COMP90025 Parallel and Multicore Computing Project 1b: computing the number of points in the Mandelbrot Set with OpenMPI

Jiayun He 938828
Hongming Yi 917352
The University of Melbourne

September 15, 2018

Abstract

The purpose of this project is to parallelise the Mandelbrot Set. We used a few approaches to achieve parallelism and ran tests to tried out. Basically, we divide the node into master node and slave node, the master node read the input and then arrange the task to the rest slave node. When the slave node finish their task, then give the result to the master node.The master node wait until all the node finish the job and add these result together.

1 Introduction

The Mandelbrot Set consists of all choices for C(where Z starts at zero and C is a complex number), the condition for choosing C is that giving iterations times, the Z will never beyond number 2.

Algorithm 1 CountMandelbrotSet

Require: real_lower, real_upper, img_lower, img_upper, num, maxiter
  count = 0
  $real\_step = \frac{real\_upper - real\_lower}{num}$
  $img\_step = \frac{img\_upper - img\_lower}{num}$
  for $real = 0$ **to** num **do**
    for $img = 0$ **to** num **do**
      $count = Inset(real\_lower + real \cdot real\_step, img\_lower + img \cdot img\_step, maxiter)$
    end for
  end for

The algorithm CountMandelbrotSet firstly read six input, real_lower and real_upper are in real axis, img_lower and img_upper are in the imaginary axis. The num represents how many parts the real and imaginary axis are divided into.The intersection of the real and imaginary axis represents a possible coordinate of C.The algorithm is to try all the possible situation that C can satisfy

the boundary requirement by using two iteration.The first loop is letting value real from zero to num, the second loop is letting value img from zero to num, each times calling the algorithm Inset to check whether C is inset or not.

Algorithm 2 Inset

Require: $real, img, maxiter$
  z_real = real
  z_img = img
  for $iter = 0$ **to** $maxiter$ **do**
    $z_2\_real = z\_real \cdot z\_real - z\_img$
    $z_2\_img = 2 \cdot z\_real \cdot z\_img$
    $z\_real = z_2\_real + real$
    $z\_img = z_2\_img + img$
    if $z\_real \cdot z\_real + z\_img \cdot z\_img > 4$ **then**
      return 0
    end if
  end for
  return 1

The algorithm Inset is a function that check whether C can satisfy the boundary requirement by using mathematical formula.

2 Method

2.1 static distribution

The static distribution evenly distributed the task to each slave node except the last slave node, the process is described in Algorithm3. The partition strategy is to "stride" the area by dividing the range on the real axis.
Master node use MPI_Isend method to send tasks to the slave nodes, and different slave node handle distinct tasks. These tasks has different lower and upper bound in real axis and share same range on the imaginary axis.
In the main function, according to the rank number, to decide call which type of nodes, if rank equals to zero, then it calls master node, else call slave node. The master will determine partial task and assign value to an array named task, which will be sent to slave node accordingly. In the array, task[0] store real_low, task[1] store real_high, task[2] store num. After sending the task to slave node, the master node will wait until all the slave node finish their task.Slave node will report their result by using MPI_Irecv, finally, the master node add there result together and get the final result.
In addition, we add #pragma omp for in two functions (namely the inset function and the mandelbrotsetCount function). These function both has two loop and each step can be executed individually, so adding the OpenMP Directives can improve the executing speed while retaining the correct answer.

Algorithm 3 static divide the task

$current\_position = real\_lower$
$real\_step = (real\_upper - real\_lower)/num$
$taskCount = worldsize - 1$
$taskSize = num/taskCount$
for $rank = 1$ **to** $worldsize - 1$ **do**
  if $rank = nodesNum - 1$ **then**
    $task[0] = current\_position$
    $task[1] = real\_upper$
    $task[2] = num - taskSize \cdot (taskCount - 1)$
  else
    $task[0] = current\_position$
    $current\_position += taskSize \cdot real\_step$
    $task[1] = current\_position$
    $task[2] = taskSize$
  end if
end for

2.2 semi-dynamic distribution

Unlike static distribution method where taskCount = worldsize - 1, in dynamic distribution method, we add a task_factor variable. The task_factor variable indicates the portion of tasks that are not assigned to slave nodes during the initial task distribution. In fact, the static distribution method is a special case with task_factor equals 1. In this semi-dynamic method, $taskCount = (worldsize - 1) \cdot task\_factor$. By testing different number of task_factor, we find when task_factor equals to worldsize, the run time is optimal. And the division of this dynamic method can be found in Algorithm 4.(code applying dynamic distribution called parallel 3). This is a semi-dynamic method because the partitions are predetermined by the size of input and worldsize.

2.2.1 improved dynamic distribution version

When doing experiment, we find that if the input num is greater or closer to the value $(worldsize - 1) \cdot task\_factor$, running the dynamic distribution program will result a error output. So in this method, we combine the static distribution method and semi-dynamic distribution method.(in experiment, this version code called parallel 4) The procedure is following:

```
if((worldsize-1)*TASK_FACTOR < num/2){
        if(myrank == 0) master();
        else slave();
        MPI_Finalize();
}
else{
    if(myrank == 0) parallel2_master();
    else parallel2_slave();
    MPI_Finalize();
}
```

Algorithm 4 dynamic divide the task

```
//initial distribution
current_position = real_lower
real_step = (real_upper − real_lower)/num
taskCount = (worldsize − 1) · task_factor
pending = taskCount
finished = 0
taskSize = num/taskCount
for rank = 1 to worldsize − 1 do
  task[0] = current_position
  current_position+ = taskSize · real_step
  task[1] = current_position
  task[2] = taskSize
  MPI_Isend(task)
  pending ← pending − 1
end for
//when slave node finish their task, they request to master node whether
there still has task to do
while finished <= taskCount do
  MPI_Irecv from slave node
  finished ← finished + 1
  if pending > 0 then
    task[0] = current_position
    if current_position > real_upper then
      task[1] = real_upper
      task[2] = (num − taskSize · (taskCount − 1))
      pending ← pending − 1
      MPI_Isend(task)
    else
      task[1] = current_position
      task[2] = taskSize
      pending ← pending − 1
      MPI_Isend(task)
    end if
  end if
end while
```

3 Experiment and Results

To experiment on different algorithms, we tried our program on Spartan, both on the Cloud partition (Only for some experimental tryouts) and the physical partition. The speedup is calculated by:

$$S = \frac{T(n)}{P(n)}$$

Where p is the number of processors, T(n) is the sequential run time and P(n) is the corresponding parallel run time.

3.1 Test on small input

Most of our experiments are with a relatively small input: -2.0 1.0 -1.0 1.0 1000 10000

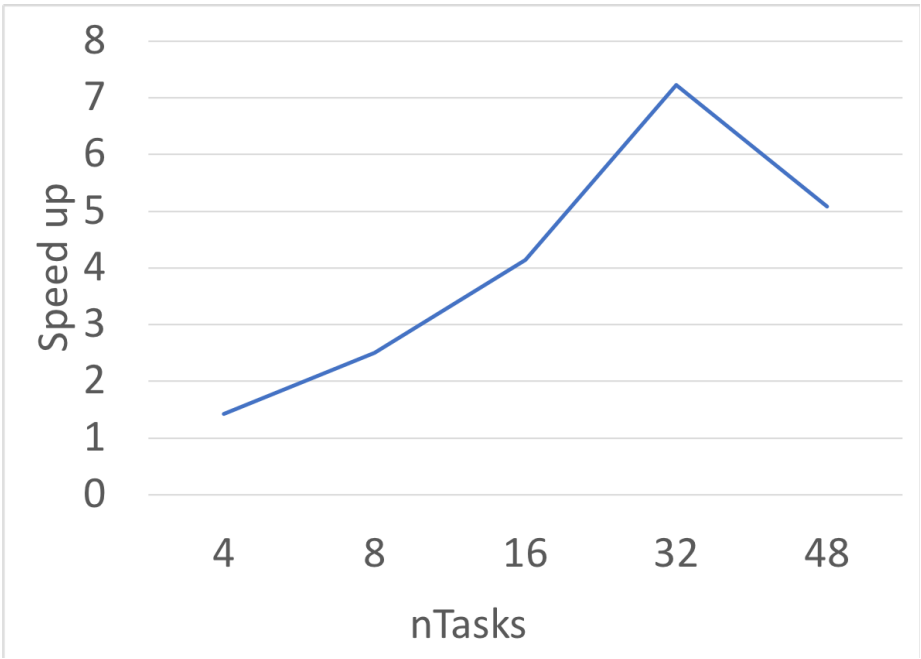

Figure 1: Speedup in parallel2 (with different nTasks)

From this figure, we can know that normally, when having more nTasks to calculate, the speedup is increasing, however, when nTasks increase from 32 to 48, the speedup is decreasing. We assumed that it was because some processors are much more heavily loaded than the others, according to the nature of mandelbrot set. We then turned to parallel3, which utilized a semi-dynamic distribution. We experiment different task factors on different number of ntasks (we used the –nodes and –ntasks-per-node options on spartan) to find an optimal value.

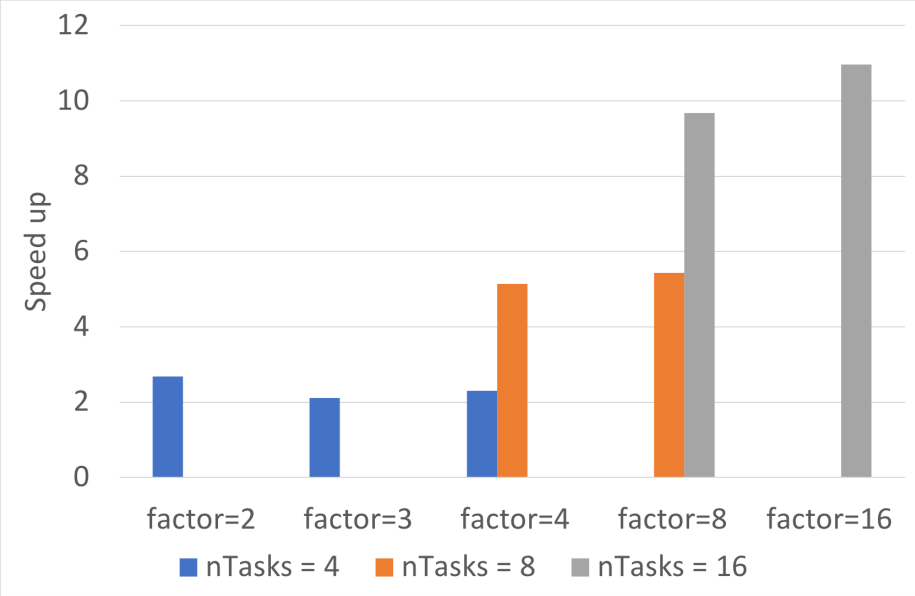

Figure 2: Speedup with parallel3 (with different nTasks and task factors)

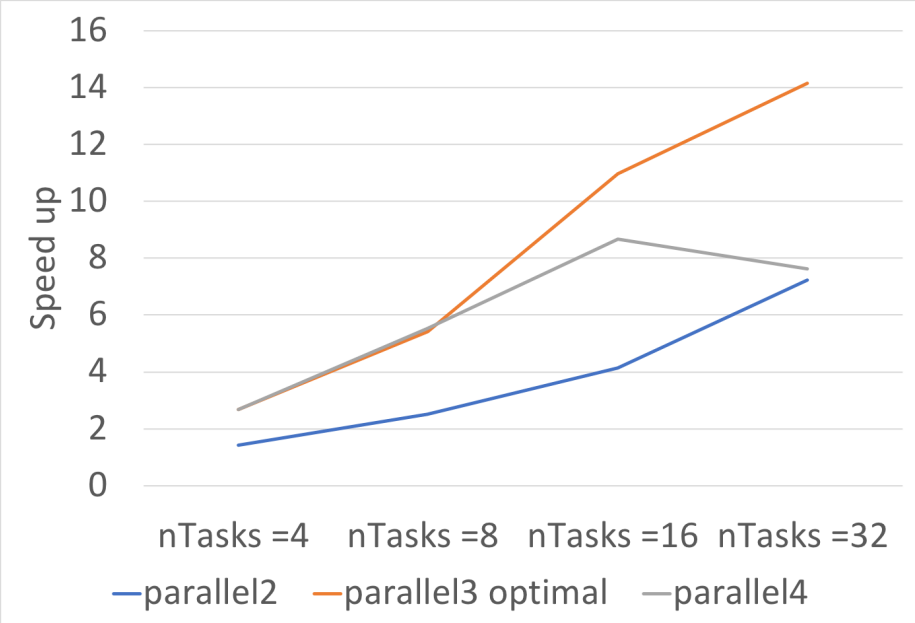

Figure 3: Speedup from three parallel algorithm

3.2 Test on larger input

We increases the input num to 10000 and compare speedups among different parallel methods. The programs are ran on Spartan's physical partition with 6

nodes (8 ntasks on each node). The parallel3 optimal is parallel3 with a task factor equals to worldsize.

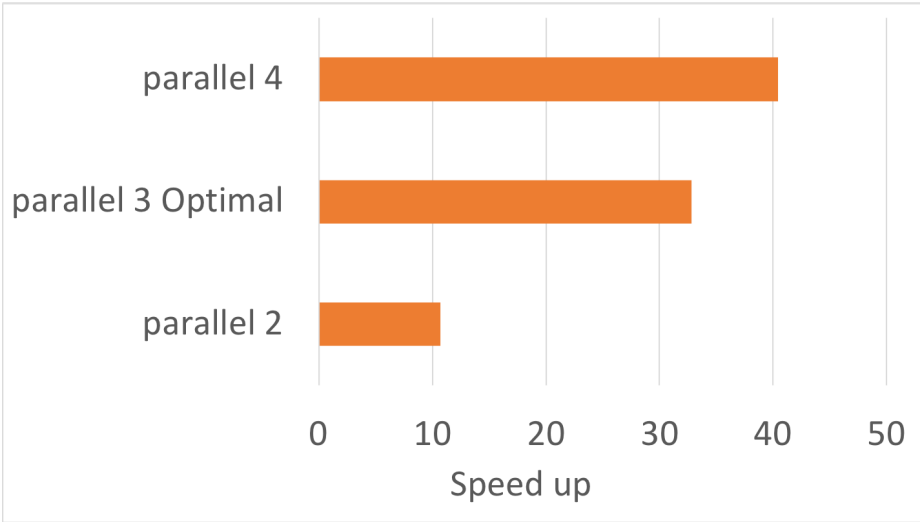

Figure 4: Speedup from three parallel algorithm on big num(nTasks = 48)

Figure 2 implies that different factors result different speedup when the nTasks are the same. From Figure 3, it seems that the optimal of parallel 3 has best speedup, however in Figure 4, when the input number becomes bigger, the parallel4 has the top speedup.

References

[1] Charousset, Dominik, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. "Native actors: a scalable software platform for distributed, heterogeneous environments." In Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control, pp. 87-96. ACM, 2013.

[2] Mandelbrot Set problem
http://wili.cc/blog/mandelbrot-mpi.html