

CS542200 Parallel Programming

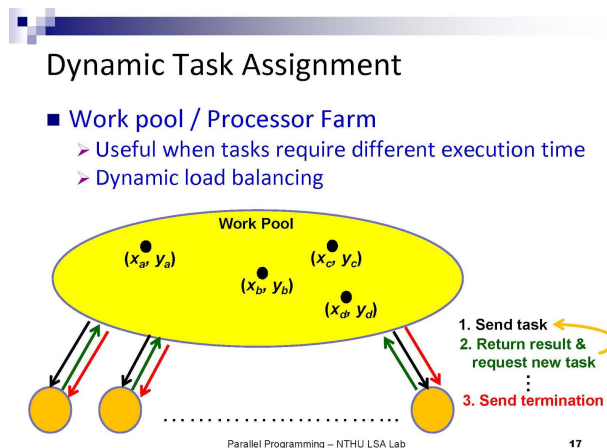
Homework 2: Mandelbrot Set Report

106030009 葉蓁

1. Implementation

(1) Hybrid version

This version I use MPI to implement dynamic task assignment as the figure below:



One MPI process serves as master and others serve as slaves. I pick **rank 0 as master**. The master process controls the task assignment and the slave processes are responsible for the computations.

The **basic unit of one task is one row** of the input data.

In each slave process, I use OpenMP to create threads that compute different pixels in a row. For the `schedule()` clause, I've tried different `CHUNK_SIZE` and found that **`CHUNK_SIZE == 1`** has the best performance.

```
146 // Master: rank 0 process
147 > void Assign_Process(){ ...
231
232 // Slave: other processes
233 > void Slave_Calculate(){ ...
356
357
358
359 > int main(int argc, char** argv) { ...
587
588
```

```

// Load Balancing
////////// Dynamic workload //////////

if(mpi_size == 1){ ...
else{
    // Load Balancing
    ////////// Dynamic workload //////////
    if(mpi_rank == 0){
        Assign_Process();
    }else{
        Slave_Calculate();
    }
}
}

```

Note that I should be careful of a **special case: -n1** when there's only ONE process. The master (rank 0) cannot send to itself, so I use change the mode to the early version I wrote:

Simply use MPI processes to get a constant number of rows for each process. Namely, row_per_data. Then, for each process, use OpenMP to create threads to accelerate the computations of each pixel in a row.

(2) Pthread version:

This version I use Pthread to meet the specification.

Since it would only have **ONE process** in this version, I create "height" number of threads so that each thread needs to calculate only one row. After they are finished, the main thread needs to wait for all threads then it would write the output image.

```

int num_threads = height;
pthread_t threads[num_threads]; // a threads_array.
thread_info thread_INFO[num_threads];

_iters = _mm_set_epi64x((long long)iters, (long long)iters);

/* mandelbrot set */
int rc;
for (int j = 0; j < height; ++j) { // Go through every pixel from y-axis view. (
    // create thread
    thread_INFO[j].thread_id = j;
    rc = pthread_create(&threads[j], NULL, Calculate, (void*)&thread_INFO[j]);
    // rc = pthread_create(&threads[t], NULL, hello, (void*)&ID[t]); // ID would
    if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
    }
}
}

```

(3) Vectorization: (Failure description)

I've tried vectorization on different versions of my implementation above.

The idea I used: for each process in its row, calculate two pixels as a pair at a time.

One row



However, I've encountered many problems.

First, I think maybe it is the problem of the MPI_Send and MPI_Recv, so I switch to the early version (MPI on Gathrev in the end, other times each process just does computations.)

However, when I verified my answer by `srun -c4 -n1 ./hw2b fast01.png`

It tooks 220 seconds to complete fast01.txt ! This is quite weird since the original non-vectorized version just took 1.67 seconds. After discussing with TA and other students, the reason might be I use too much unnecessary vectorized api in my code. So **I move the 'Declaration of __m128d , __m128i' out to the global region.** In the computation while-loop, just do assignment using this set of same variables. After this modification, I finally get the computation time faster!

However, another problem generates. **The correctness of my program is not 100% correct!**

I got 97.5% on fast01.txt. Worse yet, for slow01.txt I only got 81%. Clearly, something might be wrong in my program. After checking countless times, I finally found that the pixels I packed to the 128bit register were in the wrong sequence. I should pack as : {pixel[i+1] , pixel[i] } instead of my original code {pixel[i], pixel[i+1]}. FInally, I get 100 % correctness!

After running hw2b-judge:

Unfortunately, the **slow01 testcase got TIME_LIMIT_EXCEED problem!**

I used srun to test slow01 and found that it tooks 452 seconds to complete slow01. Now I think it's time I decreased the basic unit of task assigned to each process! I should try to make the CHUNK_SIZE smaller than a row. Maybe half row would be better!

After changing the Task_size unit from ONE row to HALF row: The performance is slower 2 seconds QQ. Note that this partition method should be even careful at boundaries.

Adding Vectorization in this version (Half_row per task) didn't help much, either. I think it might be that my implementation of vectorization is not that good enough.

(4) Vectorization (Success version)

I implemented vectorization based on the dynamic work pool version, so each slave process requests one task at a time. The basic unit of a task is a row.

(I've tried shorter task units like half row, however, the performance is worse.)

To reduce execution time and increase scalability : Need to minimize the number of vectorized operations. Avoid unnecessary vectorized registers and instructions.

For me, I think the most critical part is how to determine whether to break out of the while-loop calculating mandelbrot set.

I use `_mm_and_si128()` to bitwise AND the result from `_mm_cmpgt_epi64(_iters, _repeats)` and the result from `_mm_cmplt_pd(_len_squared, _2four)`

Here `_2four == _mm_set_pd(4, 4);`

If the bitwise AND result is equal to 0, it means that both pixels are done calculation and should break from this while-loop.

Another crucial section of vectorization is : How to know whether the `_repeats` should add 1 by using the minimum number of instructions?

After thinking about the solution for about 5 days, I finally use another bitwise AND but this time I use :

```
while(1){
    cmp_length_result = _mm_castpd_si128(_mm_cmplt_pd(_length_squared, _mm_set_pd((double)4, (double)4)));
    and_result = _mm_and_si128(_mm_cmpgt_epi64(_iters, _repeats), cmp_length_result);

    if(_mm_movemask_epi8(and_result) == 0) break;
    _repeats = _mm_add_epi64(_repeats, _mm_and_si128(cmp_length_result, _mm_set_epi64x(1,1)));
}
```

If `cmp_length_result` is correct, it would be

`0xFFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFFFF`, and AND `0x1 | 0x1` would be

`0x1 | 0x1`, which should be added to `_repeats`. Perfect!

```
_repeats = _mm_add_epi64(_repeats, _mm_and_si128(cmp_length_result, _mm_set_epi64x(1,1)));
```

(5) Record of my program improvement:

hw2a: Pthread version

After Implementing Vectorization, the performance improved from 769.93 to 406.49 !!

```
strict28.txt    5.68    accepted
strict29.txt   12.00    accepted
strict27.txt   24.83    accepted
strict30.txt    4.63    accepted
strict32.txt   11.54    accepted
strict31.txt   28.79    accepted
strict33.txt   34.14    accepted
strict35.txt   20.42    accepted
strict34.txt   44.93    accepted
strict36.txt   10.29    accepted
Removing temporary directory /home/pp20/pp20s35/.judge.730297554
Scoreboard: updated {68 769.93} --> {68 406.49}
[pp20s35@apollo31 hw2]$
```

hw2b: Hybrid version

I've tried two methods for hybrid version implementation.

- MPI processes take a **static number of tasks**, then use OpenMP to create threads for parallel computations when computing mandelbrot set.

```
155     int local_start_j = height/mpi_rank; // j = rank
156     int end_j;
157
158     int offset; // offset of row
159     int row_per_proc = height/mpi_size;
160     int remainder = height%mpi_size;
161
162
163     if(mpi_rank==0){
164         offset = row_per_proc*mpi_rank; //sizeof(MPI_FLOAT)*
165         end_j = offset + row_per_proc + remainder;
166     }
167     else{
168         offset = (row_per_proc*mpi_rank+remainder); // *sizeof(MPI_FLOAT)
169         end_j = offset + row_per_proc;
170     }
171
```



```

// use OpenMP accelerate
#pragma omp parallel num_threads(end_j-offset) // threads for each row (row_per_proc)
{
    for(int j=offset; j< end_j ; j++){
        // printf("rank %d do %d row\n",mpi_rank, j);

        double y0 = lower + j * ((upper - lower) / height);
        // For each pixel
        #pragma omp for schedule(dynamic, CHUNKSIZE)
        for (int i = 0; i < width; ++i) {
            int thread_id = omp_get_thread_num();
            int num_threads = omp_get_num_threads();
            // printf("thread %d / %d\n", thread_id, num_threads);
            double x0 = left + i * ((right - left) / width); // Along x-axis (Real-axis)

            int repeats = 0; // number of iterations. <= iters.
            double x = 0;    // x value (Real part)
            double y = 0;    // y value (Imaginary part)
            double length_squared = 0; // |Z|
            while (repeats < iters && length_squared < 4) {
                double temp = x * x - y * y + x0; // next z.real    c.real = x0
                y = 2 * x * y + y0;                // next z.imag    c.imag = y0.
                x = temp;
                length_squared = x * x + y * y;    // compute |Znext|
                ++repeats;                        // one more iteration
            }
            image[j * width + i] = repeats;
        }
    }
}

```

Use MPI_Gatherv to receive answers from each process and rank 0 output image.

```

if(mpi_rank == 0){
    output_image = (int*)malloc(width * height * sizeof(int));
    // int rcv_cnt_array[mpi_size];
    for(int i=0; i<mpi_size; i++){
        if(i==0){ // rank 0
            rcv_cnt_array[i] = (row_per_proc+remainder)*width;
            displ_array[i] = 0;
        }
        else { // other
            rcv_cnt_array[i] = row_per_proc*width; // number of pixels
            displ_array[i] = (row_per_proc*i + remainder)*width; // offset position of output_image to store items.
        }
    }
    assert(output_image);
    // MPI_Gatherv( sendarray, 1, stype, rbuf, rcounts, displacement at rcv buf, MPI_INT, root, comm);
    MPI_Gatherv(image, (row_per_proc+remainder)*width, MPI_INT, output_image, rcv_cnt_array, displ_array, MPI_INT, 0, MPI_COMM_WORLD);
}
else{
    // int* nullspace = (int*)malloc(2*sizeof(int)); // no use.
    // //rcv count // displacement array
    MPI_Gatherv(image+(offset*width), row_per_proc*width, MPI_INT, NULL, rcv_cnt_array, displ_array, MPI_INT, 0, MPI_COMM_WORLD);
}

if(mpi_rank==0){
    write_png(filename, iters, width, height, output_image);
    free(output_image);
    printf("Process 0 done writing image.\n");
}
free(image);

```

- b. **Dynamic Task Assignment** using MPI communication and use OpenMP to create threads for parallel computations when computing mandelbrot set.
 Note: should struct MPI_Datatype in this version.

```

111 // Received from each process.
112 ✓ typedef struct MPI_send_data{
113     int row_number;
114     int tag;
115 }Send_data;
116
117 // Received from each process.
118 ✓ typedef struct MPI_recv_data{
119     int row_number;
120     int rank;
121     double recv_ans[WIDTH_MAX_SIZE];
122 }Recv_data;
123

```

Send_data is for the master to send data.

Recv_data is for slaves to return results.

```

// Master: rank 0 process
> void Assign_Process(){ ...

// Slave: other processes
> void Slave_Calculate(){ ...

> int main(int argc, char** argv) { ...

✓ /// This version: want to try using bitw
  /// while(1): while( _mm_and_pd( _mm_cm
  |

```

Assign_Process follows the pseudo code taught in class.

Slave_Calculate computes mandelbrot set and return results.

Details are not shown here since the codes are a little bit long and complex.

You can refer to my hw2b.cc to see more information.

Performance history:

- (1) Before Optimization: 902.62 seconds

pp20s35	7	68	902.62	3.17	3.98	2.47	3.04	1.52	2.22	2.12	2.17
---------	---	----	--------	------	------	------	------	------	------	------	------

- (2) After 1st optimization: 755.52 seconds
(use **schedule(dynamic, CHUNKSIZE=10)**)

```
strict27.txt 27.55 accepted
strict30.txt 11.45 accepted
strict29.txt 22.98 accepted
strict32.txt 18.12 accepted
strict31.txt 60.44 accepted
strict33.txt 46.75 accepted
strict35.txt 31.41 accepted
strict34.txt 51.67 accepted
strict36.txt 17.41 accepted
Removing temporary directory /home/pp20/pp20s35/.judge.844668945
Scoreboard: updated {68 902.62} --> {68 755.52}
[pp20s35@apollo31 hw2]$
```

- (3) 2nd~4th optimization:
Focusing on tuning the parameters in `schedule()` directive, but no significant progress.
- (4) After 5th optimization: Use MPI & OpenMP by handcraft a dynamic task assignment from scratch. Improve 76 seconds! So happy !!

```
strict31.txt 40.22 accepted
strict33.txt 47.89 accepted
strict35.txt 28.18 accepted
strict34.txt 60.96 accepted
strict36.txt 13.85 accepted
Removing temporary directory /home/pp20/pp20s35/.judge.467385637
Scoreboard: updated {68 708.74} --> {68 632.94}
[pp20s35@apollo31 hw2]$
```

However, some test cases were bottleneck and should be improved. For example, `slow01` was slower than the previous implementation. Also, `strict 31 ~ 36` still performed slowly.

Original `slow1` : 16.46 Now `slow1` : 41.27

- (5) Set `CHUNK_SIZE` to 1 : improve from 632 to 603 seconds !

```
strict29.txt 10.45 accepted
strict27.txt 35.35 accepted
strict30.txt 6.33 accepted
strict32.txt 15.40 accepted
strict31.txt 40.21 accepted
strict33.txt 47.99 accepted
strict35.txt 28.38 accepted
strict34.txt 61.02 accepted
strict36.txt 13.95 accepted
Removing temporary directory /home/pp20/pp20s35/.judge.176252295
Scoreboard: updated {68 632.94} --> {68 603.34}
[pp20s35@apollo31 hw2]$
```

- (6) Vectorization :
I've tried vectorization on two versions: `MPI_Send_Recv` version and `MPI_static_number_of_rows` version.

For MPI_Send_Recv version: A slave process would receive one row from master at a time, and calculate two pixels at the same time, then send the results of this row back to master. Note that I should handle the boundary conditions when the width is not even and would have remainder when being divided by two.

However, for one task: slow01.txt, I get TIME_LIMIT_EXCEED and I'm still figuring out how to solve this problem.

(7) One slave take HALF_ROW at a time:

The performance didn't improve significantly. (603 to 604 seconds, even worse.)

```
strict30.txt    6.38    accepted
strict32.txt    15.40   accepted
strict31.txt    40.27   accepted
strict33.txt    47.97   accepted
strict35.txt    28.29   accepted
strict34.txt    61.11   accepted
strict36.txt    13.95   accepted
Removing temporary directory /home/pp20/pp20s35/.judge.656786801
Scoreboard: not updating {68 603.34} -x-> {68 604.80}
[pp20s35@apollo31 hw2]$
```

(8) One slave take HALF_ROW and Vectorized:

Took 880 seconds and gets wrong results on 7 test cases, terrible!

Get 7 wrong results(or time limit exceeded).

```
strict30.txt    11.19   accepted
strict29.txt    21.27   accepted
strict32.txt    21.22   accepted
strict31.txt    58.66   accepted
strict33.txt    47.62   accepted
strict35.txt    29.44   accepted
strict36.txt    20.72   accepted
strict34.txt    83.57   accepted
Removing temporary directory /home/pp20/pp20s35/.judge.199864803
Scoreboard: not updating {68 603.34} -x-> {61 880.60}
[pp20s35@apollo31 hw2]$
```

(9) Vectorization Ver. N!!!! (Countless versions... QQ)

```
strict29.txt    13.35   accepted
strict27.txt    27.14   accepted
strict30.txt    6.23    accepted
strict32.txt    12.65   accepted
strict31.txt    30.89   accepted
strict33.txt    36.46   accepted
strict35.txt    22.17   accepted
strict34.txt    46.48   accepted
strict36.txt    11.74   accepted
Removing temporary directory /home/pp20/pp20s35/.judge.946041953
Scoreboard: updated {68 602.52} --> {68 502.41}
[pp20s35@apollo31 hw2]$
```

```
strict34.txt 46.48 accepted
strict36.txt 11.74 accepted
Removing temporary directory /home/pp20/pp20s35/.judge.946041953
Scoreboard: updated {68 602.52} --> {68 502.41}
[pp20s35@apollo31 hw2]$
```

Finally succeeded in improving performance!!! OMG

Although it only **improves 100 seconds**, it meant a lot to me!

(I'm sorry that I am a really slow and stupid student, but I've really tried my best.)

I think the reason that the performance didn't improve by a factor of 2 is that the communication time of MPI during the work pool assignment I wrote and the time of load / store, bitwise AND, packing 128-bit registers, etc.

```
slow02.txt 6.41 accepted
slow03.txt 6.13 accepted
slow04.txt 8.53 accepted
slow05.txt 6.08 accepted
slow01.txt 28.00 accepted
slow06.txt 6.13 accepted
slow08.txt 6.53 accepted
slow07.txt 6.03 accepted
slow09.txt 6.18 accepted
slow10.txt 6.33 accepted
slow11.txt 6.18 accepted
slow12.txt 6.43 accepted
slow13.txt 6.23 accepted
slow14.txt 6.03 accepted
slow15.txt 7.13 accepted
strict01.txt 1.02 accepted
```

Note that slow01.txt : The time increased! From 19 seconds to 28 seconds. (Bad)

But on average, the total running time decreased! This is nice.

Especially for the last strict test cases:

```
strict28.txt 6.69 accepted
strict29.txt 13.35 accepted
strict27.txt 27.14 accepted
strict30.txt 6.23 accepted
strict32.txt 12.65 accepted
strict31.txt 30.89 accepted
strict33.txt 36.46 accepted
strict35.txt 22.17 accepted
```

Original time: (See 31, 33 and 34.txt)

```
strict30.txt 6.38 accepted
strict32.txt 15.40 accepted
strict31.txt 40.27 accepted
strict33.txt 47.97 accepted
strict35.txt 28.29 accepted
strict34.txt 61.11 accepted
strict36.txt 13.95 accepted
```

(10) Final version :

After deleting unnecessary registers and make the code neat,
the performance increased to 368.90 seconds! I can't believe QQ.

Note that the running time of last strict test cases decreased significantly!

(Strict34.txt : From original 61.11 seconds -> 32.19 seconds (Almost half time))

```
strict28.txt    5.63  accepted
strict27.txt   18.74  accepted
strict29.txt    9.88  accepted
strict30.txt    5.08  accepted
strict31.txt   21.46  accepted
strict32.txt    9.84  accepted
strict33.txt   25.18  accepted
strict35.txt   15.81  accepted
strict34.txt   32.19  accepted
strict36.txt    8.89  accepted
Removing temporary directory /home/pp20/pp20s35/.judge.731813504
Scoreboard: updated {68 502.41} --> {68 368.90}
[pp20s35@apollo31 hw2]$
```

(11) Delete some unnecessary registers and initialization instructions:

```
strict31.txt   21.32  accepted
strict33.txt   25.18  accepted
strict35.txt   15.75  accepted
strict34.txt   32.19  accepted
strict36.txt    8.89  accepted
Removing temporary directory /home/pp20/pp20s35/.judge.393391802
Scoreboard: updated {68 368.90} --> {68 366.69}
[pp20s35@apollo31 hw2]$
```

My ranking went from 39 to 15 !! I'm so grateful and happy since I know that I'm really not smart, but I work extremely hard to catch up with the class.

2. Experiment & Analysis

i. Methodology

(a). System Spec

I use Apollo Cluster.

(b). Performance Metrics

For the **Hybrid version**, I use **MPI_Wtime()** and measure the computing time of my program using rank 0 (the master).

I picked **strict34.txt** as the scalability test case.

```
hw2 > testcases > strict34
1 | 1022.11
2
```

It's sequential time:

I use two different environments (N=1 and N=4)

Process size: from 1 to 16 and from 4 to 48

Thread size: from 1 to 12 (different combinations with #processes)

First, I will check whether the answer is correct.

```
exp_strict34_N1n12c1.png exp_strict34_N1n12c1.png
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N1n1c12.png
ok, 100.00% 😊
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N1n4c3.png
ok, 100.00% 😊
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N1n8c1.png
ok, 100.00% 😊
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N1n12c1.png
ok, 100.00% 😊
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N4n4
exp_strict34_N4n48c1.png exp_strict34_N4n4c12.png
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N4n4c12.png
ok, 100.00% 😊
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N4n16c3.png
ok, 100.00% 😊
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N4n32c1.png
ok, 100.00% 😊
[pp20s35@apollo31 hw2]$ hw2-diff testcases/strict34.png Exp/exp_strict34_N4n48c1.png
ok, 100.00% 😊
[pp20s35@apollo31 hw2]$
```

Then, see the recorded time at slurm-ID.out.

For example, for -N1 -n4 -c3 :

```
hw2 > ≡ slurm-2635129.out
1 | Process 1 done calculation
2 | Process 3 done calculation
3 | Process 2 done calculation
4 | Process 0 done writing image.
5 | Took 58.98738 seconds.
6
```

It tooks 58.98738 seconds to complete strict34.txt.

-N1 -n1 -c12 :

```
hw2 > ≡ slurm-2635131.out
1 | Process 0 done writing image.
2 | Took 88.23528 seconds.
3
```

-N1 -n1 -c1: Like Sequential time:

```
hw2 > cat slurm-2652809.out
1 Master Took 501.07886 seconds.
2 Master CPU time: 497.57064 seonds
3 Master Comm time: 0.00000 seonds
4 Master IO time: 3.50822 seonds
5
```

-N1 -n4 -c1:

```
hw2 > cat slurm-2635327.out
1 Took 163.72470 seconds.
2
```

-N1 -n8 -c1 :

```
hw2 > cat slurm-2635128.out
1 Process 1 done calculation
2 Process 3 done calculation
3 Process 6 done calculation
4 Process 4 done calculation
5 Process 5 done calculation
6 Process 2 done calculation
7 Process 7 done calculation
8 Process 0 done writing image.
9 Took 72.25183 seconds.
10
```

-N1 -n12 -c1:

```
hw2 > cat slurm-2635125.out
1 Process 1 done calculation
2 Process 9 done calculation
3 Process 10 done calculation
4 Process 7 done calculation
5 Process 4 done calculation
6 Process 11 done calculation
7 Process 5 done calculation
8 Process 6 done calculation
9 Process 8 done calculation
10 Process 3 done calculation
11 Process 2 done calculation
12 Process 0 done writing image.
13 Took 47.21327 seconds.
14
```


# Computing Nodes	1	Time	Seq time	Speedup
# processes	# Threads per processor	(sec)	(sec)	Speedup
1	1	501.07886	1022.11	1
4	1	163.7247	1022.11	6.242858
8	1	72.25183	1022.11	14.14649
12	1	47.213271	1022.11	21.64879

-N4 -n4 -c12 :

```
hw2 > ≡ slurm-2635160.out
1 | Process 1 done calculation
2 | Process 3 done calculation
3 | Process 2 done calculation
4 | Process 0 done writing image.
5 | Took 17.56524 seconds.
6 |
```

-N4 -n16 -c3:

```
hw2 > ≡ slurm-2635182.out
1 | Process 10 done calculation
2 | Process 13 done calculation
3 | Process 9 done calculation
4 | Process 11 done calculation
5 | Process 14 done calculation
6 | Process 8 done calculation
7 | Process 12 done calculation
8 | Process 15 done calculation
9 | Process 5 done calculation
10 | Process 2 done calculation
11 | Process 3 done calculation
12 | Process 6 done calculation
13 | Process 1 done calculation
14 | Process 7 done calculation
15 | Process 4 done calculation
16 | Process 0 done writing image.
17 | Took 14.79096 seconds.
18 |
```

-N4 -n4 -c1:

```
hw2 > ≡ slurm-2635314.out
1 Took 163.33366 seconds.
2
```

-N4 -n16 -c1:

```
≡ slurm-2635324.out ✕
hw2 > ≡ slurm-2635324.out
1 Took 35.63868 seconds.
2
```

-N4 -n32 -c1:

```
hw2 > ≡ slurm-2635303.out
1 Took 19.38264 seconds.
2
```

-N4 -n48 -c1:

```
hw2 > ≡ slurm-2635304.out
1 Took 14.39892 seconds.
2
```

# Computing Nodes	4	Time	Seq time	Speedup
# processes	# Threads per processor	(sec)	(sec)	Speedup
4	1	163.33366	1022.11	6.257804
16	1	35.63868	1022.11	28.67979
32	1	19.38264	1022.11	52.73327
48	1	14.39892	1022.11	70.98518

For the Pthread version, I use `clock_gettime(CLOCK_MONOTONIC)` to record the start time and end time of my program.

-N1 -n1 -c1

```
hw2 > ≡ slurm-2641510.out  
1  Took 487.35855 second  
2
```

-N1 -n1 -c2

```
hw2 > ≡ slurm-2641509.out  
1  Took 245.13978 second  
2
```

-N1 -n1 -c4

```
hw2 > ≡ slurm-2641507.out  
1  Took 123.57049 second  
2
```

-N1 -n1 -c6

```
hw2 > ≡ slurm-2641506.out  
1  Took 83.87010 second  
2
```

-N1 -n1 -c8

```
hw2 > ≡ slurm-2641505.out  
1  Took 63.69064 second  
2
```

-N1 -n1 -c10

```
hw2 > ≡ slurm-2641504.out
1 Took 51.65350 second
2
```

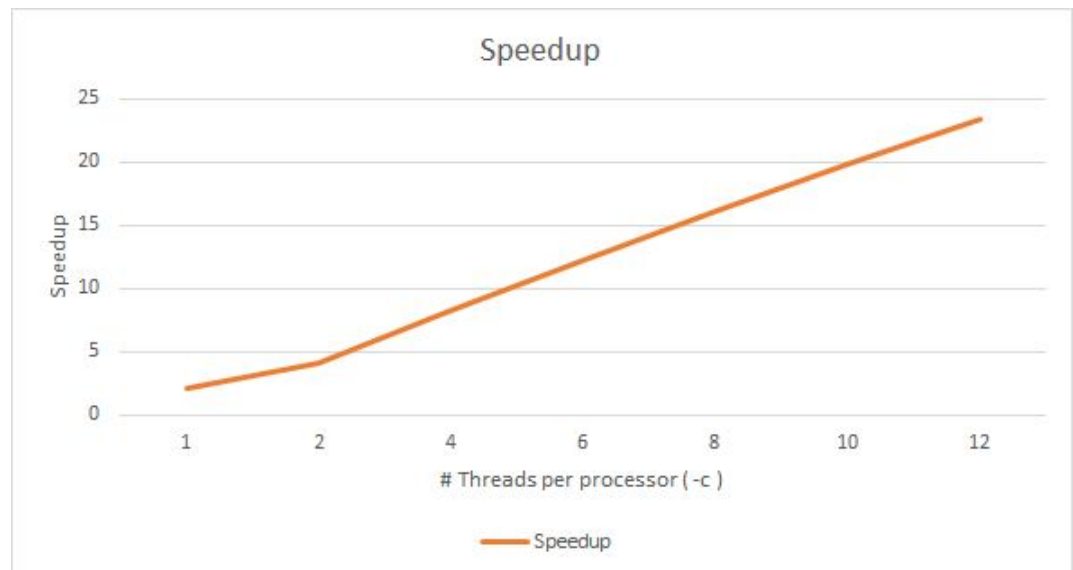
-N1 -n1 -c12

```
hw2 > ≡ slurm-2641502.out
1 Took 43.68944 second
2
```

hw2a:	N=1	n = 1	-
# Threads per processor	Time (sec)	Seq time (sec)	Speedup
1	487.35855	1022.11	2.09724442
2	245.13978	1022.11	4.16949872
4	123.57049	1022.11	8.27147323
6	83.8701	1022.11	12.1868222
8	63.69064	1022.11	16.048041
10	51.6535	1022.11	19.7878169
12	43.68944	1022.11	23.3948982

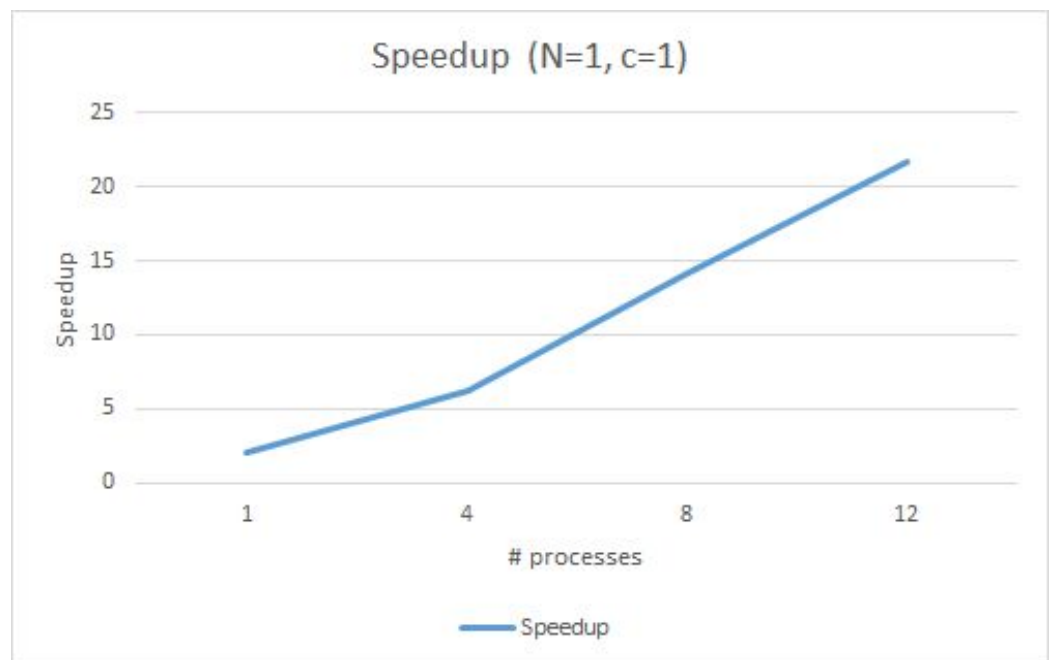
ii、 Plots: Scalability & Load Balancing

hw2a : Pthread

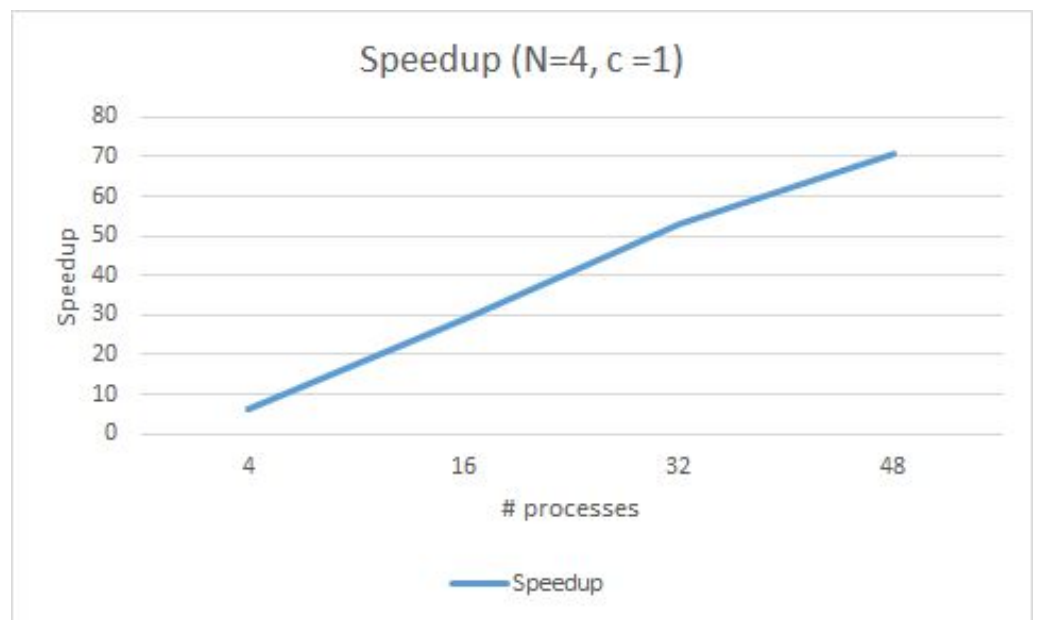


hw2b: Hybred

(1) $N = 1$



(2) $N = 4$



(b) Load Balancing Diagram:

hw2a Pthread version:

```
1 Thread 0 took 0.10160 second
2 Thread 207 took 7.78412 second
3 Thread 19 took 7.78616 second
4 Thread 62 took 7.78699 second
5 Thread 172 took 7.78559 second
6 Thread 1 took 7.80597 second
7 Thread 58 took 7.79674 second
8 Thread 208 took 7.79699 second
9 Thread 52 took 7.79832 second
10 Thread 10 took 7.80968 second
11 Thread 246 took 7.81687 second
12 Thread 258 took 7.83573 second
13 Thread 8 took 7.83892 second
14 Thread 124 took 7.83741 second
15 Thread 24 took 7.84883 second
16 Thread 266 took 7.85547 second
17 Thread 26 took 7.86904 second
18 Thread 232 took 7.89227 second
19 Thread 236 took 7.90654 second
20 Thread 256 took 7.91538 second
21 Thread 128 took 7.93539 second
22 Thread 28 took 7.96313 second
```

```
Thread 4017 took 61.63892 second
Thread 3452 took 77.17161 second
Thread 3230 took 79.87644 second
Thread 4208 took 56.08747 second
Thread 4055 took 56.86255 second
Thread 4090 took 56.51299 second
Thread 3932 took 65.77690 second
Thread 3869 took 64.99791 second
Thread 3271 took 80.31923 second
Thread 4137 took 59.15464 second
Thread 4160 took 57.83609 second
Thread 4256 took 54.73688 second
Thread 3193 took 79.50957 second
Thread 3816 took 67.29052 second
Thread 3797 took 67.47126 second
Thread 3999 took 62.22261 second
Thread 4112 took 58.87762 second
Thread 4121 took 59.18852 second
Thread 4089 took 56.55137 second
Took 123.81866 second
```

Since the way I implemented hw2a is to create **“height” number of threads**, so each thread just calculates one row. Because the pixels have locality, which means close pixels may need close number of iterations, so some threads would take longer computation than other threads.

This is a point that I could improve my performance: by creating less number of threads and **setting a smaller CHUNK_SIZE** (instead of a whole row) for each thread. In this way I can **overlap the waiting time of free threads and the computing time of busy threads**.


```
int num_threads = height;
pthread_t threads[num_threads]; // a threads_array.
thread_info thread_INFO[num_threads];
```

```
int rc;
for (int j = 0; j < height; ++j) { // Go through every pixel from y-axis view. (All
    // create thread
    thread_INFO[j].thread_id = j;
    rc = pthread_create(&threads[j], NULL, Calculate, (void*)&thread_INFO[j]);
    // rc = pthread_create(&threads[t], NULL, hello, (void*)&ID[t]); // ID would be
    if (rc) {
        printf("ERROR: return code from pthread_create() is %d\n", rc);
        exit(-1);
    }
}
```

hw2b Hybrid version:

I've let each process to print out their computation time and communication time:

strict34.txt on -N4 -n16 -c1 :

```
hw2 >  slurm-2653606.out
 1 rank 12 Computation time: 11.03508 seonds
 2 rank 12 Comm time: 0.19356 seonds
 3 rank 11 Computation time: 11.03672 seonds
 4 rank 7 Computation time: 11.05184 seonds
 5 rank 11 Comm time: 0.19316 seonds
 6 rank 7 Comm time: 0.17813 seonds
 7 rank 10 Computation time: 11.04047 seonds
 8 rank 10 Comm time: 0.19266 seonds
 9 rank 5 Computation time: 11.07497 seonds
10 rank 5 Comm time: 0.16816 seonds
11 rank 13 Computation time: 11.03630 seonds
12 rank 13 Comm time: 0.20734 seonds
13 rank 6 Computation time: 11.07832 seonds
14 rank 8 Computation time: 11.06585 seonds
15 rank 8 Comm time: 0.18823 seonds
16 rank 6 Comm time: 0.17569 seonds
17 rank 14 Computation time: 11.03958 seonds
18 rank 14 Comm time: 0.21634 seonds
19 rank 3 Computation time: 11.10114 seonds
20 rank 3 Comm time: 0.15864 seonds
21 rank 2 Computation time: 11.10446 seonds
22 rank 2 Comm time: 0.15897 seonds
23 rank 1 Computation time: 11.10627 seonds
24 rank 1 Comm time: 0.15758 seonds
25 rank 4 Computation time: 11.09506 seonds
26 rank 4 Comm time: 0.16985 seonds
27 rank 15 Computation time: 11.03520 seonds
28 rank 15 Comm time: 0.22926 seonds
29 rank 9 Computation time: 11.07217 seonds
30 rank 9 Comm time: 0.19259 seonds
31 Master Took 14.78206 seconds.
32 Master CPU time: 0.09333 seonds
33 Master Comm time: 11.17591 seonds
34 Master IO time: 3.51283 seonds
35 |
```

As you can see, all **slave processes** have close computation time.

strict34.txt on -N4 -n4 -c4

```
hw2 > ≡ slurm-2653547.out
 1  rank 1 Computation time: 41.10508 seonds
 2  rank 1 Comm time: 0.13231 seonds
 3  rank 2 Computation time: 41.09626 seonds
 4  rank 2 Comm time: 0.15584 seonds
 5  rank 3 Computation time: 41.08990 seonds
 6  rank 3 Comm time: 0.17204 seonds
 7  Master Took 44.78153 seconds.|
 8  Master CPU time: 0.09679 seonds
 9  Master Comm time: 41.17369 seonds
10  Master IO time: 3.51106 seonds
11
```

Record as a table:

Load Balancing	Computation time	Communication time	I/O time	(sec)
Process 1	41.10508	0.13231	0	
Process 2	41.09626	0.15584	0	
Process 3	41.0899	0.17204	0	Total time
Proces 0	0.09679	41.17369	3.51106	44.78154

As you can see, each process has a **balanced amount of workload** .

iii、 Discussion (must base on the results in the plots)

(a) Compare and discuss the **scalability** of your implementations.

For scalability, the Pthread version and Hybrid version with two number of computing Nodes versions all have nice scalability. (Compare with my scalability plot in hw1 - odd-even sort XD)

One thing I noticed was that in the hybrid version -N1 and -N4, the **-N4 plot didn't increase as ideal** as the -N1 version. **I think it was the communication overhead that resulted in this phenomenon.**

(b) Compare and discuss the **load balance** of your implementations.

For the Pthread version, I should cut the `CHUNK_SIZE` smaller to get better load balancing.

For the hybrid version, since I implemented the dynamic work pool, the workload for each process (thread) is quite balanced.

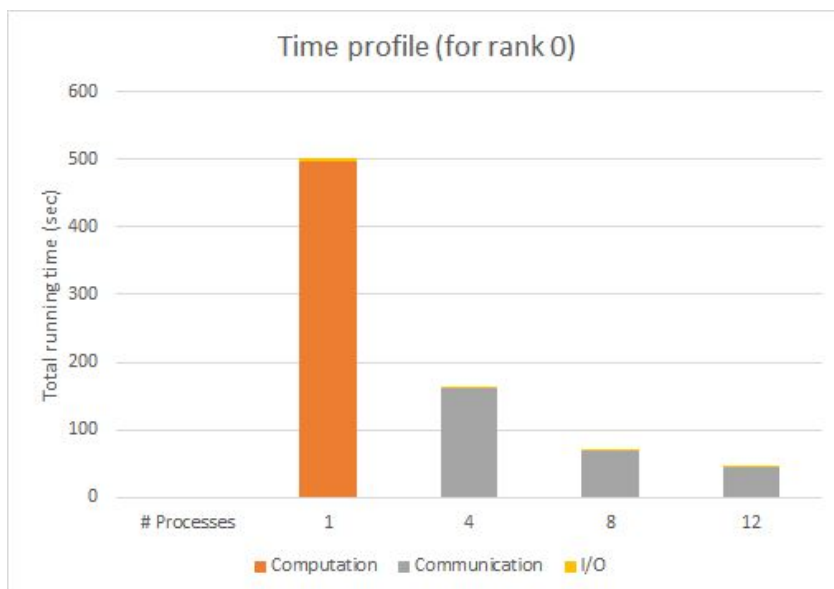
iv、Others

Time profile :

(N = 1, c = 1, n = 1, 4, 8, 12)

Time Profile	Total Time	Computation	Communication	I/O	Bottleneck	Percentage
# Processes						%
1	501.07886	497.57064	0	3.50822	CPU	99.299
4	163.76195	0.09699	160.1532	3.51176	Comm	97.796
8	72.1891	0.09679	68.58429	3.50802	Comm	95.006
12	47.24935	0.09534	43.6414	3.5126	Comm	92.364

When there's only ONE process, the computation takes the most time (for the master process). When there are other slave processes to help computation, the dominating factor of the master process becomes the communication time. It make sense.



3. Experience & Conclusion

(1) Conclusion

The most critical factor of this assignment is the **Load Balancing** of our program. Since every pixel requires different number of iterations when calculating whether it is mandelbrot set, if the load of each process or threads are not balanced, one process or thread might spend a lot of time waiting for another very busy process (thread), which is not a good program in maximizing the utilization of multiple processes (threads).

(2) What I have learned

- a. I've learned how to implement a dynamic task assignment work pool.
- b. I've learned how to construct custom MPI DataType using MPI_Struct.
- c. I've learned how to use MPI_Gatherv().
- d. I've learned how to accelerate computation time in SIMD structure when the data are independent using Vectorization
- e. I've learned how to search instructions on Intel Intrinsics User Guide.
- f. I've learned how to improve the performance of a scientific computation program using load balancing in parallel computation.

(3) Difficulties I encountered

- a. In the beginning, I tried to complete the hw2a and hw2b-judge first. In my hw2b first version, I used very simple implementation and tried to get all test cases correct. I simply divided rows by the number of processes and passed them evenly. However, I have some problems when gathering results from each data. Since there might be remainder when dividing the number of rows per process, I assign rank 0 to handle all remaining rows. Therefore, rank 0 might calculate a different number of rows from other ranks and I would have to use **MPI_Gatherv()** to gather different numbers of pixels to rank 0. **After a while of debugging and revising the code, I finally got AC.**
- b. Then, I seek to improve the performance. I tried to implement a dynamic task assignment work pool using MPI communication such as MPI_Send and MPI_Recv, and use OpenMP for each MPI process to create local threads for computation. However, since the master process now needs to know what row number did the slave process send back, the message Slave sended cannot be a simple MPI DataType. So I implement Struct :

```

111 // Received from each process.
112 typedef struct MPI_send_data{
113     int row_number;
114     int tag;
115 }Send_data;
116
117 // Received from each process.
118 typedef struct MPI_recv_data{
119     int row_number;
120     int rank;
121     double recv_ans[WIDTH_MAX_SIZE];
122 }Recv_data;
123
124

```

Send_data is used when the Master sends tasks to Slaves.

Recv_data is used when a Slave returns its results to Master.

Implementing the dynamic task assignment is not that easy, since many details need to be considered and handled.

- c. After completing hw2b_version2, I started to work on Vectorization. Here I encountered countless of problem:
 1. The Vectorization code has compile errors.
 2. The Vectorization on fast01.txt took 206 seconds! (OMG)
 3. The Vectorization ran a little faster, but the correctness didn't pass. (97.5%, 81.7%, ect)
 4. The Vectorizaion passed compile and correctness, but was not faster. (About the same time as the non-vectorized version.)
 5. Finally, it is 100% correct and about 1.5 factor faster!

Some solutions to the above problem I used are :

1. move the declaration of vectorized 128-bit registers to global instead of local.
2. Reuse registers and reduce the number of declared registers.
3. Use bitwise AND to determine whether break from while-loop instead of store to int & double and use sequential comparison.