# Parallel Programming

Manderlbrot Set

Peter Huang

## 1、 Design

### a、 What the differences between the implementation of six version

**In MPI & OpenMP version** The implementation is almost the same.

**In static version:** MPI and OpenMP evenly assigns points to N processors or cores at the beginning of programming, once N processors finished assigned task, master or main program is responsible for display result.

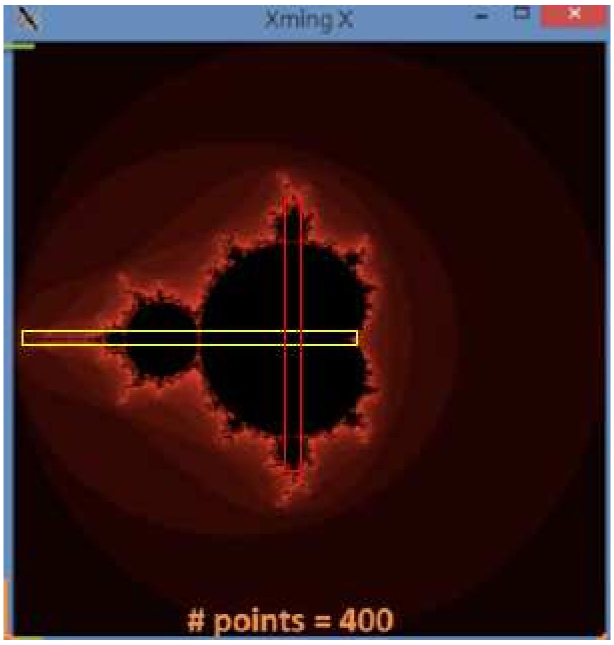
**In dynamical version:** MPI and OpenMP just assigns ONE X points to each processors or cores, once any processors finished its task, the program assign the next X points, which can get good load-balance in each processors or cores.

**In hybrid version** both static version and dynamic version combine the MPI & OpenMP's implementation in program.

**In dynamical version** the program assigns the amount of #OpenMP_NUM_Thread to each processors or cores, whose functionality is to get good trade off between the communication time and the utilization of each processors, because in hybrid one compute node may have many processors, which can make the utilization of each processor maximized.

### b、 How do you partition my tasks

I use column strip to partition all X points. And each processors will finish the assigned X points * the number of Y points in each computing task. Why I based on idea to partition my task? The way is to reduce the communication time and also get good utilization and performance. Because if I partition task by (x,y) points, there are too many communication time, because I will send the number of X point * the number of Y point points to other processors. However, why I used column-major but not row major to partition my task, which is because the column-major may cover less heavy workloads, when compared with row-major by visual check in graph as following graph. The yellow mark's length will longer than red mark's length.



## c、 What technique do you use to reduce execution time and increase scalability

**Reduce communication time**

**1**、 In one send & receive action, I try to send the ONLY one time with all computing information to each processors.

**2**、 In one send & receive action, I try to minimize the amount of my transmitted data. If I can pre-cache my sent information, the program will cache it, once receiver send results back to master, the receiver will not send cached data, and master can directly to use pre-cached data, which is for reducing the transmitted data amount and guaranteeing the only ONE transmitted task between sender and receiver.

**3、** Fixed transmitted data buffer. For example. If data size on each transmitted task is variable, I should send data size before sending the real data, which makes more than one communication times between sender and receiver, which will increase the communication time.

**Reduce execution time**

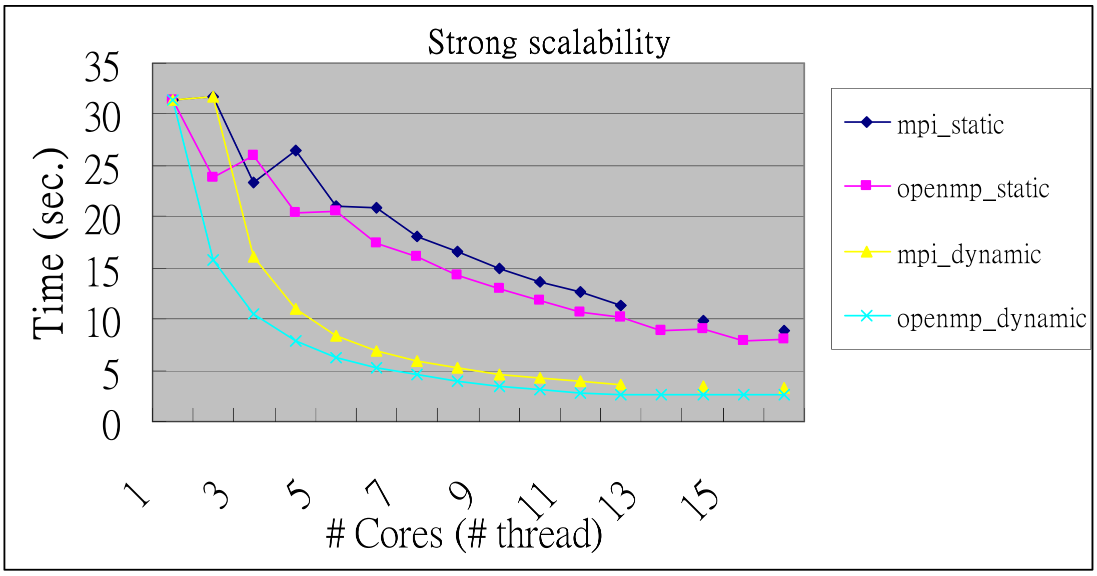
**1、** Use unblocked mechanism to reduce execution time. For example: use ISend instead of Send

**2、** Avoid some extra data handling in the programs, and strip off some unnecessary codes.

## 2、 Performance Analysis

### a、 Scalability chart - Strong and Weak Scalability

**Strong Scalability**



**Total points (600*600)**

| #Cores / threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| mpi_static | 31.362 | 31.732 | 23.265 | 26.41 | 20.385 | 20.881 | 18.12 | 16.582 |
| openmp_static | 31.359 | 23.87 | 25.978 | 20.399 | 20.546 | 17.475 | 16.108 | 14.345 |
| mpi_dynamic | 31.364 | 31.7 | 16.091 | 10.95 | 8.356 | 6.841 | 5.846 | 5.277 |
| openmp_dynamic | 31.361 | 15.846 | 10.521 | 7.908 | 6.305 | 5.276 | 4.53 | 3.966 |

| #Cores / threads | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|
| mpi_static | 14.96 | 13.575 | 12.647 | 11.322 |  | 9.907 |  | 8.942 |
| openmp_static | 12.919 | 11.894 | 10.647 | 10.261 | 8.859 | 9.009 | 7.886 | 8.129 |
| mpi_dynamic | 4.623 | 4.293 | 3.922 | 3.612 |  | 3.45 |  | 3.232 |
| openmp_dynamic | 3.527 | 3.173 | 2.873 | 2.654 | 2.642 | 2.63 | 2.577 | 2.564 |

|  | 12 |
|---|---|
| hybrid_static | 10.96984 |
| hybrid_dynamic nodes=2;ppn=(#thread -1 ) | 3.591366 |

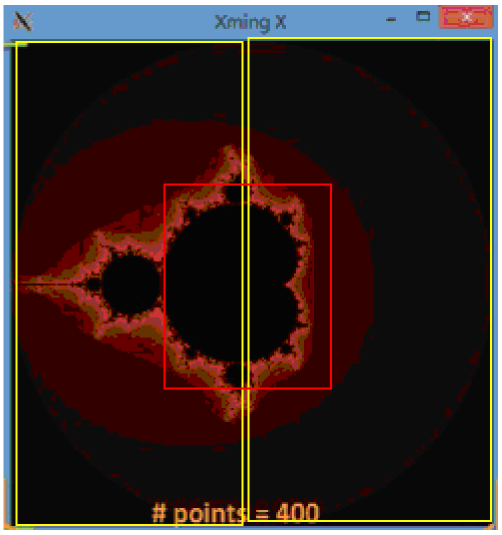
**[Analysis]**

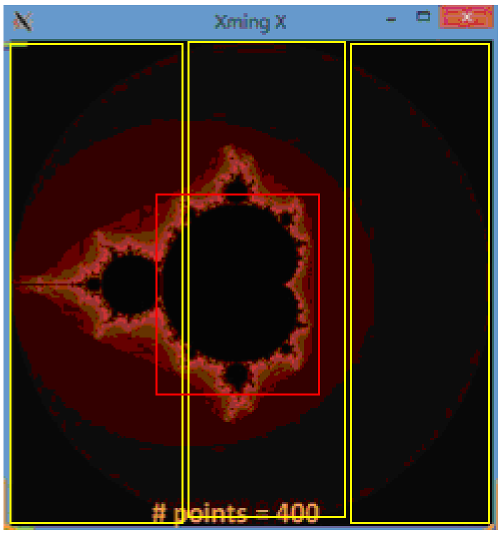
1、 The execution time: openmp_dynamic > (faster) mpi_dynamic > openmp_static > mpi_static in the same amount of threads.

Because openmp uses shared memory, all processors can get shared memory in their threads. However, in mpi each processors should send their data to another processors when they need to access data in different processor. So the communication time will be more in mpi version when compared with openmp version. Hence, it is the reason when openmp version always faster than mpi version.

2、 in dynamic version, when threads get more, the execution time will be less. Because in dynamic version, I adopt the loading-balance mechanism, when process finishes their jobs or is ready. Then, the master send next jobs to the ready process, so the bottle neck will not be bound on the heavy-loading processors, which happens in the static version.

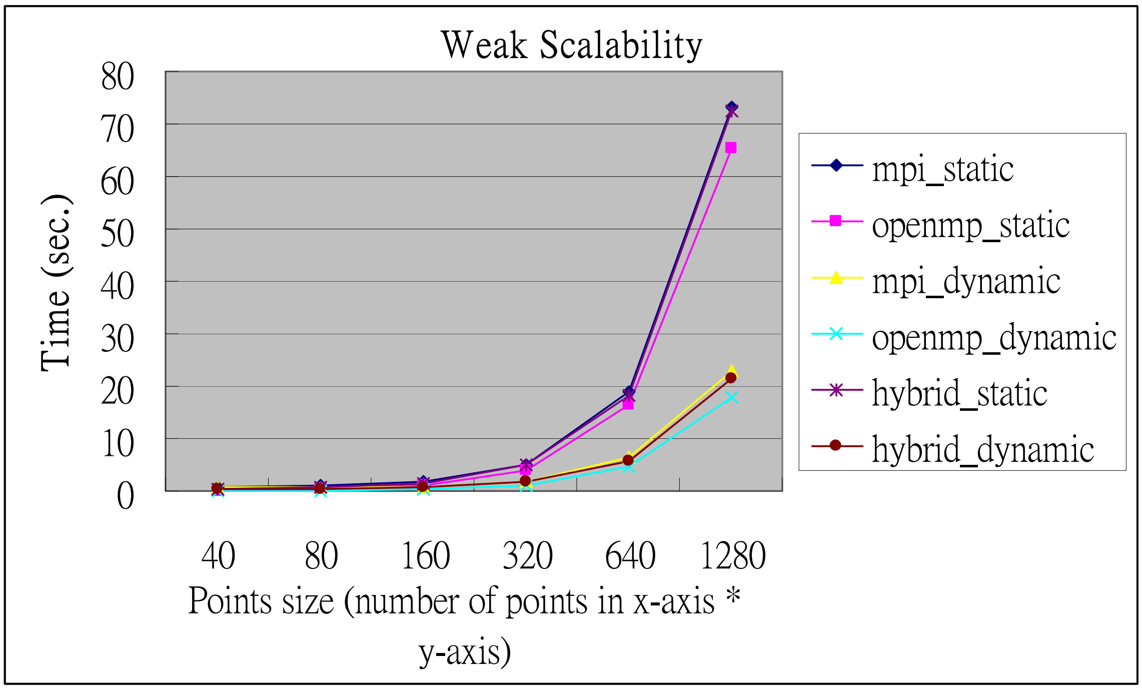
3、Why in static version the execution time is poor when they get more threads or cores like above colored square. Because in static version, the partitioned task dominate how long the job has done, in above case, although in static version it gets more thread, however, one of threads have assign more heavy-loading tasks, when compared with less threads assigned.　Seeing following diagram.





The upper-left diagram has only 2 threads. The heavy-loading part( red-square) has evenly separated into two threads. However, when it get more threads like #thread=3, the heavy-loading tasks has been totally assigned into one thread. Although it get more threads, the performance get worse, because of concentrating heavy-job on one thread.

## Weak Scalability



**mpi_static, mpi_dynamic, hybrid_static, hybrid_dynamic :**
**1 master does nothing and just receives points, and another #proc=7 are computing processors**

| (core=8) sec / points | **40** | **80** | **160** | **320** | **640** | **1280** |
|---|---|---|---|---|---|---|
| **mpi_static** | 0.686 | 0.917 | 1.791 | 5.141 | 18.76 | 73.211 |
| **openmp_static** | 0.076 | 0.282 | 1.03 | 4.057 | 16.281 | 65.482 |
| **mpi_dynamic** | 0.609 | 0.729 | 0.89 | 1.858 | 6.342 | 22.908 |
| **openmp_dynamic** | 0.042 | 0.096 | 0.309 | 1.146 | 4.481 | 17.847 |
| **hybrid_static** | 0.492 | 0.697 | 1.548 | 4.977 | 18.089 | 72.323 |
| **hybrid_dynamic** | 0.445 | 0.512 | 0.817 | 1.643 | 5.87 | 21.407 |

**[Analysis]**

1、 The execution time: openmp_dynamic > (faster) hybrid_dynamic mpi_dynamic > openmp_static > hybrid_static > mpi_static in the same amount of threads.

Time complexity: In static version & dynamic version, there only data size is different into different processors.

phase1: communication time: x points is send to slave

t (comm1) = p( t(start_up) + t(variable data) )

phase2: computation -- compute points

t (comp) <= max_iter *n /p

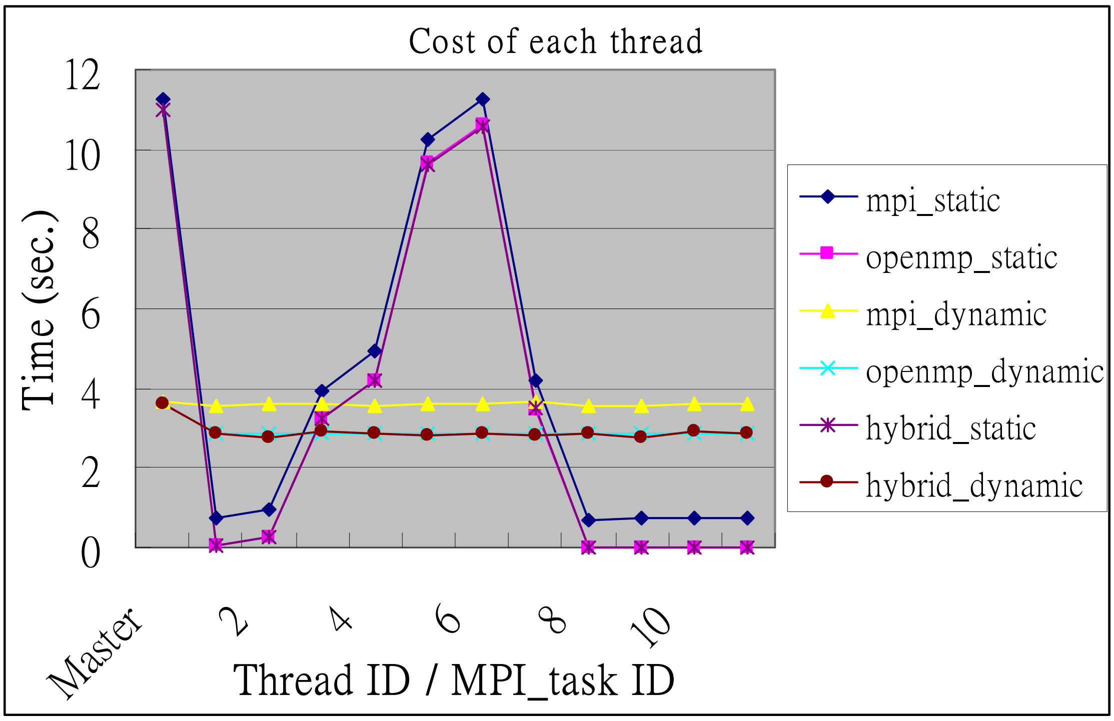
phase3: communiation - sned data back to master

t (comm2) <= n/p (t(start_up) + t(data) )

Total:

tp <= n/p + (n/p +p )(t (start_up) + t(data))

## b、 Load Balance Chart



**[settings]**

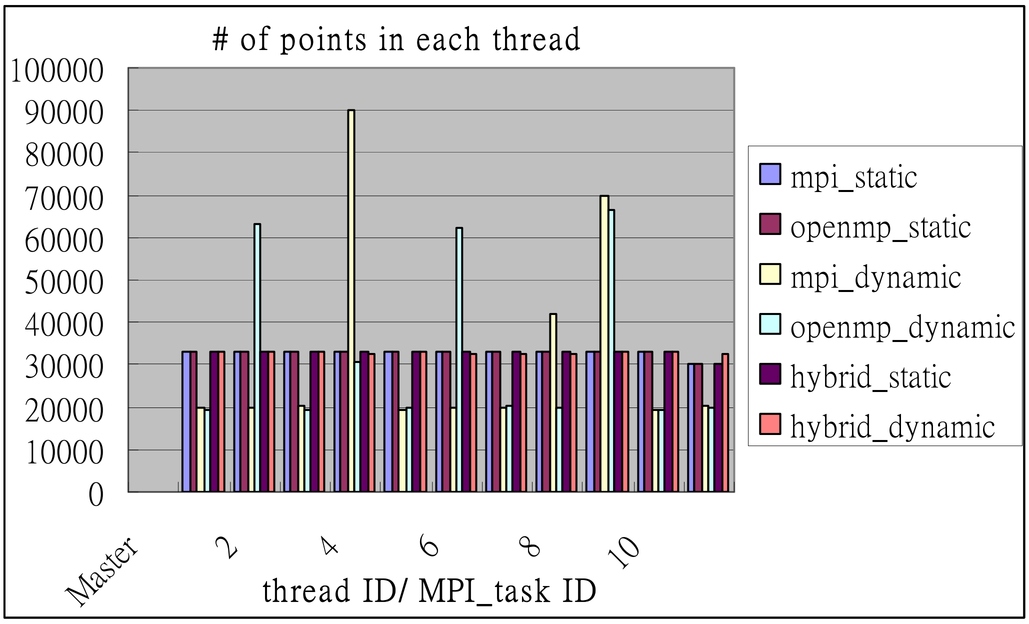
**mpi=12 threads, openmp=11 threads (openmp : no master)**

**hybrid_static, hybrid_dynamic:**

**nodes=2:ppn=11, #openmp=11；master has #process:10 unused.   total #proc:12**

| Total Points(600 * 600) / sec | **Master** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| **mpi_static** | 11.26925 | 0.760599 | 0.969078 | 3.95264 | 4.927485 | 10.26933 |
| **openmp_static** |  | 0.054485 | 0.262411 | 3.238425 | 4.220837 | 9.675128 |
| **mpi_dynamic** | 3.64091 | 3.582675 | 3.616137 | 3.599344 | 3.566058 | 3.624363 |
| **openmp_dynamic** |  | 2.84853 | 2.843956 | 2.850107 | 2.84397 | 2.850395 |
| **hybrid_static** | 10.96984 | 0.054524 | 0.26045 | 3.236427 | 4.22029 | 9.615084 |
| **hybrid_dynamic** | 3.591366 | 2.887186 | 2.759426 | 2.910624 | 2.85677 | 2.826508 |

|  | **6** | **7** | **8** | **9** | **10** | **11** | **total** |
|---|---|---|---|---|---|---|---|
| **mpi_static** | 11.26099 | 4.169173 | 0.71072 | 0.735677 | 0.727339 | 0.719162 | 11.26925 |
| **openmp_static** | 10.59832 | 3.457131 | 0.00109 | 0.000901 | 0.00086 | 0.000678 | 10.63434 |
| **mpi_dynamic** | 3.591053 | 3.62938 | 3.557659 | 3.574389 | 3.632587 | 3.607686 | 3.64091 |
| **openmp_dynamic** | 2.843942 | 2.853878 | 2.848893 | 2.843949 | 2.855265 | 2.856771 | 2.883211 |
| **hybrid_static** | 10.55314 | 3.49545 | 0.001042 | 0.000856 | 0.000825 | 0.000644 | 10.96984 |
| **hybrid_dynamic** | 2.883398 | 2.804247 | 2.855996 | 2.756541 | 2.917748 | 2.876983 | 3.591366 |



**[settings]**

**mpi=12 threads, openmp=11 threads (openmp : no master)**

**hybrid_static, hybrid_dynamic:**

**nodes=2:ppn=11, #openmp=11；master has #process:10 unused. total #proc:12**

| Total Points(600 * 600) | **Master** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **mpi_static** | 0 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 30000 |
| **openmp_static** | | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 30000 |
| **mpi_dynamic** | 0 | 19800 | 19800 | 20400 | 90000 | 19200 | 19800 | 19800 | 42000 | 69600 | 19200 | 20400 |
| **openmp_dynamic** | | 19200 | 63000 | 19200 | 30600 | 19800 | 62400 | 20400 | 19800 | 66600 | 19200 | 19800 |
| **hybrid_static** | 0 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 33000 | 30000 |
| **hybrid_dynamic** | 0 | 33000 | 33000 | 33000 | 32400 | 33000 | 32400 | 32400 | 32400 | 33000 | 33000 | 32400 |

**[Analysis]**

Based on the above analysis, you will find out that the dynamic version in MPI, OpenMp, and Hybrid get almost the same computing time in each threads, because we assign points to each process by processer's loading. If one of process get heavy-loading jobs, which will be executed more time, we won't assign new jobs to it. We just assign task to some processors, which is ready to do next job. So on the diagram [Cost of each thread] you can see dynamical version almost get horizontal line in diagram. And, also you see the amount of points which each thread gets in dynamic version are more variable than static version, which is due to the mechanism that we assign tasks into spared processors dynamically when programs running.

However, in static version, each threads get variable execution time but almost equal the amount of tasks in each threads. It is due to programs assigns almost equal amount of data in advance. However, between the data we don't know which one is heavy-loading or light-loading job. So the most of time we get poorer performance, because one of processor gets heavy-loading jobs, which dominates our execution time-complexity.

## c、 Other experiments

### (1)、The best distribution between MPI tasks and threads

**Loading balance:**

Partition tasks by points > Partition tasks square > Partition tasks by column & row

**Communication times:**

Partition tasks by column & row > Partition tasks square > Partition tasks by points

So as we found out that there are some trade off between loading balance & communication times. However, if we partition our tasks more smaller, there will get better performance, but it increases communication time. However, no matter what method we use, **first**, if make sure each thread can take almost the same time in each threads, **second**, also if we can reduce more data handling or data transmission action, which can guarantee that our program get best loading balance. Finally, I adopt column-strip to validate my thought, because I could reduce my communication time and finish more jobs in one transmission task.

| Total Points(600 * 600) / sec | **Master** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| **mpi_dynamic** | 3.64091 | 3.582675 | 3.616137 | 3.599344 | 3.566058 | 3.624363 |
| **openmp_dynamic** |  | 2.84853 | 2.843956 | 2.850107 | 2.84397 | 2.850395 |

|  | **6** | **7** | **8** | **9** | **10** | **11** | **total** |
|---|---|---|---|---|---|---|---|
| **mpi_dynamic** | 3.591053 | 3.64938 | 3.557659 | 3.574389 | 3.632587 | 3.607686 | 3.64091 |
| **openmp_dynamic** | 2.843942 | 2.853878 | 2.848893 | 2.843949 | 2.855265 | 2.856771 | 2.883211 |

## (2)、The best distribution of cores between machine

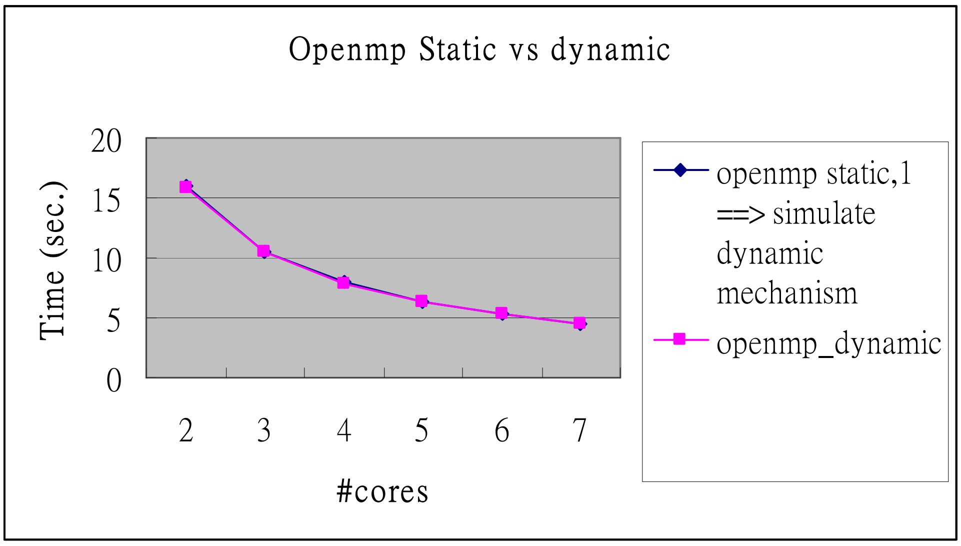
| #Threads or Cores=9 | 600 * 600 | 1000*1000 |
|---|---|---|
| hybrid_dynamic nodes=2;ppn=8,   num_mpi_per_node=1, openmp=8 | 4.681 | 11.905 |
| hybrid_dynamic nodes=1;ppn=9,   num_mpi_per_node=9, openmp=1 | 4.496 | 11.654 |
| hybrid_dynamic nodes=3;ppn=4,   num_mpi_per_node=1, openmp=4 | 4.527 | 11.675 |
| hybrid_dynamic nodes=5;ppn=2,   num_mpi_per_node=1, openmp=2 | 4.462 | 11.575 |
| hybrid_dynamic nodes=9;ppn=2,   num_mpi_per_node=1, openmp=1 | **4.39** | **11.354** |

### [Analysis]

If #process is larger than 1machine, which means we need to cross computing nodes, the rule that we can get best performance is to reduce the communication time and also increase the amount of computing data in one sender and receiver transmission. By the rule, in the hybrid the openmp's thread is dominated our performance, because if we assign more thread into openmp's thread, there are more processors can compute data in one communication request. However, once we do experiment, the result is totally different.

However, why the last row, which uses less openmp threads but uses more head node, which increase more communication time, get better performance than others case. There are two reason. First is that in our case, we just send data in one computing node without crossing computing nodes. The second reason is that we do really cross different computing node but the transmitted amount of sent or received data are not take too much cost of transmission. And also, as we mention before, the heavy-loading tasks dominates our performance, however, if the receivers process handling data, one of them need to handling heavy-loading data, the rest of process will take a rest without asking new request. So this is the reason why we use more computing node will get better performance, because they handle data individually, once one of processor finished their data, the processor can make new requests without idle or taking a rest.

(3)、To check if OpenMp static version can reach the same performance of dynamic version of openMp.



| **(600 * 600)** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|
| **openmp static,1** | 15.968 | 10.581 | 8.031 | 6.304 | 5.28 | 4.543 |
| **openmp_dynamic** | 15.846 | 10.521 | 7.908 | 6.305 | 5.276 | 4.53 |

[Analysis]

Based on the setting, the openmp_static program ties to partition tasks like what dynamic version of openMP do, and thus the static version reaches the same performance of dynamic version.

(4) To compare different chunk size in dynamic version of mpi

| **(600 * 600) mpi_dynamic, chunk size** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
|---|---|---|---|---|---|---|---|
| **1 x point** | 6.831 | 5.846 | 5.16 | 4.617 | 4.263 | 3.922 | 3.652 |
| **2 x points** | 6.804 | 5.811 | 5.173 | 4.601 | 4.139 | 3.89 | 3.613 |
| **3 x points** | 6.814 | 5.831 | 5.15 | 4.693 | 4.189 | 3.992 | 3.713 |
| **6 x points** | 7.13 | 6.034 | 5.113 | 4.771 | 4.496 | 4.118 | 4.086 |
| **12 x points** | 7.679 | 6.429 | 5.343 | 5.032 | 4.705 | 4.553 | 4.295 |
| **24 x points** | 7.367 | 6.947 | 6.306 | 5.968 | 6.019 | 5.984 | 6.053 |

### [Analysis]

Because the performance is dominated on heavy-loading task, in dynamic version I set chunk size =1, which is try to get better loading-balance when master assign un-done job into processors. However, in mpi there are a communication time between each send and receive action. So I try to adjust my chunk size to evaluate if there are good trade off between how to partition task and the communication time when processors deliver data.

Performance is dominated by “heavy-loading task” & “communication time”. In chunk size 1~3 x points, in one task  master needs to send chunk X size * num of y points (like 1 * 600 , 2* 600 , 3 *600). If chunk size gets larger, the communication times will be reduced. And in chunk size 1~3 x points, there are not too much different between the execution time, although there are some variation between them. However, the variation value should be tolerated and be limited in certain scope. It is due to “heavy-loading task” & “communication time” different combination.

However, if the chunk size increase more, like x points from 6 to 24, the performance get worse, although the communication times is reduced. By the communication times is reduced, but the performance get worse. It is due to more heavy-loading tasks assign to one processors. Finally, the total execution time get worse.

## 3、 Experience

### a、 What I have learned from this assignment

1、 How to get better loading balance in processors and validate if my idea can be implemented by better performance.

2、 How to mix mpi with openmp into different cores and how to assign your task by different handling action. For example: When threads’ execution time is variable, we should minimize our thread’s task and assign the most of processors into MPI but not openmp in the same and limited cores. Because of variable data computing, if we assign more cores to openmp, once one of them finish the job, they will be idle without utilizing maximum processors. However, if the threads’ execution time is fixed, we can assign the most of processors into openmp, because we can largely reduce the communication time.

## b、 What difficulty did I encounter when implementing this assignment

1、 How to reduce my execution time and make improved in my performance like pre-fetch or pre-cache some values for reducing communication time. In the begin, I send too many data to slaver, which I get worse performance. Once I try to modify my code to cut off some unused-information and send data once, and then finally the performance get better.

2、 How to do some experiment to make sure if my idea is match the my expected result.

## c、 Any feedback.

None.