Parallel Programming HW3: Mandelbrot Set

9961244 EE14 吳德霖

◆ Implementation and Design

1. MPI

The main parallelization is in rows. Divide rows into several tasks so that the whole problem could be parallelized.

- ✧ **Static**

  The master node of rank 0 acts as an I/O dealer as well as one of the computing nodes. If the given number of processors can't divide the row, the master node should finish the residual rows so that the evenly distribution among slaves could be achieved. After the master has done its own job, it would start receiving the computation results one by one in order to maintain statically assigning jobs. If the requested slave hasn't done yet, the master would be kept waiting till the current slave is done. After receiving all data from all other slaves, the master node would then start doing the I/O, if enable tag is presented, it would draw the graph.

- ✧ **Dynamic**

  The master in this time only acts as the I/O dealer, it doesn't attend any computing tasks. The rest of tasks are all slaves. The master would first assign one row to each slave, and then kept itself in situation of waiting to receive computed data. As soon as a slave finishes its row, it would send back the results and a request tag, and then the master would sense such tag to reassign another row to such slave. This is done recursively until no more row is not computed, then the master would trigger a terminate tag and hence the whole process is finished. The master would draw the graph the moment it received the feedback if enable tag is presented.

2. OpenMP

The main mechanism is as same as the MPI version.

- ✧ **Static**

  The master node is assigned to be simultaneously the I/O dealer and a computing node just like the MPI version. Omp for function is applied to conduct the iteratively computing each row, and the clause of which is set to be static. The master would start drawing if enabled if all the computations is done. An omp_barrier is applied at the end of the computing block to synchronize all threads.

- ✧ **Dynamic**

The very same mechanism but the clause is set to be static for the Omp for function block.

3. Hybrid

Rows are parallelized as same as OpenMP version, while the columns is redistributed as the same way MPI does.

- **Static**

  For the row computations, it is done as same as the OpenMP version by the Omp for loop. But here the columns are also divided into several sub blocks to be solved using MPI method. For example if given 2 nodes with 8 processors per node, then it would divide columns into upper half and lower half to different MPI_task, while each task would create 8 threads to deal with the row computations. So, totally 16 cores are used in this case. Several variables in this version would be the both shared and distributed memory simultaneously to be taken advantages of.

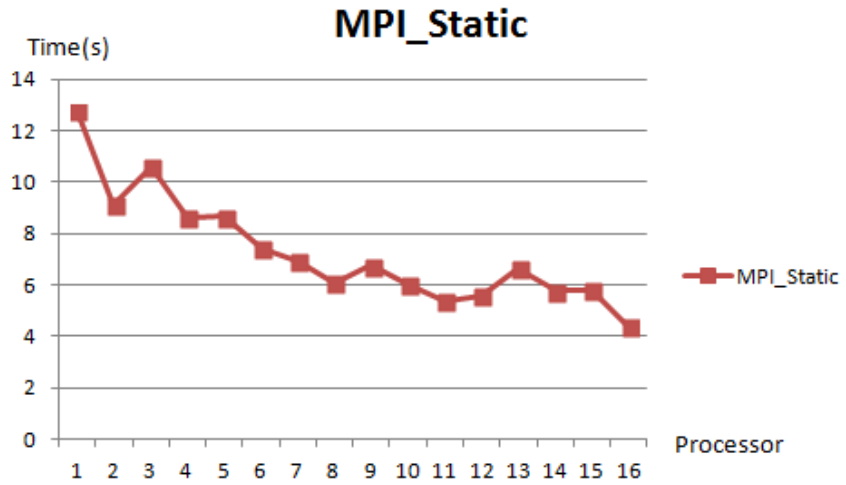
- **Dynamic**

  Just like the MPI_dynamic version, there is a master node doing only the I/O and task assigning. When a slave MPI yask received the assigned row, it would generate several threads to complete such row using omp for, and then send back the computed results. The master would start drawing the output upon receiving required row data.

◆ Experiment & Analysis:

1. Strong Scalability

- **MPI_Static**



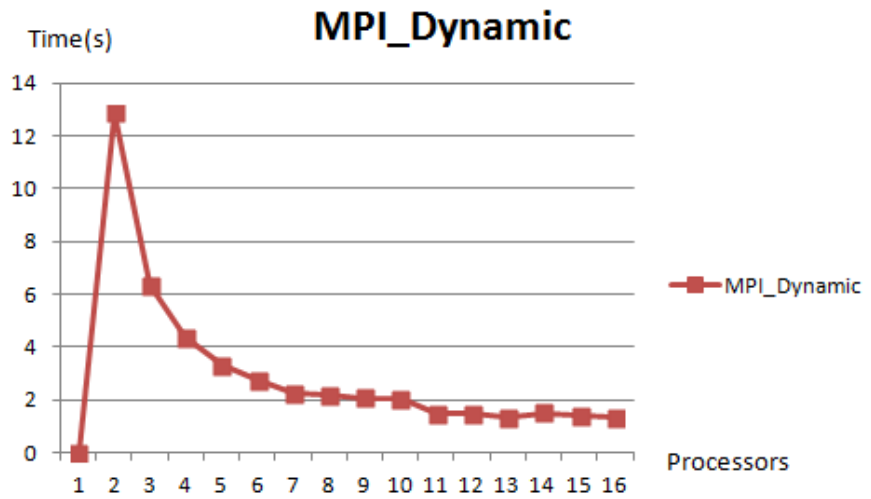
#Procs	1	2	3	4	5	6	7	8
Time(s)	12.75	8.63	10.61	8.63	8.65	7.43	6.95	6.1
#Procs	9	10	11	12	13	14	15	16

Time(s)	6.75	5.98	5.39	5.56	6.63	5.76	5.77	4.36

(X/Y = 400/400)

The above graph implies that the running time does decreased as long as the increment of the number of processors, but rather not so scalable. The main reason causing such result is that my algorithm was simply dividing the original problem into several sub blocks with respect to row numbers. So, first, the load wasn't balanced at all, each rank receives same amount of computations, but the complexity is relatively different. Secondarily, the master rank should wait and receive computed results from other slave ranks statically, that is, one by one from the beginning till the end, which possesses very low efficiency. One can notice that sometimes when the given processors number is odd, the resulting time turned out to be worse than providing less processors. I conclude such thing as that since the problem size is even and not divisible by odd number, there would be one rank dealing with more computations than average, causing the load to be more unbalanced. The communication time here, if taken into account, however, did not contribute to any significant issue. The size of data this time is relatively small when taking the communicating into account, though it does affect slightly to the total running time, but it will not be the main cause of any irrational phenomenon.

✧ **MPI_Dynamic**

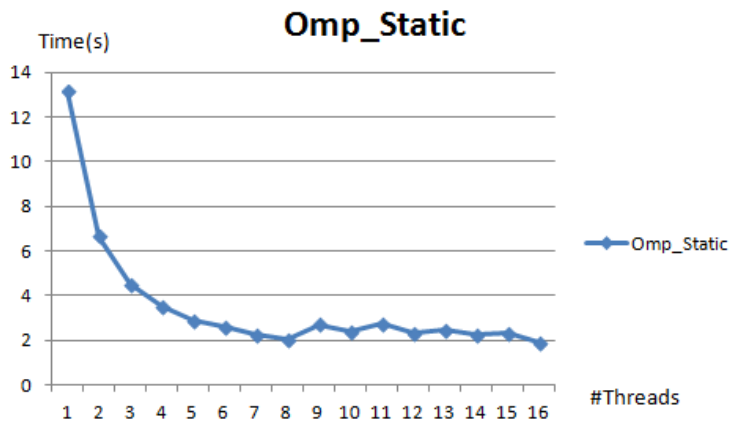


#Procs	1	2	3	4	5	6	7	8
Time(s)	NA	12.87	6.34	4.35	3.34	2.75	2.28	2.21
#Procs	9	10	11	12	13	14	15	16
Time(s)	2.08	2.06	1.49	1.47	1.37	1.42	1.41	1.32

(X/Y = 400/400)

Since I implemented this version with master-slave mechanism, there will be no computation if only given one processors, which is the master who only does I/O. The rest of the graph shows great scalability; the resulting time had been scaled down as the increment of the number of processors, reaching pretty much of saturation while providing more than 8 cores. The computing power does enhanced if providing more cores, but ever since the consuming time is small enough for communication time to have more significant impact on the results with such problem size. It turned out that the consumed time does decreases but barely decreases scaly.

- **OpenMP_Static**

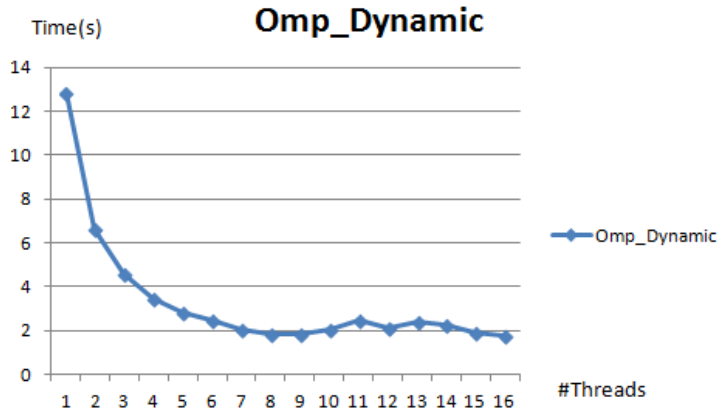


#Procs	1	2	3	4	5	6	7	8
Time(s)	13.15	6.67	4.52	3.54	2.89	2.63	2.25	2.05
#Procs	9	10	11	12	13	14	15	16
Time(s)	2.71	2.42	2.75	2.32	2.49	2.23	2.3	1.89

(X/Y = 400/400)

The graph above shows pretty good scalability, the resulting time until number of threads reaches 8 exhibit great scaling down effect. The reason causing the scalability saturates may be attributed to the static method of implementing the static omp for function. Since some threads may have better performance of computing power to do the next assigned task, but got stuck owing to being have to wait for other threads to finish the job, that is, taking turn to solve the whole problem may cause some traffic jam for those functional threads. And 8 and 16 shows the best performance; I guess they're the best way to divide the total 400 rows.

- **OpenMP_Dynamic**

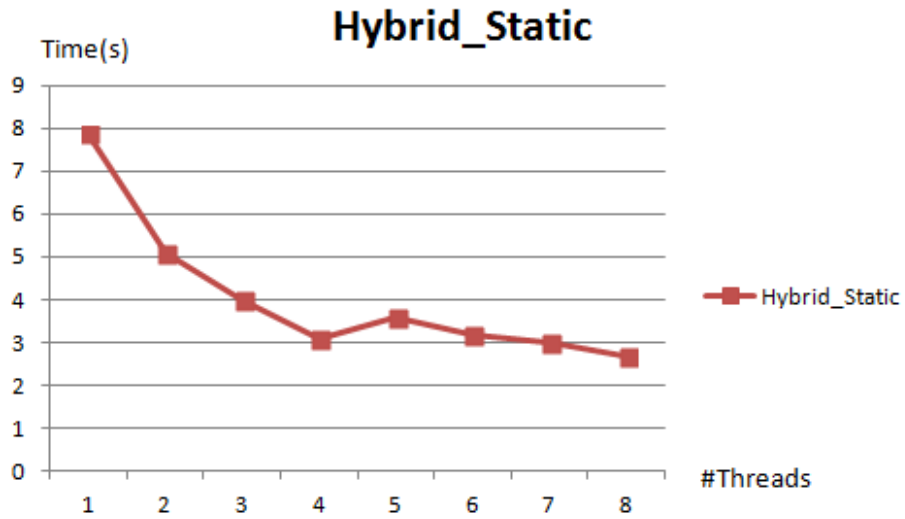


#Procs	1	2	3	4	5	6	7	8
Time(s)	12.81	6.63	4.6	3.47	2.81	2.48	2.1	1.924
#Procs	9	10	11	12	13	14	15	16
Time(s)	1.85	2.04	2.45	2.1	2.44	2.28	1.91	1.76

(X/Y = 400/400)

The dynamic version shows again great scalability, since these two versions are actually of the same style differ only the way how the omp for do the jobs iteratively whether in static or dynamic way. The overall performance is better using the dynamic way, the discussion of such issue would be go through in the corresponding comparison section. One can see that sometimes the resulting time does not scale down as the threads increases; I guess it's because of the interference of threads since there was no need to communicate thereby no communication consumed time. The other possibility may be due to the differences in computing power of each thread. Again, 16 threads made the best performance.

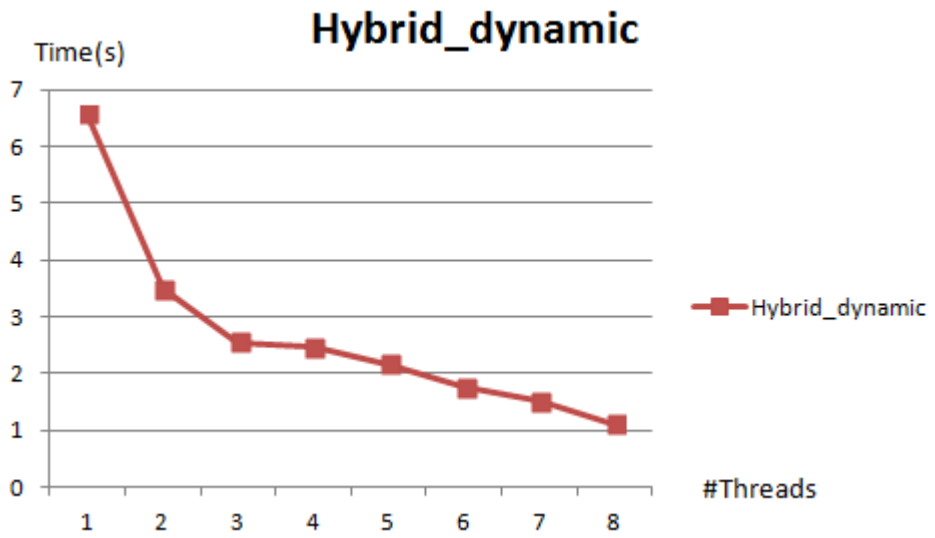
✧ **Hybrid_Static**



#Threads	1	2	3	4	5	6	7	8
Time(s)	7.87	5.07	4.01	3.09	3.58	3.19	3	2.66

(X/Y = 400/400) (*MPI_task=2, output is recorded by #Threads/MPI_task)
According to the graph, we can see that this version shows quite good scalability. But there's still sometimes when providing more processors, the resulting time turned out to be not surpassing when giving less processors by one. The reason is that, maybe the computing power is strengthened, but the required static waiting time for the communication substantially increased, causing the resulting time increases a little bit. However, overall performance is relatively getting better when adding more processors.

✧ **Hybrid_Dynamic**

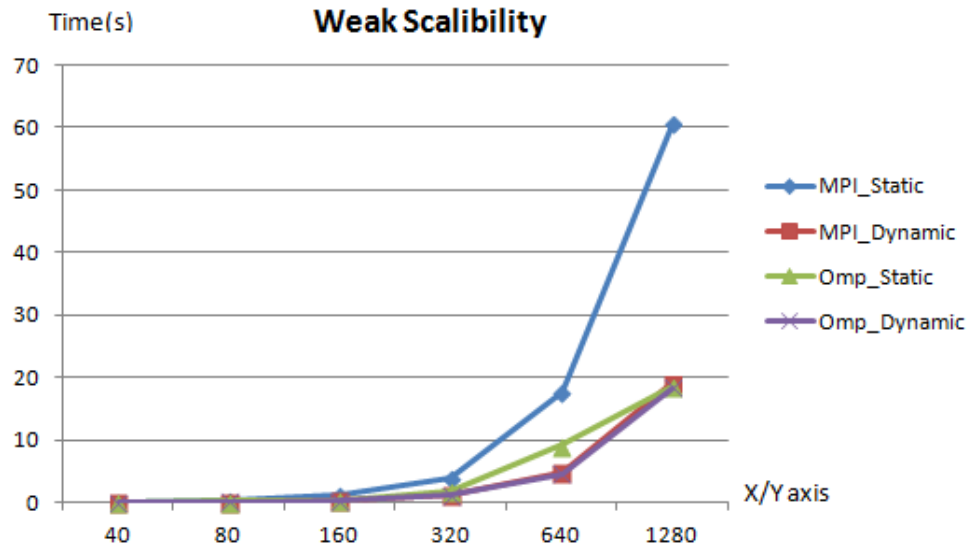


#Threads	1	2	3	4	5	6	7	8
Time(s)	6.57	3.49	2.55	2.46	2.17	1.76	1.51	1.11

(X/Y = 400/400) (*MPI_task=3, output is recorded by #Threads/MPI_task)
One of the MPI_task acted as the master, so total cores here is 2*#Thraeds + 1. Although it doesn't show prefect scaled down while increasing the threads, it somehow showed great execution time reduction. The communication time required is merely the same for each case, so there's no need to take it into account when dealing with the scaling down effect.

2. Weak Scalibility

✧ **MPI/OpenMp**



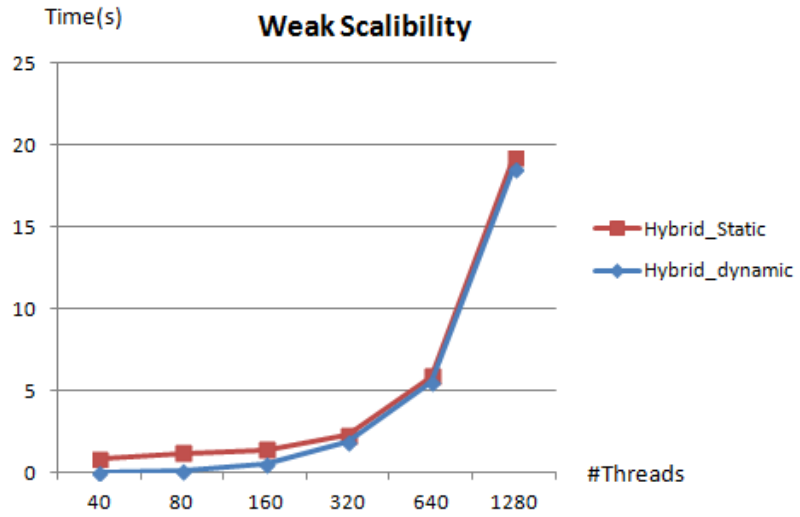
MPI_Static	40	80	160	320	640	1280
Time(s)	0.09	0.25	1.28	4	17.59	60.54
MPI_Dynamic	40	80	160	320	640	1280
Time(s)	0.03	0.1	0.33	1.25	4.86	19.14

Omp_Static	40	80	160	320	640	1280
Time(s)	0.09	0.25	1.28	4	17.59	60.54
Omp_Dynamic	40	80	160	320	640	1280
Time(s)	0.03	0.1	0.33	1.25	4.86	19.14

(Core = 8)

The graph shows quite much information to deduce the weak scalability and comparisons among four implementation versions. The worst performance is of course exhibited by the MPI_static version, although it shows quite well linearity in the beginning cases, but deviated from the linear line severely when problem size increases. Omp_static is the second worst, while it shows as well good linearity but sometimes deviated from linear a bit due to thread traffic jams. The working mechanism is quite the same between two dynamic versions, so they both show perfect weak scalability. However, MPI requires some extra consumption of time to deal with the communication, the performance of which is slightly worse than that of the Omp_dynamic version. I thus concluded that the best performance is achived by applying OpenMP method if given the same threads and same problem size.

- **Hybrid**



Hybrid_Static	40	80	160	320	640	1280
Time(s)	0.8	1.2	1.4	2.31	5.89	19.2
Hybrid_Dynamic	40	80	160	320	640	1280
Time(s)	0.05	0.15	0.52	1.88	5.55	18.52

(Core=8)

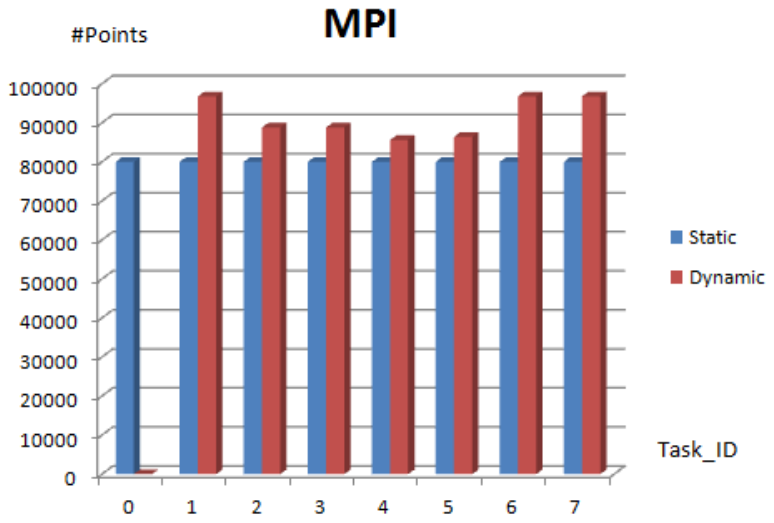
Both of the static and dynamic show similar good weak scalability, but dynamic version tends to have better overall performance and linearity. Due to the chunk size of the omp for is set to 1, the load balancing effect didn't show too much significance, but somehow still reduce the total execution time.

3. Load Balance

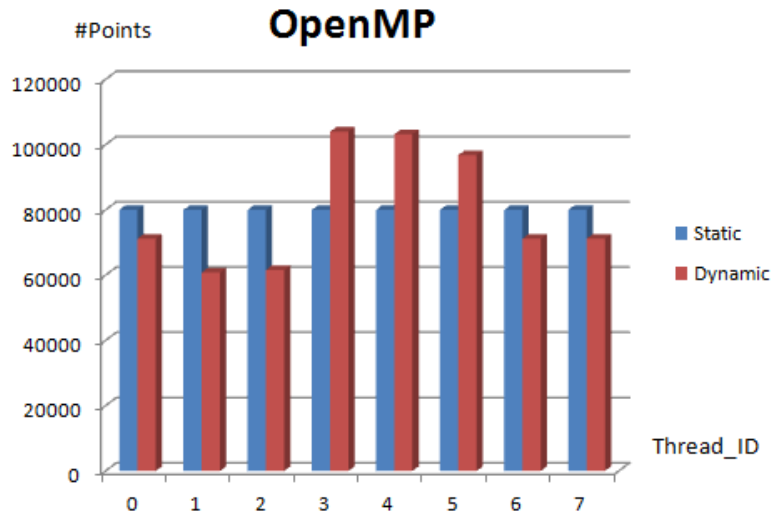
Here the setting of the experiments is all under the following instructions:

./Exe 8 -2 2 -2 2 800 800 disable

✧ **# of points in each thread(Core = 8)**



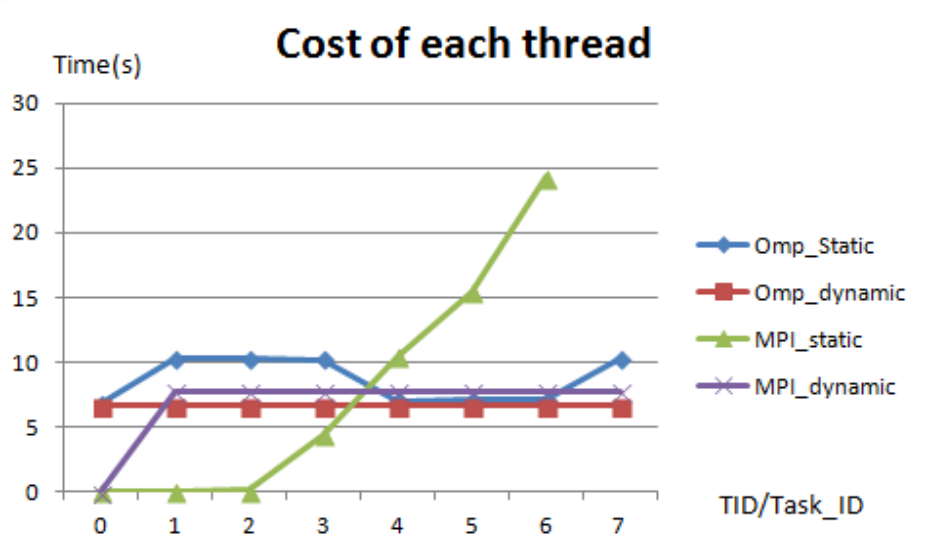
MPI_Static								
Task_ID	0	1	2	3	4	5	6	7
#Points	80000	80000	80000	80000	80000	80000	80000	80000
MPI_Dyanamic								
Task_ID	0	1	2	3	4	5	6	7
#Points	0	96800	88800	86500	86400	9680.0	96800	96800



Omp_Static								
Thread_ID	0	1	2	3	4	5	6	7
#Points	80000	80000	80000	80000	80000	80000	80000	80000
Omp_Dyanamic								
Thread_ID	0	1	2	3	4	5	6	7
#Points	71200	60800	61600	104000	103200	96800	71200	71200

In Both MPI and OpenMP version, the number of points computed in static version is the same, which is 80000 points each. While the dynamic process is applied, due to its load balancing functionality, the number of points is distributed unevenly but equally in time consumption.

✧ **Cost of each thread(Core = 8)**



Omp_Static								
Thread_ID	0	1	2	3	4	5	6	7
Time(s)	6.82	10.3	10.22	10.2	7	7.1	7.1	10.25
Omp_Dyanamic								
Thread_ID	0	1	2	3	4	5	6	7
Time(s)	6.64	6.64	6.65	6.63	6.63	6.65	6.65	6.62
MPI_Static								
Task_ID	0	1	2	3	4	5	6	7
Time(s)	0.033	0.037	0.04	0.12	4.4	10.43	15.49	24.31
MPI_Dyanamic								
Task_ID	0	1	2	3	4	5	6	7
Time(s)	0	7.73	7.72	7.74	7.73	7.73	7.73	7.73

Observe the above graph and the corresponding format; we can thus deduce much information about the load balancing done by each version. We can conclude that the MPI_static has the worst load balancing efficiency since it barely really conducted any load balancing method. Dynamic versions in both MPI and OpenMP exhibit similarly great load balancing result, with the time consumed slightly more using MPI than that of using OpenMP. Static version of OpenMP doesn't show as severe issue as the static version of MPI, and the reason was quite simple. I assigned the chunk to be 1 in the Omp for loop, so the unbalance of the load turns out to be not as significant as the MPI version did.

4. Additional Analysis

✧ **The best distribution between MPI_tasks and threads?**

To judge which one is the best memory architecture, we have to fix the

total available cores. We here conduct all the experiments while the number of cores is fixed to 8 and the problem size fixed to 640 X 640.

MPI_Static					
Exp	1	2	3	4	Average
Time(s)	15.81	16	15.49	16.8	16.025
MPI_Dyanamic					
Exp	1	2	3	4	Average
Time(s)	5.06	5.04	4.98	4.86	4.985
Omp_Static					
Exp	1	2	3	4	Average
Time(s)	4.44	4.85	4.47	4.6	4.59
Omp_Dyanamic					
Exp	1	2	3	4	Average
Time(s)	4.58	4.57	4.47	4.46	**4.52**
Hybrid_Static					
Task*Thread	1*8	2*4	4*2	8*1	Average
Time(s)	**5.6**	**5.65**	10.96	15.31	9.38
Hybrid_Dyanamic					
Task*Thread	2*4	2*4	4*2	8*1	Average
Time(s)	**5.62**	**5.68**	6.05	6.83	6.045

The format above shows that the best performance is achieved with pure OpenMP dynamic version. The reason is q1uite clear, this version doesn't require any extra communication time between threads while the hybrid version suffer such issue. Notice that the MPI_dynamic version surpasses the Hybrid_dynamic version, I guess it's because the hybrid version can only create 4 threads in each task, while the communicating time be held quite the same as the MPI_dynamic version did. The MPI version thus can balance the load with higher accuracy and hence reduce the execution time. For hybrid version only, the less the MPI task, the better the performance, since more tasks implies more extra communicating time required and more issue on variation of computing power through the MPI_tasks.

✧ **The best distribution of cores between machines?**

Following the previous problem, we have known that the pure OpenMP dynamic version achieved the best performance, but it's got to be limited by the real hardware cores distribution. That is, what if my node only has 4 processors? And thus the extra 4 threads would cause traffic jam between

cores since the hardware has to do the mapping itself, resulting in lower efficacy. So the following experiment is conducted to push the limit to the hard ware server to prove my assumption, which is distributed the excessive requiring threads to another MPI_task.

Problem size is again fixed to 640 X 640.

And this time I only selected two version to compare with, since OpenMP version is proved to be the best one among all versions in the previous experiments, here I want to test if the redistribution of cores can surpass the original result.

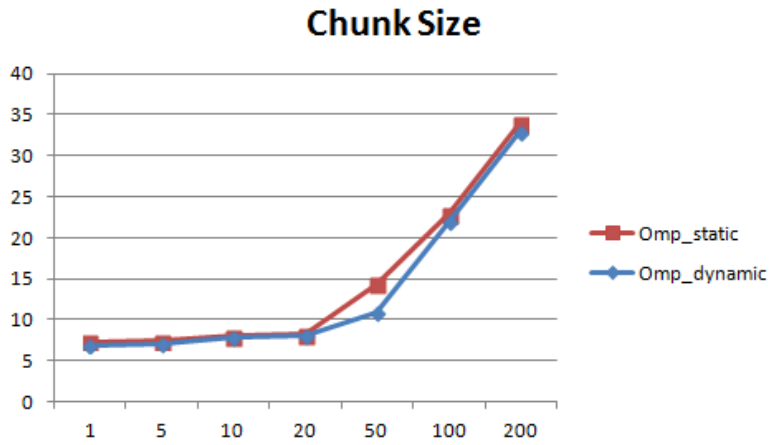
Hybrid_Dyanamic					
Task*Thread	2*8	2*8	4*4	4*4	Average
Time(s)	4.15	4.1	4.94	4.86	**4.5125**
Omp_Dyanamic					
Thread	16	16	16	16	Average
Time(s)	5.584	5.668	4.89	4.62	5.1905

The experiment is done under multi job queue, which has 8 processors in each node. The result showed that the evenly distributed threads throughout all MPI_tasks from Hybrid_dyanmic version had stricken the better performance of the two. This met my expectation since the Hybrid version here is truly parallelized and not limited by the hard ware processors on one node, while excessive threads created by the OpenMP version lower the overall performance severely due to the mapping traffic jam. To sum up, if the provided cores in one node is sufficient for the corresponding test, OpenMP should be applied, if not, the evenly distributed method of Hybrid version should be instead

- **Omp for chunk sizes?**

  The Omp for relies strongly on the specified chunk size of how to do the job iteratively by threads, and I conducted a simple experiment on such issue to see if anything interesting turned out.

  The problem and cores are fixed to 800 X 800 and 8 cores.

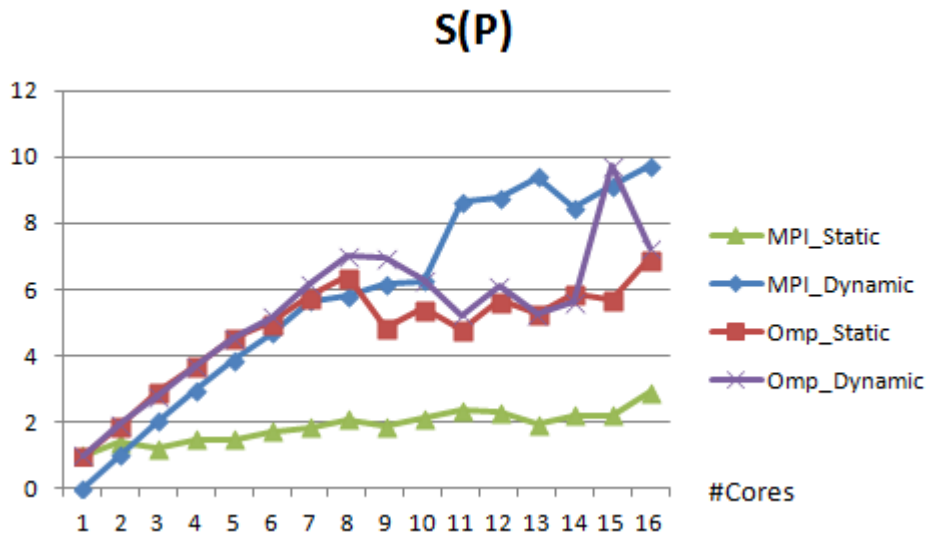


Omp_Static							
Chunk_size	1	5	10	20	50	100	200
Time(s)	7.357	7.416	8	8.225	14.361	22.95	34.05
Omp_Dyanamic							
Chunk_size	1	5	10	20	50	100	200
Time(s)	6.94	7.13	7.92	8.12	10.96	21.94	33

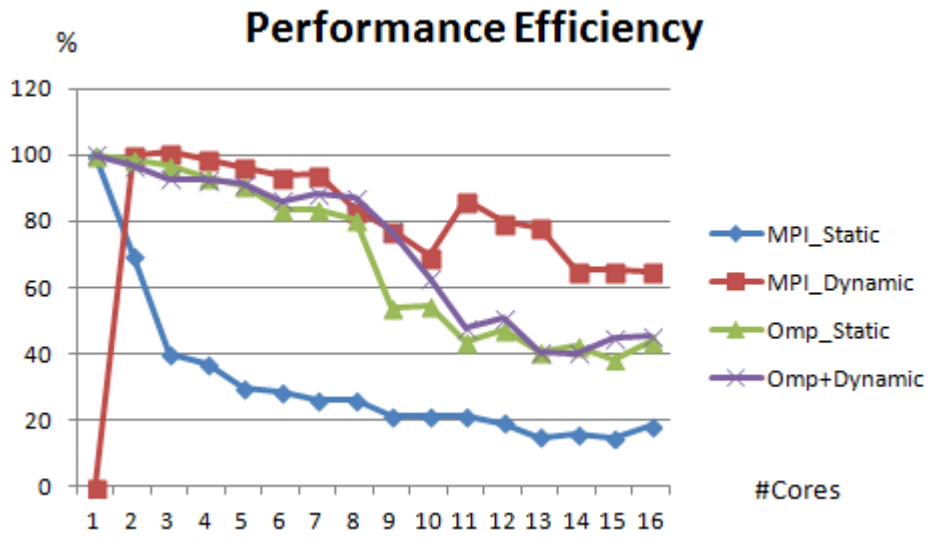
From the graph and the format, we can notice that as long as the size of chunk increases, the performance will get worse, since the each thread functions more and more independently, thus reduce the efficacy of the load balancing. The load of each thread couldn't be balanced well due to the increment of size of chunks, since the complexity of the output graph of the Mandelbrot set is relatively unevenly distributed. Another interesting thing is that the two curves deviated the most from each other while the chunk size ranges from 20 to 100. My explanation to such phenomenon is that the load balance efficiency isn't significant enough when the chink size is too small or too large, which the former tends to balance the load through continuous work assignment even if static clause is specified; while the latter usage diminish the advantage of dynamically work assigning since each thread got too much load to digest.

✧ **Performance efficiency**

Analysis of the system efficiency, here I provide speed up factor and the performance efficiency charts.



The above graph shows the speed up factor of the four versions of implementations. As one can see, the MPI_Static suffers from the worst speed up factor due to its bad functionality of distributing the load evenly. Omp_Dynamic has pretty much the same outcome as its Static version does; the deviation is not too significant but still can be noticed. The main reason is as well the sizes of chunks. Overall, MPI_dynamic exhibits the best S(p), although it kind of violent my expectations. But I guess the worsen performance of the Omp version is attributed to the interference between VMs.



The performance efficiency has quite similar result to the speed up factors since they're correlated. But interestingly, except for the MPI_Static version which is ensured to be the worst, all other three versions show declinations of the efficiency while the given cores is more than 8. The most convincing possibility is still the interference for the Omp versions. As for the MPI

version, the communication time might have to be taken into account this time, though I still think the contributor to such effect should still be attributed to problems of VMs.

5. Experience

✧ 1. What have I learned?

During this assignment, I have learned several new APIs of the new library, OpenMP. Like pthread, it's architecture is implemented by shared memory, so it feels as though I was again writing pthread codes to deal with threads. But in this time, since the library provides novel APIs with much more efficiency, I tried to apply them to complete the assignment. However, the tradeoff of such high efficiency somehow brought ambiguity for variables shared throughout all threads, so one may pay more attention on designing code more explicit. For example, I must have been quite cautious when applying Omp for to do the iterative work automatically assigned by the mapping mechanism of the hard ware. I also reviewed some resources about distributed memory, and thought of the difference between which and shared memory. By mastering these two kinds of memory architecture, I can decide for myself which is the best way to deal with the current problem or its sub problems. And thus, for the hybrid version, I can combine those two methods to take advantages on both sides to achieve even higher accomplishment.

✧ 2. Any difficulties?

At first, it was quite hard for me to realize how exactly the Omp for functions. And ever since I decide such API would be the core and the most essential part of my OpenMP version, I suffered a little from deciding how to parallelize the original sequential version through this not-so-clear defined function block. Fortunately, I found out it was quite simple for the mechanism and hence simply apply it to achieve my goal with ease. For the MPI version, I sometimes would encounter mistakenly using variables as it were shared. The trouble dealing with distributed memory is significantly noticed when communicating from one task to another, and it added a lot more complexity to finish a simple block that could be done without too much effort if shared memory is applied. Such issue still strokes me when I was designing my own hybrid version. But anyhow, I found my way out to independently use two memory architectures and eventually combined them all into a master rank to deal with the final I/O. Hence, the difficulty was solved and efficiency of using both architectures turned out to be clearer.

✧ **3. Feedback**

Sometimes when I subscribe the job to the system queue, it would return daemon killed information and terminate the assignment, is this something wrong with the VMs? Another issue is that, I sometimes couldn't get the output eo file if I use the core16 job queue, is this again the problem of the system? Since I can get my resulting output file written on my own to record some needed information but I just can't get the system output.

6. Source codes

Please refer to the attached files and the instructions of how to compile the source code as well as execute the executable file at the very beginning of this report.

Compile:

MPI: mpicc –o MS_MPI_dynamic –lX11 MS_MPI_dynamic.c

OpenMP: gcc –o MS_Omp_dynamic -fopenmp –lX11 MS_Omp_dynamic.c

Hybrid: mpicc –o MS_Hybrid_dynamic -fopenmp –lX11 MS_Hybrid_dynamic.c

Run:

MPI: mpirun ./MS_MPI_dynamic 8 -2 2 -2 2 400 400 enable outputfile

OpenMP: ./MS_Omp_dynamic 8 -2 2 -2 2 400 400 enable outputfile

Hybrid: mpirun –n 2 ./MS_Hybrid_dynamic 8 -2 2 -2 2 400 400 enable outputfile