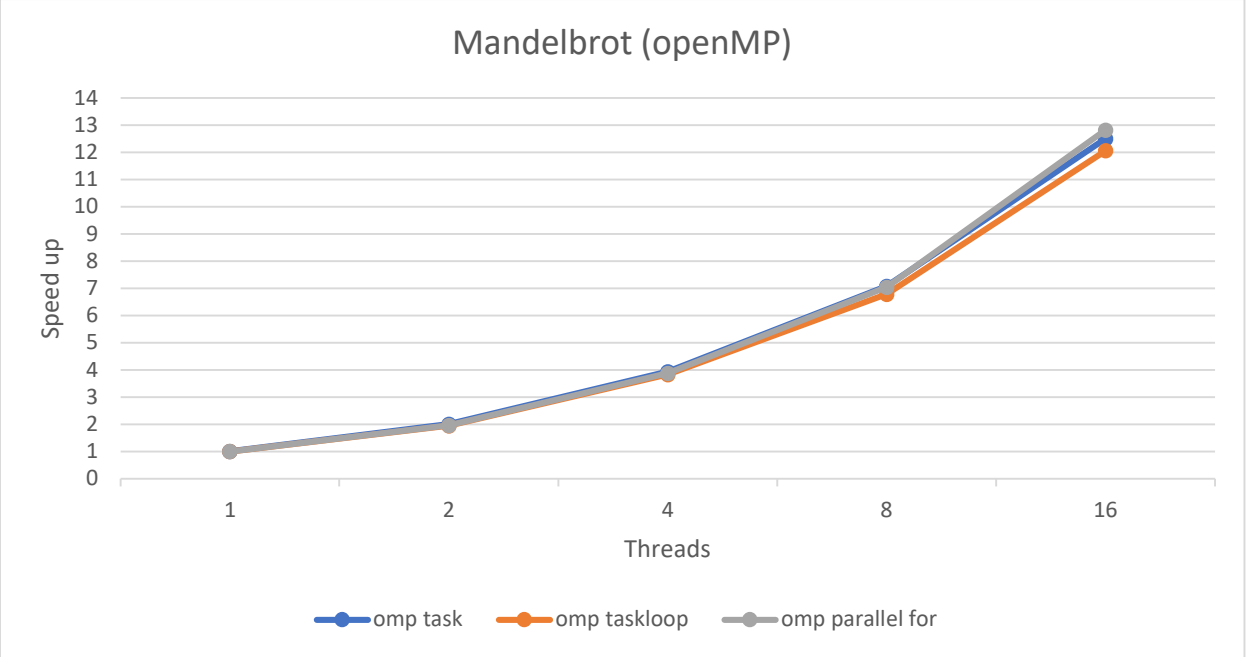


Fig-1: Mandelbrot speed up curve for different versions of code.

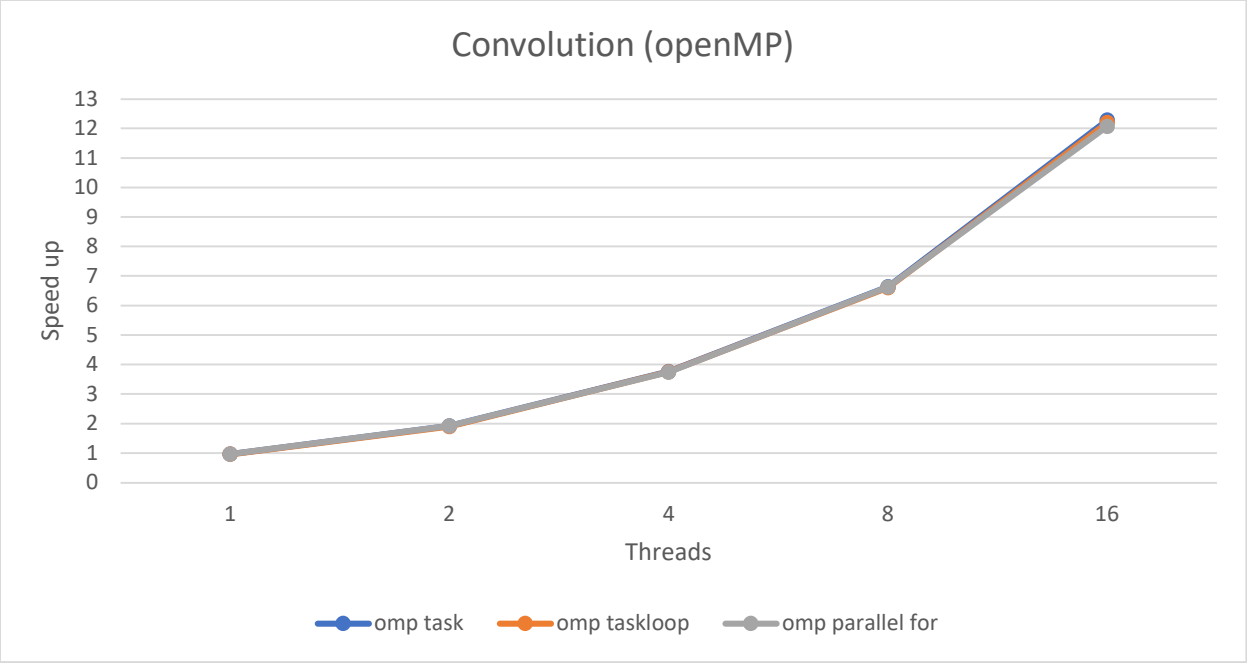


Fig-2: Convolution speed up curve for different versions of code.

Graph Description:

Three different version were tried to parallelize the code. First version was only using 'omp task' and second version was 'omp taskloop' and another version was 'omp parallel for'. All version of code compiled and run in the ALMA multiple times and the result were almost same for each time run. The calculated speed up data for different version of Mandelbrot part shown in Fig-1 and convolution part shown in Fig-2. Both figures show the comparison between 'omp task', 'omp taskloop' and 'omp parallel for' version of the code. In these both speed up curve figures, x-axis contains thread numbers and y-axis contains speedup values.

If we see Fig-1, for Mandelbrot part there are three speed up curve. One curve is for 'omp task', one for 'omp taskloop' and another one for 'omp parallel for'. From these curves is clearly seen that All versions were performed almost same but for 16 threads the 'omp parallel for' version performed slightly better than others and 'omp taskloop' a little slow. For parallelizing different threads were used, they are 1, 2, 4, 8 and 16. When the threads number are increasing for parallel for versions of code, the speedup value is also increasing, which means the parallelization is working.

Similarly, From Convolution part in Fig-2 we also can see that when the threads number are increasing the speedup value of all versions also increasing. In this part all the versions of code perform almost same. There was small difference which is not visible in the curve.

## Discussions:

- Performance differences between omp task, taskloop and parallel for versions: There is almost no performance difference between these versions. 'omp Parallel for' version divide the total loop iteration (depends on how many processor using for parallelization), then these division of

iterations execute parallelly. This way code executes faster than sequential code which means it boost performance. Similarly, when we tried 'omp task' version it created multiple tasks in single threads and other threads parallelly execute these tasks. During tasking we must take care of synchronization if we want data racing condition free code. Here we used omp atomic for synchronization. Taskloop version also need synchronization. In 'omp taskloop' version we set how many tasks will created and how many iterations will run in a task. All versions performance is almost similar but in Mandelbrot part we saw that only for 16 threads the 'omp parallel for' version performs slightly better. In all versions for 16 threads, we got almost around 12x performance.

- Task granularity (small vs big tasks): We also tested some other version of small tasking inside the nested loops. 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 there are a large number of tasks generated and all these tasks couldn't finish at the same time. Then one thread has to wait for other. After all small tasing takes more time. But if we divide large work then it gives better result. For taskloop, it has seen that if we try to create a small number of task or a huge number of tasks for both case the performance dropped. If we balance the task depending on using threads number, then it gives best output.
- Distributed the work/task generated:
  During parallelization with different numbers task were generated but form them the thread*40 and grainsize image height*40 gives better result.
  Task generated for taskloop version:
  1 thread - 1 * 40 = 40
  2 threads - 2 * 40 = 80
  4 threads - 4 * 40 = 160
  8 threads - 8 * 40 = 320
  16 threads - 16 * 40 = 640
  The formula for calculating the number of tasks is number of tasks = ((number of threads) * 40.

  For parallel for version:

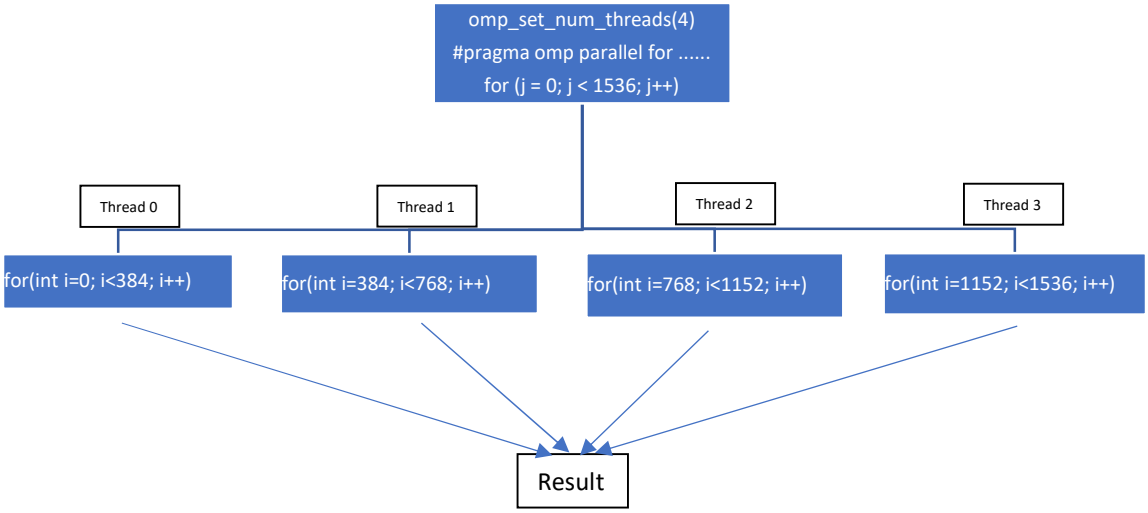


Fig-3: Distribution of work in parallel for

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 calculated for both part Mandelbrot and convolution. Each part has three different version 'omp task', 'omp taskloop' and 'omp parallel for'. For both Mandelbrot and convolution part when all the versions using 1 thread, they almost give same speedup as sequential code and for 2, 4, 8 threads the speed up was increasing. The only exception happened in Mandelbrot part. For 16 threads the 'omp parallel for' performs slightly better than 'omp task' and 'omp taskloop' version. In 'omp parallel for' version for 16 threads we got speed up ~12.8237, in 'omp tasloop' version speedup was ~12.063 Furthermore, in 'omp task' version we got speed up ~12.5071.
- Differences in speedup I observed with different clauses: Yes, I observed different speed up in some cases with different clauses. During parallelize with 'omp taskloop' version first when we used only '#pragma omp taskloop', the performance was not too good but after adding num_task() and grainsize() clause the code gave better performance. Similarly, 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. At 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 around 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 synchronization. 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 omp atomic inside the nested loop. Similarly, for 'omp taskloop' to synchronize we used omp atomic and adding num_task(), grainsize() clauses give better results.

  After that I was trying to find another solution. Then I tried 'omp parallel for' version like this '#pragma omp parallel for', this time also the execution speed most likely as exception but the same problem raised. The pixel counts still giving wrong result. Then with 'omp 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 had been 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.