




# Parallel Programming

Peyman Shobeiri

ID: 40327586

Fall 2024



## Table of Contents

<b>QUESTION 1</b> .....	<b>3</b>
A) .....	3
B) .....	3
<b>QUESTION 2</b> .....	<b>5</b>
<b>QUESTION 3</b> .....	<b>5</b>
A) .....	6
B) YES, THE EFFICIENCY AND SPEED CAN BE IMPROVED BY USING A BLOCK-CYCLIC MAPPING. HERE, THE COLUMNS ARE DISTRIBUTED CYCLICALLY AMONG THE PROCESSORS: PROCESSOR $P_0$ GETS COLUMNS 0, 3, AND 6, PROCESSOR $P_1$ GETS COLUMNS 1, 4, AND 7, AND PROCESSOR $P_2$ GETS COLUMNS 2, 5, AND 8. IN THIS WAY, DURING THE EARLY DIAGONALS, ALL PROCESSORS ARE COMPUTING AND THEREFORE HAVE LESS IDLE TIME. IN THE CASE OF $N = 9$ , THE COMPUTATION TIMES FOR THE DIAGONALS ARE NOW MORE EVENLY DISTRIBUTED AMONG THE PROCESSORS AND ARE CALCULATED LIKE THIS: .....	7
<b>QUESTION 4</b> .....	<b>7</b>
<b>REFERENCES:</b> .....	<b>9</b>

## Question 1

**Q.1. [20 marks]** In the context of parallel search techniques for discrete optimization problems, consider the *distributed tree search* scheme in which processors are allocated to different parts of the search tree dynamically as follows: initially all processors are assigned to the root. When the root node is expanded (by one of the processors assigned to it), disjoint subsets of processors at the root are assigned to each child node based on a processor-allocation strategy. One such possible strategy is to divide the processors equally among the child nodes. This process continues until there is only one processor assigned to a node; at this time the processor searches the tree rooted at the node sequentially. If a processor finish searching the search tree rooted at the node, it is reassigned to its parent node. If the parent has other child nodes still being unexplored, then this processor is allocated to one of them. Otherwise, the processor is assigned to its parent. This process continues until the entire tree is searched. Answer the following questions: (a) What is (are) the advantage(s) of this scheme? (b) Can it cause load imbalance? Explain your answer. If your answer is “yes”, give a modified scheme to fix it.

a) Advantages of the Scheme:

1. This scheme allows the searching of several subtrees simultaneously when processors are reassigned. This can speed up the search process.
2. The scheme is easy to implement as the processors can be allocated easily without much overhead.
3. The scheme guarantees that all nodes in the search tree will be traversed. This reduces the possibility of a part of the search space being missed.

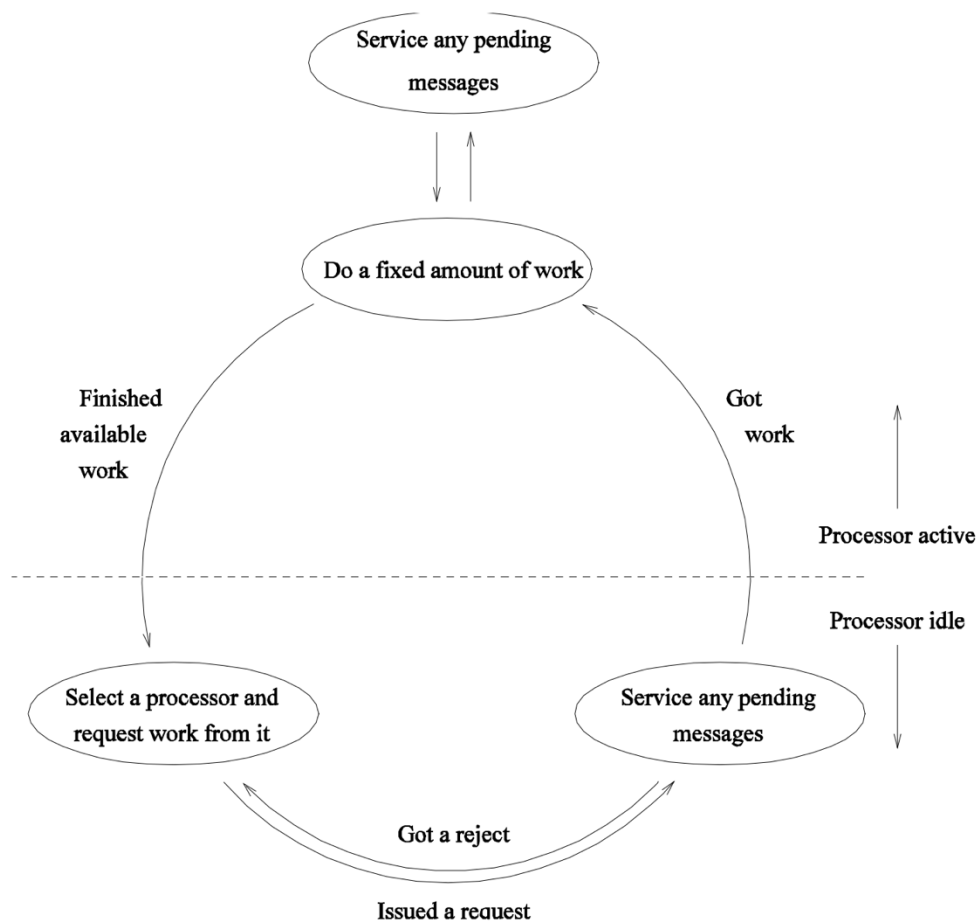
b) Yes, this scheme can lead to load imbalance based on the following reasons:

1. In scenarios where the tree is imbalanced, certain subtrees may require much more processing time than others. For example, if a parent node has four child nodes, and two of them require extensive computation while the other two finish quickly, the processors assigned to the completed nodes will become idle.
2. Processors that complete their tasks early cannot be reallocated to assist other busy processors. This results in inefficiency and idle resources.
3. Many processors remain idle during the initial early stages of the tree because there are limited parallelization opportunities.

In order to handle the problem of load imbalance, dynamic load balancing techniques can be used. In this technique, extra work from an overloaded processor is given to an idle processor. For instance:

- When one processor finishes its job, it asks for more work from other processors that are heavily loaded.

The figure below shows how dynamic load balancing works. A processor that has completed its work requests some work from another busy processor, enabling computational tasks to be more evenly distributed throughout the system. This approach helps to achieve better workload distribution and reduces idle time for processors, improving overall efficiency.



## Question 2

**Q.2. [20 marks] Prove that the *modified Diskstra's token-based termination detection algorithm*, discussed in the context of dynamic load balancing in parallel search for discrete optimization problems (chapter 11.4.4 of textbook and the slides) is correct, i.e., the token initiator processor receives a green colored token if and only if all other processors have terminated; moreover, if the initiator processor is idle when it receives a green token, then termination is signaled.**

In the modified Dijkstra's token-based termination detection algorithm, we can summarize that if the initiator receives a green token, then all processors have completed their work. Also, it can be stated that unless all processors have finished their tasks, the final token will not be green. Therefore, if the initiator is idle and receives a green token, the algorithm is complete.

In this algorithm, when a processor is idle, processor  $P_0$  detects termination by making itself green and passing a green token to  $P_1$ . If  $P_1$  is green, it will then pass the green token to  $P_2$ , and so on. However, if it encounters a red processor  $P_i$ , after  $P_i$  completes its work, it sends a red token back and makes itself green again. Since the token is red, the process must be repeated.

We can prove this by the contradiction method. Let's assume that the initiator receives a green token, but at least one processor has not completed its work. Let's call this processor  $P_x$ , which has not finished its tasks, while the token was passed as green. Since  $P_x$  is not idle, it can't release a token unless it has become idle. This is our first contradiction.

Moreover, consider the scenario where  $P_x$  went idle and released a green token, after which it was assigned more work by another processor,  $P_n$ . In this case, we can be sure that all processors with IDs less than  $n$  (those with ranks lower than  $P_n$ ) must have become idle and sent out their green tokens. Therefore, the processor that assigned the work to  $P_x$  must indeed be a processor with an ID rank greater than  $n$ , meaning it has not yet received the token. Accordingly, we can conclude that  $P_n$  will have a red color token. Consequently, a red token will reach the initiator. This contradicts our initial assumption of receiving a green token.

Therefore, with the contradiction method, we can say that our initial assumption was false, and we were able to verify the proof.

## Question 3

**Q.3. [20 marks] For the parallel formulation of the longest common subsequence problem (section 12.3.1 from textbook and the slides), as we discussed in class the maximum efficiency is upper bounded by 0.5 for very large  $n$ . Now consider the case for a small  $n$ , say  $n = 9$ , and a coarse-grained solution where more columns are mapped per processor. Answer the following questions: (a) considering  $n = 9$  and number of processors,  $p = 3$ , calculate the maximum possible speedup and corresponding efficiency when block mapping is used (i.e. processor  $P_0$  is mapped**

**columns 0, 1, and 2; P1 is mapped columns 3, 4, and 5; and so on). (b) Can the maximum efficiency (and speedup) be improved by using a different mapping? If your answer is “yes”, then illustrate and explain for the case:  $n = 9$  and  $p = 3$ .**

In the serial implementation of the longest common subsequence problem, we work with an  $n \times m$  matrix. Since it takes constant time to process each entry in the matrix, the time complexity is given by  $O(nm)$ . In this case,  $n = m$  so the time complexity is  $O(n^2)$ .

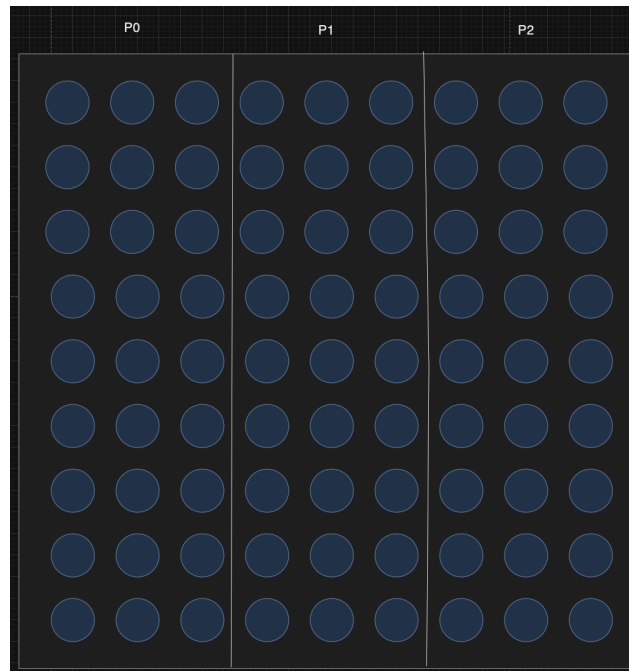
In the parallel implementation, processing an entry  $F[i,j]$  requires either the value of  $F[i-1,j-1]$  or the value of  $F[i,j-1]$  from the processing element to its left. Each processing element must communicate with its nearest neighbors, resulting in a communication overhead of  $t_s + t_w$ , where  $t_s$  is the startup time and  $t_w$  is the per-word transfer time. Additionally, the computation time for each element will be  $t_c$ . The processing takes place in a diagonal fashion, so the total number of iterations will be  $2n - 1$ , where  $n$  is one side of the  $n \times n$  matrix. A processing element will be computing an entry on the diagonal. So, the efficiency will be:

$$\text{Efficiency} = \frac{n^2 \times t_c}{p(2n-1)(t_s + t_w + t_c)}$$

Here,  $p$  is number of processors. If we assume that it is possible to communicate values between processing elements instantaneously, then  $t_s = t_w = 0$ .

$$E = \frac{n^2}{p(2n-1)}$$

a) When block mapping is used, size of matrix is  $9 \times 9 = 81$  and number of processors is 3. So, mapping will be done like this:



Each processor computes all entries for its assigned columns. For a diagonal with  $k$  elements, the processors work sequentially, with  $P_0$ ,  $P_1$ , and  $P_2$  each handling  $k/p$  elements. With this distribution, we calculate efficiency as follows:

$$E = \frac{n^2 \times t_c}{p \times T_p}$$

The total parallel execution time involves summing the computation times across all diagonals. When  $n = 9$ , there are a total of  $2n - 1 = 17$  diagonals. Suppose  $t_s = t_w = 0$ , then the computation time for each diagonal in terms of  $t_c$  is:

$$T_p = 11 + 17 + 17 = 45$$

$$S = 81/45 = 1.8$$

This simplifies to:

$$E = \frac{81 \times t_c}{3 \times 45 \times t_c} = 0.6$$

b) Yes, the efficiency and speed can be improved by using a block-cyclic mapping. Here, the columns are distributed cyclically among the processors: processor  $P_0$  gets columns 0, 3, and 6, processor  $P_1$  gets columns 1, 4, and 7, and processor  $P_2$  gets columns 2, 5, and 8. In this way, during the early diagonals, all processors are computing and therefore have less idle time. In the case of  $n = 9$ , the computation times for the diagonals are now more evenly distributed among the processors and are calculated like this:

$$T_p = 35$$

$$S = 81/35 = 2.31$$

So, we have:

$$E = \frac{81 \times t_c}{3 \times 35 \times t_c} = 0.77$$

## Question 4

**Q.4. [20 marks] Consider the following program (Note:  $S_i$  represents statement i):**

**S1:**  $a = b + c$ ;

**S2:** ....

....

```

Sn: ....
for (int i = 0; i < m; i++) {
Sn+1: x[i] = c + d;
Sn+2: ....
....COMP 428/6281 – Fall 2024
Assignment 3 – page 2 of 2
Sn+20: ....
}

```

Referring to the above program, statements  $S_1$  through  $S_{n+1}$  must execute in sequence due to data dependencies among themselves. In each iteration of the loop, there is dependency between every pair of alternate statements, i.e., there is data dependency between statements  $S_{n+j}$  and  $S_{n+j+2}$ ,  $1 \leq j \leq 18$ . However, there are no loop-carried dependencies among different iterations of the loop. Assume that each statement takes equal amount of time to execute and there are no additional overheads. The program is automatically parallelized. For a given  $n$  and  $m$ , what is the maximum possible speedup  $S_{Max}$ ? Show your calculations.

The program begins with a sequence of statements  $S_1$  through  $S_n$ , followed by  $S_{n+1}$ . All statements need to be performed sequentially because of the data dependency among them. The result of one statement becomes an input for another. There is no parallelization in this part.

After the sequential statements, a loop iterates  $m$  times. In every iteration, there are 20 statements, namely  $S_{n+1}$  to  $S_{n+20}$ . In every iteration, between every pair of alternate statements, there is a data dependence: more precisely, statement  $S_{n+j}$  depends on  $S_{n+j-2}$  for  $3 \leq j \leq 20$ . That means not all statements within an iteration can run in parallel due to intra-iteration dependencies.

However, it is important to note that there are no loop-carried dependencies among different iterations of the loop. This is a crucial point, as it implies that iterations of the loop are independent of each other and can be executed in parallel. So, this loop is a good candidate for parallelization.

To compute the total sequential execution time, we could consider the time it takes to execute all statements without utilizing parallelism. Each statement takes an equal amount of time  $t$  to execute. The sequential portion before the loop consists of  $n + 1$  statements ( $S_1$  to  $S_{n+1}$ ), while the loop contains 20 statements per iteration, resulting in a total of  $20m$  statements for  $m$  iterations. Therefore, the total sequential execution time is expressed as:

$$T_s = (n + 1 + 20m) \times t$$

For the parallel execution time, it is recognizable that the initial sequential statements  $S_1$  to  $S_{n+1}$  still need to be executed sequentially, which takes  $(n + 1) \times t$  time. However, the loop iterations can be executed in parallel since there are no dependencies between them.



Within each loop iteration, due to the dependencies between alternate statements, the statements form two independent chains:

1. Chain A: Consists of statements  $S_{n+1}, S_{n+3}, \dots, S_{n+19}$
2. Chain B: Consists of statements  $S_{n+2}, S_{n+4}, \dots, S_{n+20}$

Each chain contains 10 statements, and within a chain, the statements must be executed sequentially because of the dependencies. However, since the two chains are independent of each other, they can be executed in parallel within the iteration. Consequently, the critical path length for each iteration is the time taken to execute the longest chain, which is  $10 \times t$ .

Since all loop iterations are independent and can be executed simultaneously, the total time to execute all iterations in parallel equals the time to execute one iteration's critical path. Therefore, the total parallel execution time is:

$$T_p = (n + 1) \times t + 10 \times t = (n + 11) \times t$$

Finally, the maximum possible speedup is calculated by dividing the total sequential execution time by the total parallel execution time:

$$S_{\max} = T_s / T_p = (n + 1 + 20m) \times t / (n + 11) \times t = n + 1 + 20m / n + 11$$

## References:

[1]: Grama et al, 2003, Introduction to parallel computing second edition