Parallel Programming Exercise 6 - 13

Author	張凱捷 (r08922054@ntu.edu.tw)
Student ID	R08922054
Department	資訊工程所

(If you and your team member contribute equally, you can use (co-first author), after each name.)

1 Problem and Proposed Approach

Problem

In 1970, Princeton mathematician John Conway invented the game of Life. Life is an example of a cellular automaton. It consists of a rectangular grid of cells. Each cell is in one of two states: alive or dead. The game consists of a number of iterations. During each iteration, a dead cell with exactly three neighbors becomes a live cell. A live cell with two or three neighbors stays alive. A live cell with less than two neighbors or more than three neighbors becomes a dead cell. All cells are updated simultaneously.

Write a parallel program that reads from a file an $m * n$ matrix containing the initial state of the game. It should play the game of Life for j iterations, printing the state of the game once every k iterations, where j and k are command-line arguments.

Approach

Partition the problem by rows. Initially, the first process read a matrix from a file and scatter the corresponding rows to each process. In each iteration, each process sends the first and the last rows to the neighbor processes and updates all the cells by the give rules. Every k iterations, $Process_0$ gather all the rows and save the result matrix to a file.

2 Theoretical Analysis Model

All processes need to spend nmr unit of time until $Process_0$ has read the matrix, where r is the basic time unit to read a single character. After spending $nm/\beta + \lambda \log p$ on scattering the matrix, we start to update the matrix for several iteration. The total required time on each iteration is $nm\chi/p$ plus the sending and receiving time $2m/\beta$. Finally, gathering and output take $nm/\beta + \lambda \log p + nmw$ for j/k times, where w is the basic time unit to write a single character to the file.

Hence, the expected execution time of the parallel algorithm is approximately:

$$nmr + (nm/\beta + \lambda \log p) + j \cdot (nm\chi/p + 2m/\beta) + j/k \cdot (nm/\beta + \lambda \log p + nmw) \tag{1}$$

The time complexity:

$$O(nm + jnm/p + j/k \log p) \tag{2}$$

We can further formalize the speedup related to processors by the isoefficiency relation:

$$\begin{cases} \sigma(n,m,j,k) = nmr + j/k \cdot nmw \\ \phi(n,m,j,k) = jnm\chi \\ \kappa(n,m,j,k,p) = (nm/\beta + \lambda \log p) + j \cdot (2m/\beta + 2\lambda \log p) + j/k \cdot (nm/\beta + \lambda \log p) \\ \psi(n,m,j,k,p) \leq \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)} \end{cases} \tag{3}$$

$$nmj \geq C(mj + nmj/k) \tag{4}$$

This method is not scalable if the given k is small.

3 Performance Benchmark

Use Equation 1 and the real execution time of one processor to get the value of χ. λ is approximately equal to 10^{-6} second. We can also do some experiments to get $w = 5 \cdot 10^8$, $r = 10^8$. Then, use the same equation again to get all the estimation execution time.

I wrote a script to automatically submit the jobs with $p = [1, 8]$. For each job, I ran the main program for 20 times, and eliminate the smallest / largest 5 record. The value in the table is the average of the rest ten records.

Table 1: The execution time (second), ($n = 1000,\ m = 1000,\ j = 10,\ k = 10$)

Processors	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Real execution time	0.91590	0.49060	0.35010	0.27779	0.23623	0.20716	0.18701	0.17216
Estimate execution time	0.91590	0.48946	0.34732	0.27625	0.23360	0.20517	0.18487	0.16964
Speedup	1.00000	1.86690	2.61616	3.29708	3.87711	4.42134	4.89759	5.32019
Karp-flatt metrics		0.07129	0.07336	0.07107	0.07241	0.07141	0.07155	0.07196

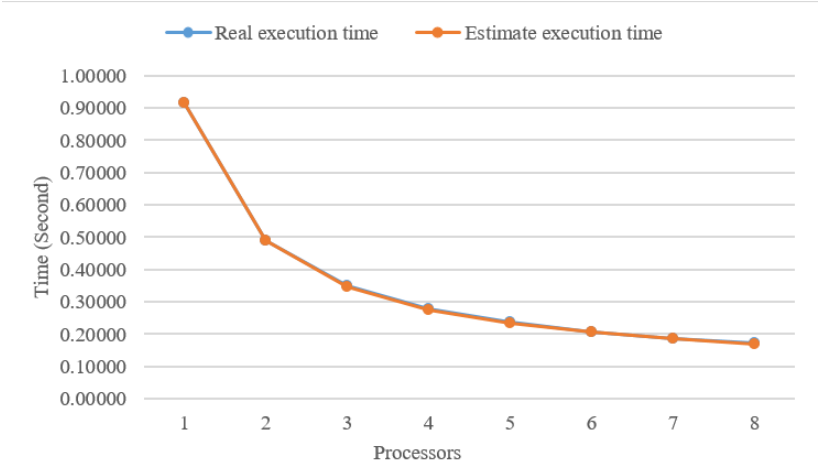


Figure 1: The performance diagram.

I also analyze another test case to show our estimation equation is robust enough to deal with different input.

Table 2: The execution time (second), ($n = 10000,\ m = 10,\ j = 100,\ k = 100$)

Processors	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Real execution time	0.70784	0.36225	0.24704	0.18584	0.15186	0.12868	0.11286	0.09961
Estimate execution time	0.70784	0.35707	0.24015	0.18169	0.14662	0.12323	0.10653	0.09400
Speedup	1.00000	1.95401	2.86523	3.80886	4.66102	5.50079	6.27186	7.10633
Karp-flatt metrics		0.02354	0.02352	0.01673	0.01818	0.01815	0.01935	0.01797

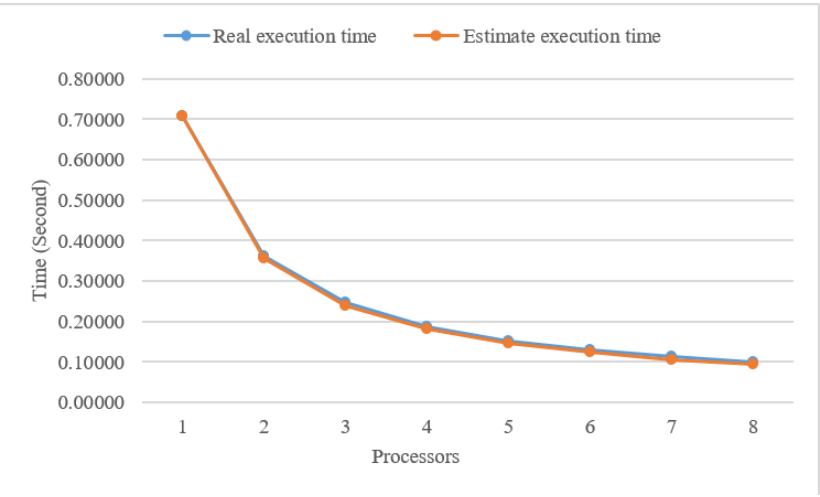


Figure 2: The performance diagram.

4 Conclusion and Discussion

4.1 What is the speedup respect to the number of processors used?

The speedup is less than p since there exist some overhead while doing parallel programming. Especially on this problem, I/O and communication contributes a lot of run time overhead. We can found that the larger m would contribute more overhead by comparing the above two tables.

4.2 How can you improve your program furthermore?

We can use the OpenMP to further optimize the column update on each process.

4.3 How do the communication and cache affect the performance of your program?

In this problem, scattering and gathering contribute a certain portion of run time. Maybe we can just use *lseek* method to directly read and write the correspond part of each processors to the file.

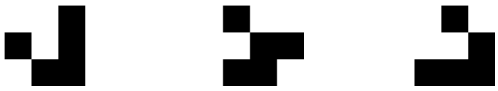
4.4 How do the Karp-Flatt metrics and Iso-efficiency metrics reveal?

The Karp-Flatt metrics values fluctuate is stable because the input size is large enough.

Appendix (Optional)

I wrote a python script to generate the images for all the outputs. Following are a example of the moving figure.

(a) $t = 0$ (b) $t = 1$ (c) $t = 2$



(d) $t = 3$ (e) $t = 4$ (f) $t = 5$