Assignment2 Report

116010264

杨云腾

Use a different machine. May have a little bit different result

How to run the program

One constraint to run the program, the number of tasks you set must be a factor of the pixel number. There are thee methods I used to complete the problem. For MPI implementation of this problem, I use dynamic scheduling and static scheduling to complete the problem. Besides, I use static scheduling for pthread.

To run the dynamic scheduling, first compile it:

```
mpic++ mandelbrot_dynamic.cpp -lX11 -o mandelbrot_dynamic.out
mpirun -n [number of processes] ./mandelbrot_dynamic.out [number of tasks] [width] [height]
```

To run the static scheduling, mpi version:

```
mpic++ mandelbrot_static.cpp -lX11 -o mandelbrot_static.out
mpirun -n [number of process] ./mandelbrot_static.out [width] [height]
```

To run the static scheduling, pthread version:

```
g++ mandelbrot_pthread.cpp -lX11 -lpthread -o mandelbrot_pthread.out
./mandelbrot_pthread.out [number of threads/tasks] [width] [height]
```

To simplify TA's work, I implement three runnable files. Each contains the runnable file set with a default value. They are named after .sh in the submitted zip file.

Introduction

In this assignment, the problem is about calculating the Mandelbrot set. To calculate out the Mandelbrot set. From the description in the assignment document, it describes Mandelbrot set as a set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when iterating the following function:

$$z_{k+1} = z_k^2 + c$$

In the formula above, z_{k+1} is the (k + 1)th iteration of the complex number z = a + bi and c is a complex number giving the position of the point in the co mplex plane. For the image with height H and width W, $c = \frac{x-height/2}{height/4} + \frac{y-width/2}{width/4} \times i$.

To start the problem, the initial value for z_0 is zero. The iterations continued until the magnitude of z_k is greater than a threshold or the maximum number of iterations have been achieved. The magnitude of z_k is computed as:

$$z_k = \sqrt{a^2 + b^2}$$

Then, computing the complex formula $z_{k+1} = z_k^2 + c$ is simplified by recognizing that:

$$z^2 = a^2 + 2abi + b^2 i^2 = a^2 + 2abi - b^2$$

Therefore, real part is the $a^2 - b^2$ while the imaginary part is 2abi. The next iteration values can be produced by computing:

$$z_{real} = z_{real}^2 - z_{imag}^2 + c_{real}$$

$$z_{imag} = 2 z_{real} z_{imag} + c_{imag}$$

In a roll, it requires two versions of the problem's solution. One is about using message passing interface, the other one is about using posix thread.

Design

For my design, I use two kinds of methods to solve the MPI version of the problem. The first method I use is the static scheduling. Following shows how I design this way of solving the problem:

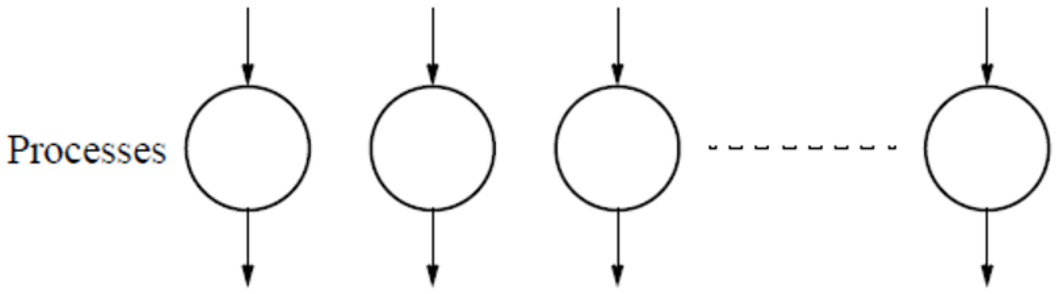

Figure 2.1

In my design of the static scheduling, the jobs will be spread out to all processes currently had as long as it can be spread out equally. After all the jobs are finished, I use MPI_Gather to gather all the outcome to an output. In the end, the output will be shown via X-server.

The second way I use is the dynamic scheduling. Following shows my design:

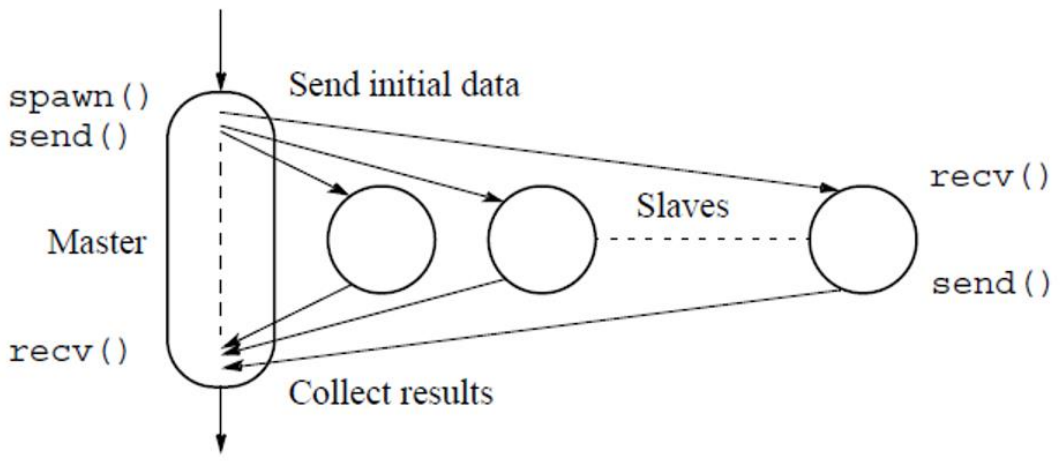

Figure 2.2

I use the picture from lecture to show my design of the MPI dynamic scheduling. My design is basically the same as the dynamic scheduling method shown in class. The master process is used to distribute jobs and collect solution from slave processes. After the calculating steps got finished, the master process will receive results from the slave processes. After all the calculating units are finished, the master process will show the outcome in a graph.

For pthread design of the problem, I continued using the method of static scheduling and did a comparison between these two methods. Here is the design of the program:

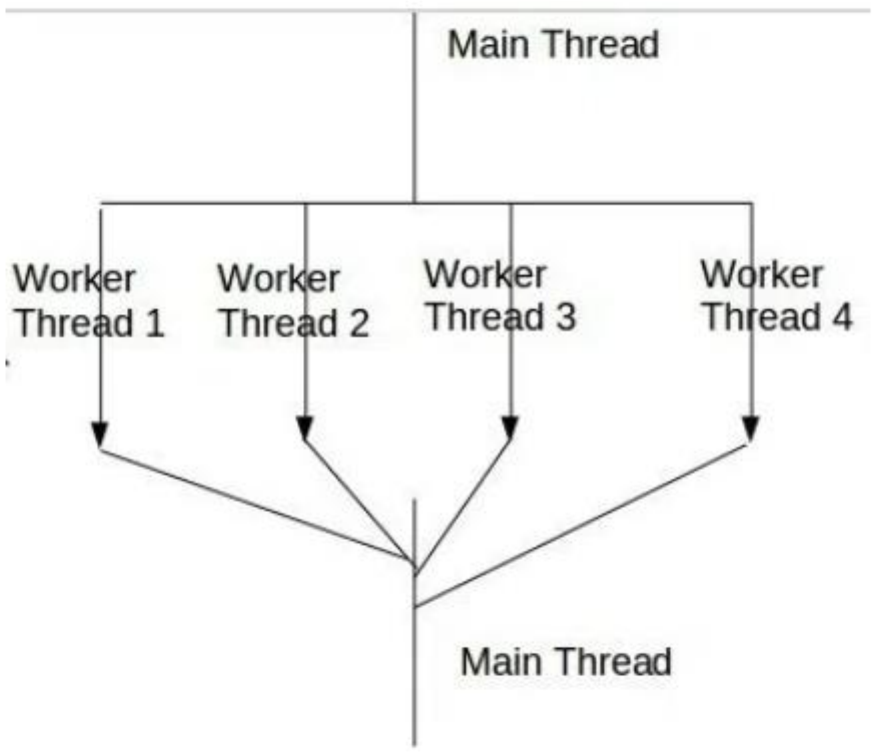

Figure 2.3

The design above is similar to the design in Figure 2.1. Both of them spread out all tasks to all processing units and gather the results when all finished. But there should be some differences for computing tasks rise up.

Result

Here is the result after run the program of normal Mandelbrot:

singgle process process time with problem scale				
	runtime			
300	0.024172			
500	0.051304			
800	0.134294			

1000	0.208299			
3000	1.687845			
5000	4.596164			
8000	13.52695			
10000	18.93942			

Table 3.1

From the result of the normal Mandelbrot, it is easy to find out that the program runs slower when the problem scale rises up. The complexity is very large for the Mandelbrot set if we run it only on a single process.

Here is the result of the static scheduling pthread Mandelbrot:

			pthread methed with core use and problem scale						
	2	4	8	10	20	25	40	50	100
300	0.009689	0.029159	0.072545	0.021274	0.2777	0.019224	0.016596	0.421405	0.026158
500	0.028589	0.02486	0.081242	0.101555	0.12983	0.137295	0.597819	0.038831	0.024975
800	0.085988	0.071942	0.084657	0.088311	0.249817	0.299435	0.553757	0.212507	0.875186
1000	0.124405	0.141768	0.097179	0.122693	0.122693	0.211778	0.073604	0.311518	0.297044
3000	0.907231	0.798948	0.655163	0.777543	0.622842	0.475584	0.7569	0.609269	1.685077
5000	2.594188	2.444682	1.928874	1.80159	1.492073	1.528655	1.250693	1.252454	1.812151
8000	6.909438	6.106821	4.711582	4.279023	2.748813	3.121304	3.010961	2.901538	3.019437
10000	3.019437	8.685925	6.942865	6.604144	4.611702	4.278967	4.263814	3.994809	4.51046

Table 3.2

Here is the result of the mpi static scheduling Mandelbrot:

			static scheduling for mpi with core use and problem scale					
	2	4	8	10	20	25	40	50	100
300	0.011481	0.00212	0.007712	0.041508	0.029848	0.092989	0.085666	0.109258	0.316834
500	0.047616	0.04326	0.030983	0.074468	0.126188	0.217616	0.233553	0.161292	0.236122
800	0.068201	0.087133	0.057885	0.089595	0.073747	0.179218	0.179218	0.313407	0.271278
1000	0.107932	0.09927	0.097901	0.132851	0.132851	0.0785	0.214183	0.243923	0.494428
3000	0.953302	0.917478	0.886655	0.581298	0.604502	0.560687	0.528479	0.809416	1.145455
5000	1.145455	2.261829	1.73237	1.560949	1.134327	1.084836	1.377376	1.377376	1.953886
8000	7.820087	6.4225	4.593568	4.087999	2.626297	2.626297	2.626297	2.559864	2.559864
10000	10.7248	9.39691	7.30686	6.216922	5.022068	3.366214	3.855249	4.686104	7.469493

Table 3.3

For mpi dynamic scheduling Mandelbrot, it is quite different from static scheduling, dynamic scheduling requires one more variable. It’s called task in my document. It’s basically used for splitting the matrix to each task and then execute. To show the result easily, I choose only the result with 400 tasks. Here is the table about the result of the dynamic scheduling:

	dynamic scheduling for mpi with core and problem size and 400 tasks							
	2	4	8	10	20	25	40	50
300	0.02762	0.02912	0.037169	0.028998	0.045108	0.032619	0.031592	0.069139
500	0.04895	0.037316	0.028007	0.016346	0.035663	0.053061	0.051804	0.116082

800	0.227466	0.075104	0.108759	0.034924	0.109406	0.069263	0.069298	0.073494
1000	0.22974	0.101611	0.049155	0.041934	0.085462	0.057145	0.058011	0.063776
3000	2.029629	0.643207	0.466071	0.582616	0.532588	0.709463	1.541146	2.928463
5000	5.470395	1.832775	0.823054	0.730543	0.740718	1.093472	1.969538	4.005194
8000	12.9722	4.463091	2.08525	1.737901	1.205785	2.824345	2.091846	2.559735
10000	19.56752	7.051574	3.194672	2.653058	1.722656	1.954442	2.623371	7.16715

Table 3.4

To show the result more vividly, in the performance analysis, I will show more comparing result and trends.

Performance analysis

For a single process, executing the size of the problem will be:

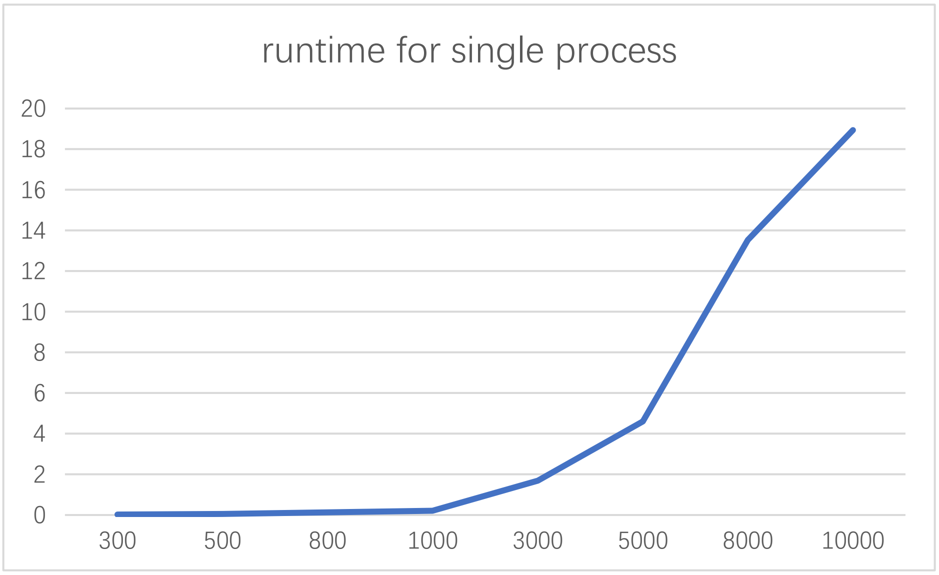

Figure 4.1

From figure 4.1, it is easy to obvious that with the rise of the problem scale, the running time rise up very fast.

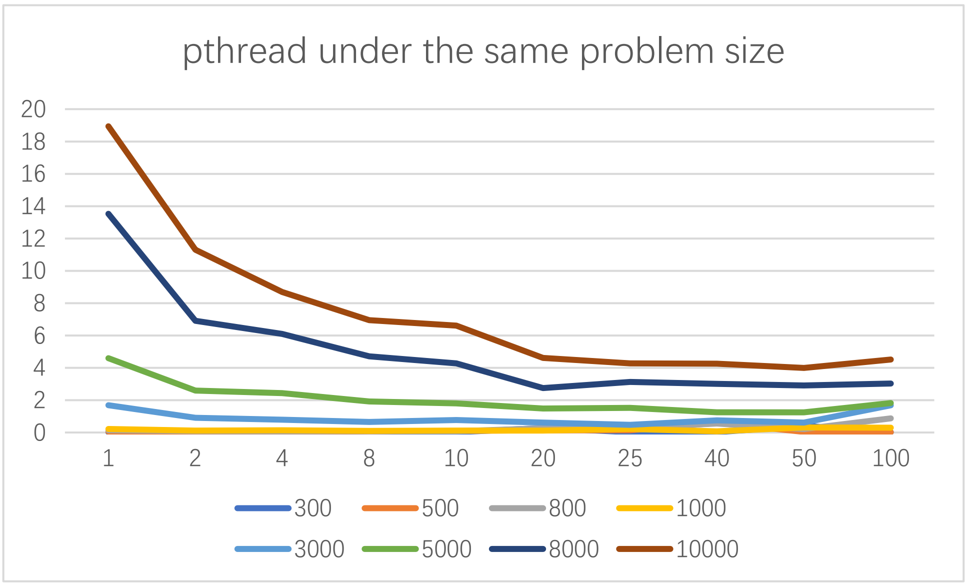

Figure 4.2

Different from a single process, using pthread in this problem will efficiently low down the computing time comparing to single process running. Also, the pthread method has a constraint on this problem and it should be the same on every problem. The sharing area of pthread is not big enough, we can access finite number of cores in a server using pthread. Therefore, from the graph in figure 4.2 we can easily found out that when the number of threads exceed 20. The computing will not get lowed down efficiently. In the opposite, because of the switch between threads, the computing time will get rose up a little bit.

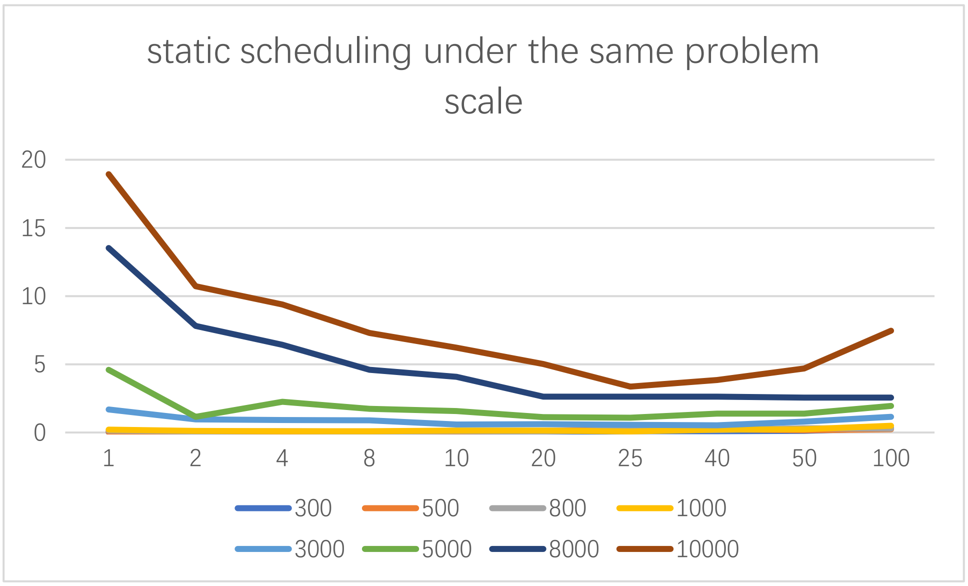

Figure 4.3

We can observe from figure 4.3 that mpi static scheduling method have the same effect on this problem, they are pretty much the same when the problem size is not big enough. However, when the problem size goes up, the mpi static scheduling can show its powerful point. Mpi can easily spot out the globally lowest running time comparing to pthread. Because the mpi can message outside the cluster, it gives the problem better ability on horizontal expand.

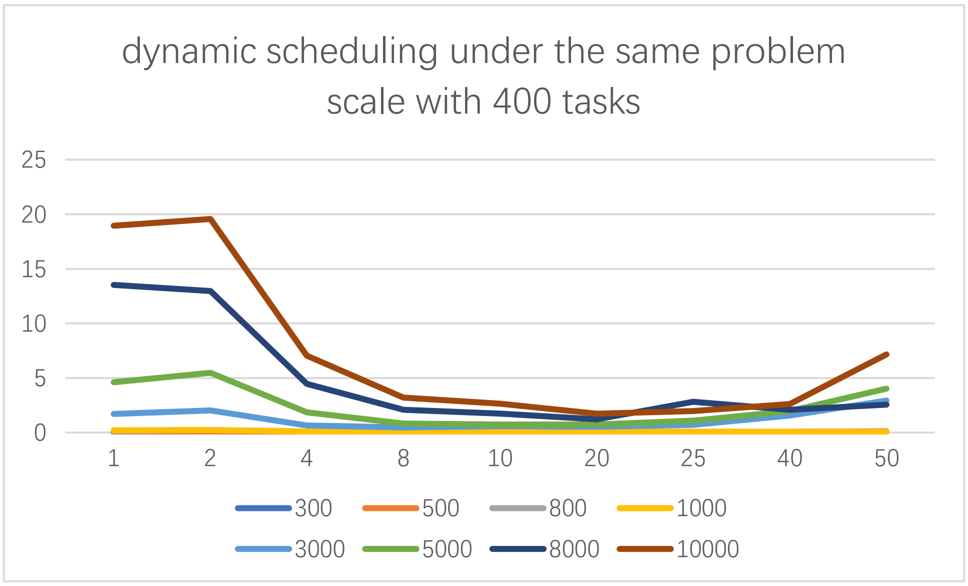

Figure 4.4

For the dynamic scheduling, using two cores spends more running time comparing to using only one core. It is because dynamic scheduling uses one core for arranging tasks for other processes. The communication time added to the computing time makes it costs even more run time comparing to the normal Mandelbrot program. However, when I use more cores, the performance of the dynamic scheduling is outstanding. Comparing to static scheduling, it plots much more beautiful graph.

When I do a comparison between pthread static scheduling， mpi static scheduling and the dynamic scheduling . Following shows the comparison:

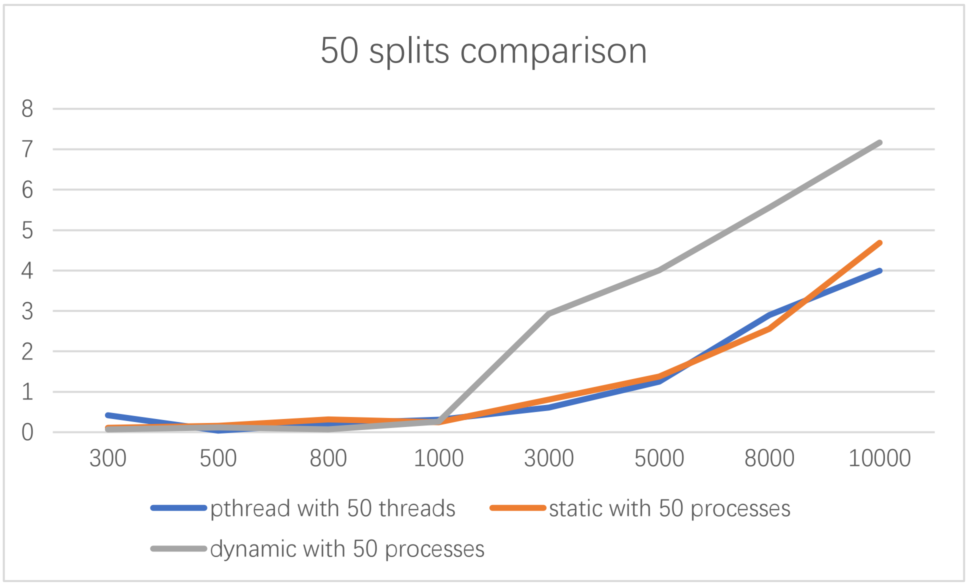

Figure 4.5

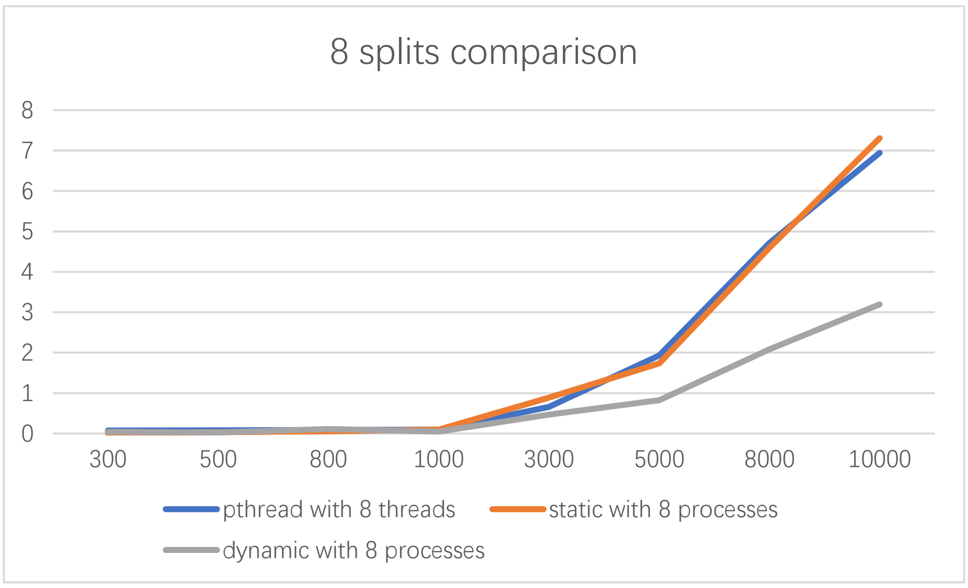

Figure 4.6

In my first impression, mpi will run much slower than pthread when I use less cores and run much faster than pthread when I use more cores. However, from the figure 4.5 and figure 4.6, I found out that pthread version performs similar functionality comparing to mpi static scheduling. Because of the differences of the cluster, this machine has a large cluster with 32 CPU and allow threads for 64 hyper-thread. Therefore, the machine shows similar functionality with pthread and mpi static scheduling. In my assumption, pthread will face its bottle neck when the threads used exceed the number of cores in a cluster.

Interestingly, from these two figures, we can find that dynamic scheduling is pretty powerful when there are only 8 processors but perform quite wired when there are 50 processors. In my opinion, I think dynamic scheduling has more powerful function when there are not enough processes but huge amount of works. According to my assumptions and observation from the figures above, running with too many cores will make there cost too much running time spreading jobs from master process to the slave process, the runtime goes up insanely when there are too many processes and enough blocks.

But when it is under the same problem scale, here shows the comparison:

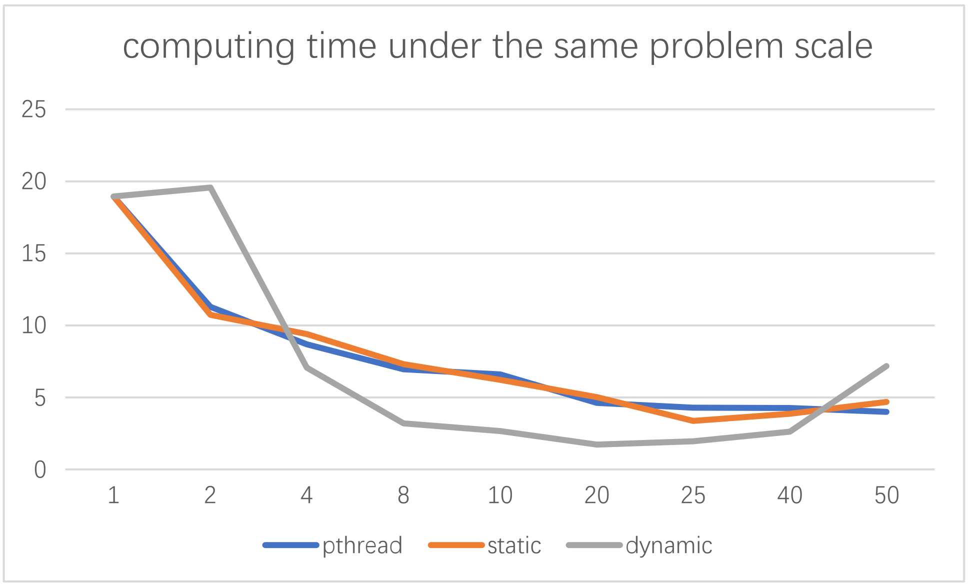

Figure 4.7

This graph shows a very clear comparison among these three kinds of problem solutions. Very interestingly, dynamic scheduling shows a line that plots better running time when the number of cores used are in the middle. When the number of cores used exceed some point, mpi static scheduling has better performance comparing to the dynamic scheduling. Running in this machine, due to the reason of the cluster size, pthread should perform well when there are less threads. Because of the cluster, the pthread method has almost equally performance comparing to the static scheduling. In my opinion, in a small cluster, the pthread will show bad performance when the number of threads exceed the threshold.

Also, I tested the mpi dynamic scheduling with 200 tasks, it has similar outcome comparing to using 400 tasks. The difference is that the range of the low run time changed to another area. As long as we can find the good performance range without machine limitation, dynamic scheduling should be most powerful methods out the three methods.

Conclusion

The conclusion for scheduling is pretty much like the conclusion we can get from the first assignment. No matter for pthread, dynamic scheduling or static scheduling, the best run time shows itself at some point somewhere in the middle. The proper cores used is arising with the scale of the problem. Importantly, when the problem scale is not so big, using pthread will be a better choice rather than mpi. The communication cost is very high when you can run it locally in one cluster. Also, amazingly, comparing to static scheduling, dynamic scheduling does save a lot of time, if you have enough cores to run the program. Static scheduling will be the best choice. If we can find the ratio with the proper number of cores used and the problem size, dynamic scheduling will be a good solution. However, if you have only a few cores, using pthread will be a better choice.

Experience

In the future, for problem size that is not big enough, I will choose to use thread to solve the problem rather than using the message passing. It wastes too much time comparing with thread computing. Also, if the good performance shows with some given problem size and a certain area, I would like to use dynamic scheduling rather than static scheduling when I have enough cores to run the program. Even if one of the process only distributed works, dynamic scheduling saves a lot of time comparing to static scheduling. But if the core numbers exceed some point, currently owned cores cannot support us to find the optimal range in dynamic scheduling, static scheduling will be a better choice.