Introduction to Parallel Computing—Homework 2

王立友

103033121

Implementation:

I used dynamic scheduling and OpenMP to parallelize the program. I separate the pixels into different rows. Each row equals to the width of the picture. In master process, it pushes the slave nodes and the tasks into queue. The task number is equal to the height of the picture.

```
    // Push all slave nodes into the idlequeue
    for(int i=1;i<world_size;++i){
        idlequeue.push(i);
    }

    // Push tasks into the taskqueue
    for(int i=0;i<(height*width);i+=width){
            taskqueue.push(i);
    }
```

As the method we did in lab2, the master will stuck in a while loop when there is task not finished. When there is task coming and the idle queue is not empty, a slave will be taken out to do the task. Master will send the starting pixel to the slave.

```
// Distribute new task to the idle node
while(taskqueue.size() > 0 && idlequeue.size() > 0)
{
    int pixelNum = taskqueue.front();
    taskqueue.pop();
    int node = idlequeue.front();
    idlequeue.pop();
    MPI_Isend(&pixelNum, 1, MPI_INT, node, 0,MPI_COMM_WORLD, &request);
    //MPI_Isend(&msg, 1, MPI_MSGTYPE, node, 0,MPI_COMM_WORLD, &request);
}
// TODO: ..... Nonblocking check for whether any nodes have finished
```

Then Master will keep listening if there is slave finishes.

```
// TODO: ..... Nonblocking check for whether any nodes have finished the previ
//MPI_Iprobe( ????? );
MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);

if(flag)
{
```

If there is slave sending message, master receives the message. The message is an array containing the colored pixels of that row. Master assigns the pixel colors into the output image array. In this part, I use OpenMP to parallelize the for loop.

```
        //tag == COMPLETE means there has no any points outside the edge
        if(status.MPI_TAG == COMPLETE)
        {
            // Push the node back into the idlequeue
            int node = status.MPI_SOURCE;
            idlequeue.push(node);
            #pragma omp parallel for num_threads(num_threads)
            for(int k=0; k<width; k++)
            {
                int pixelNum = (int)msg[4*k + 3];;
                int i = pixelNum / width;
                int j = pixelNum % width;
                image[i][4*j + 0] = (unsigned char)msg[4*k + 0];  //r
                image[i][4*j + 1] = (unsigned char)msg[4*k + 1];  //g
                image[i][4*j + 2] = (unsigned char)msg[4*k + 2];  //b
                image[i][4*j + 3] = 255;  //a

                current_pixel++;
                //print progress
```

If all the tasks and slave have been completed, master mark the flag complete and break the while loop.

```
    // All tasks and slave nodes have been completed
    if(taskqueue.size() == 0 && idlequeue.size() == world_size-1)
    {
        complete = 1;
    }
}
```

It also send the “complete” message to slaves inform them the jobs completes.

```
    // Send "complete" signal to all the slave nodes
    for(int i=1;i<world_size;++i)
    {
        int msg;
        MPI_Isend(&msg, 1, MPI_INT, i,COMPLETE,MPI_COMM_WORLD, &request);
        //.....................
    }

}
```

In slave, it enter a while loop at the beginning. In each run in while loop, it will listen if master send jobs to it.

```
    int complete = 0;
    while(!complete)
    {
        // TODO: ............. Nonblocking test for a message from master node
        //MPI_Iprobe(????);
        //MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
        MPI_Iprobe(0, MPI_ANY_TAG , MPI_COMM_WORLD, &flag, &status);
        //.....................
        if(flag)
        {
```

If there is a “complete” flag, it will break the loop. Otherwise it take the job.

```
if(flag)
{
    // if receive COMPLETE tag then break the while loop
    if(status.MPI_TAG == COMPLETE)
        break;

    // TODO: ..... Recv task from master node
    // MPI_Recv(&msg, 1, MPI_MSGTYPE, ?????);
    MPI_Recv(&pixelNum, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    //.....
```

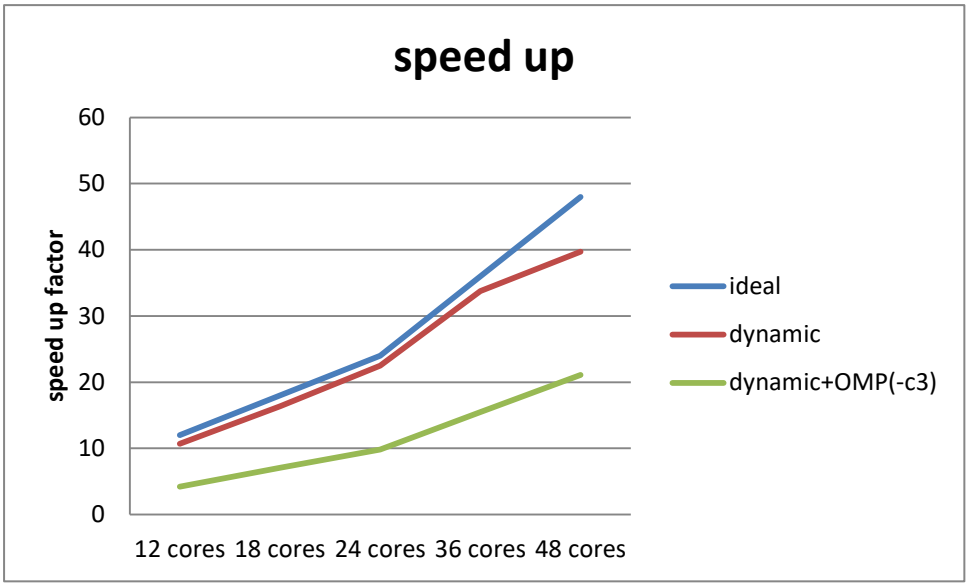
I used OpenMP to parallelize the for loop of the row pixels. The following is as same as the origin sequential code.

```
#pragma omp parallel for num_threads(num_threads)
for(int j=0; j<width; ++j)
{
    vec4 fcol(0.);  //final color (RGBA 0 ~ 1)

    //anti aliasing
    for(int m=0;m<AA;++m){
        for(int n=0;n<AA;++n){
```

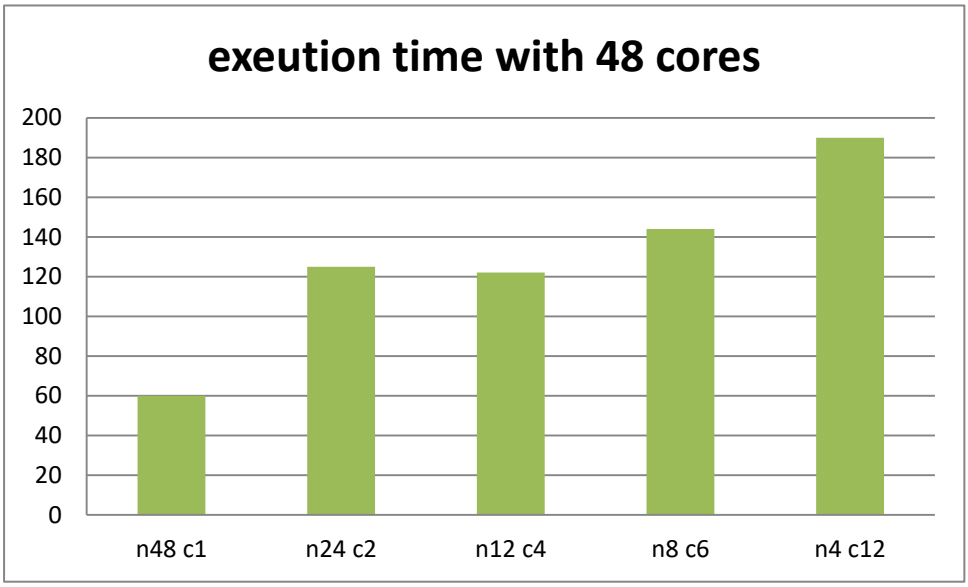
After finishing one pixel, it stores the colors and the corresponding pixel information to the msg array. After finishing the whole row of pixels, it sends the array to master.

```
            }
        }//for(int m=0;m<AA;++m)

        fcol /= (double)(AA*AA);
        //convert double (0~1) to unsigned char (0~255)
        fcol *= 255.0;

        msg[4*j + 0] = fcol.r;
        msg[4*j + 1] = fcol.g;
        msg[4*j + 2] = fcol.b;
        msg[4*j + 3] = (double)(pixelNum+j);
    }//for
    // TODO: ..... Send result back to the master node
    MPI_Send(msg, 4*width, MPI_DOUBLE, 0, COMPLETE, MPI_COMM_WORLD);
    //.....
```

Analysis (all the following is only using case00)

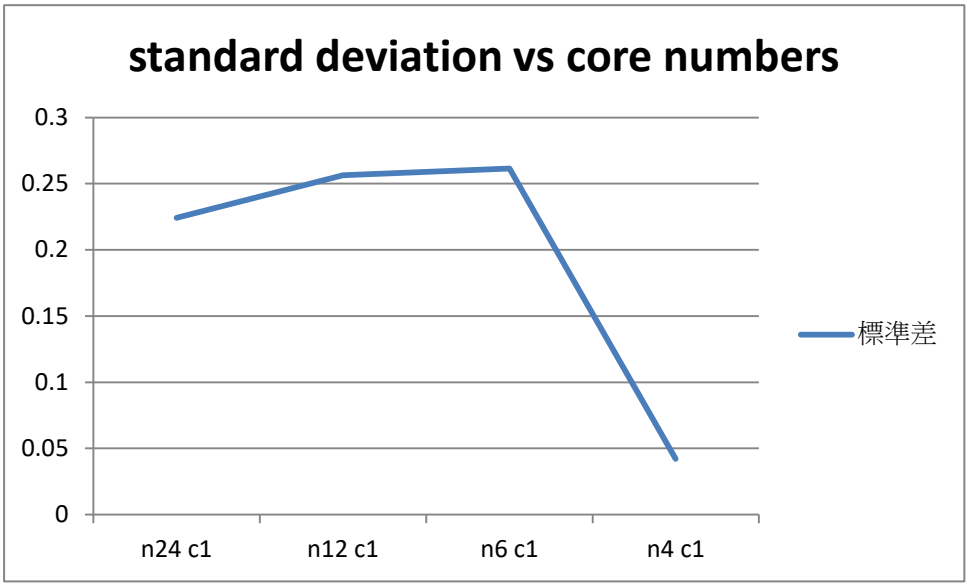


The above figure is comparing the speed up chart. The blue line is the ideal speed up model. The sequential code is running 45 minutes. The red line is using only one thread per MPI process, that is, the OpenMP part is no effect. The green line is combining dynamic scheduling and openMP, with constant thread number (-c 3). Since the server is busy when I test the green line, the time may be a little slower.

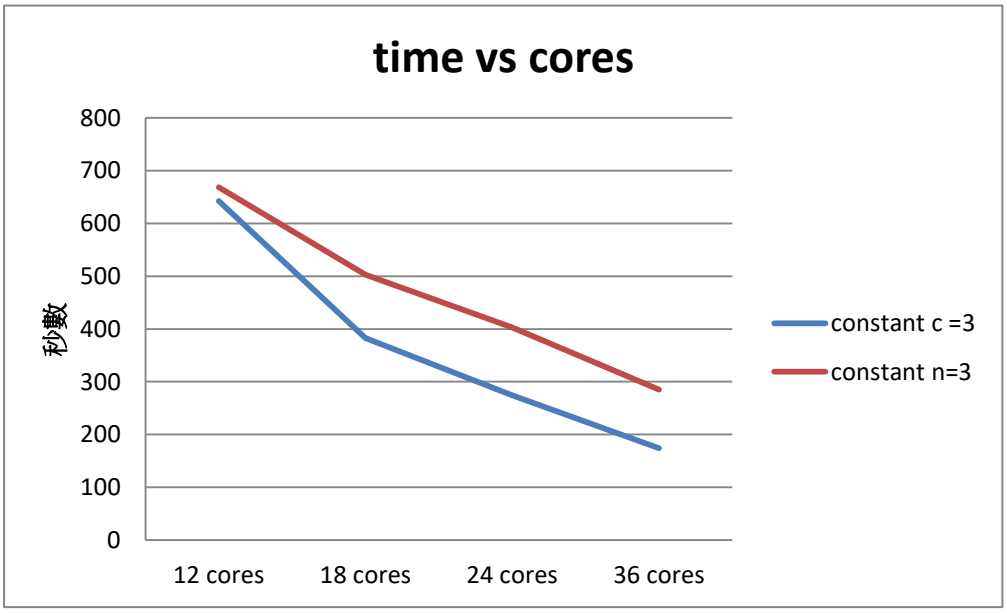


The above figure is the execution time (seconds) using total 48 cores with different process and thread numbers. Showing that when running with 48 MPI processes and 1 thread has the best performance. On the other hand, if we add up the thread number to 12 and reduce the processes number to 4, the performance is one third as

the best one. This figure tells us that the parallelizing effect of MPI is better than OpenMP in my code. However, it doesn't mean that OpenMP have no effect. Adding OpenMP method can actually reduce time comparing to my origin version which simply used MPI.



The above figure present the standard deviation of the worker execution time using MPI_Wtime() when running with different number of MPI processed. It is surprised that almost all the execution time of slaves is same and have little standard deviation. However, this doesn't mean that the load balance distributed in a good way. It is possible that the load balance is good. But there is also a possibility that all the workers are together slow down.



The above figure compares the performance with one parameter constant. The red line is using 3 MPI nodes combining with different amount of threads. From left to right, they are 4, 6, 8 and 12 threads, respectively. The blue line is using constant thread number with different MPI nodes. From left to right, they are 4, 6, 8 and 12 MPI nodes, respectively. This figure tells that the performance of my program is mainly dominated by the MPI parallelizing method. Using OpenMP parallelizing is not as effective as using MPI.

Bonus

In the bonus part, I try several arguments of the colors and the powers. I found that when the power is lager, the shape is less irregular. I select power number 50 and found it look most beautiful. It is look like a sphere shape and the color is not that wired. The background color is also modified, to make it look more harmonious. The execution method is same as the normal one.

Conclusion

At the beginning, this assignment looks very difficult since I can hardly understand how the mandelbulb woks. It took me a lot of time trying to figuring how the code work but I still can't get it. Thankfully, the TA tells me that I can modify it without understand how it work inside. And once I understand the code of lab2, I can start working on this assignment. It take me 90% time to think how to code and 10% time to acutely code. The most achieving part is figuring out that it is not as hard as it look like.