P2.

The problem is to find the Cholesky decomposition of a given matrix. We will be using the row-based Cholesky-decomposition algorithm for this purpose. For example:

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 13 & 23 \\ 4 & 23 & 77 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 4 \\ 0 & 3 & 5 \\ 0 & 0 & 6 \end{bmatrix}$$

This is of the form: $A = L \times L^T$

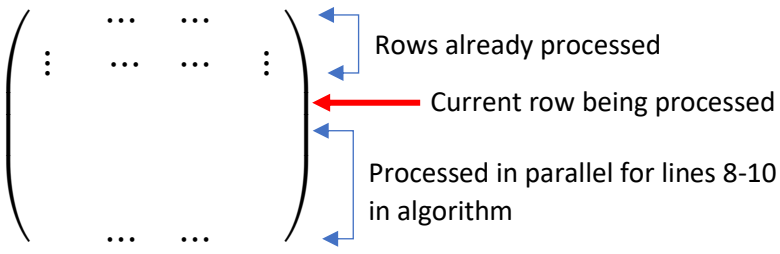
The algorithm is as follows:

```
1.  procedure CHOLESKY(A):
2.  begin
3.      for k := 0 to n-1 do
4.          begin
5.            A[k, k] := √A[k, k]
6.            for j := k + 1 to n-1 do
7.               A[k, j] := A[k, j]/A[k, k]
8.            for i := k +1 to n-1 do
9.               for j := i to n-1 do
10.                 A[i, j] = A[i, j] - A[k, i] x A[k, j]
11.     endfor;
12. end CHOLESKY
```

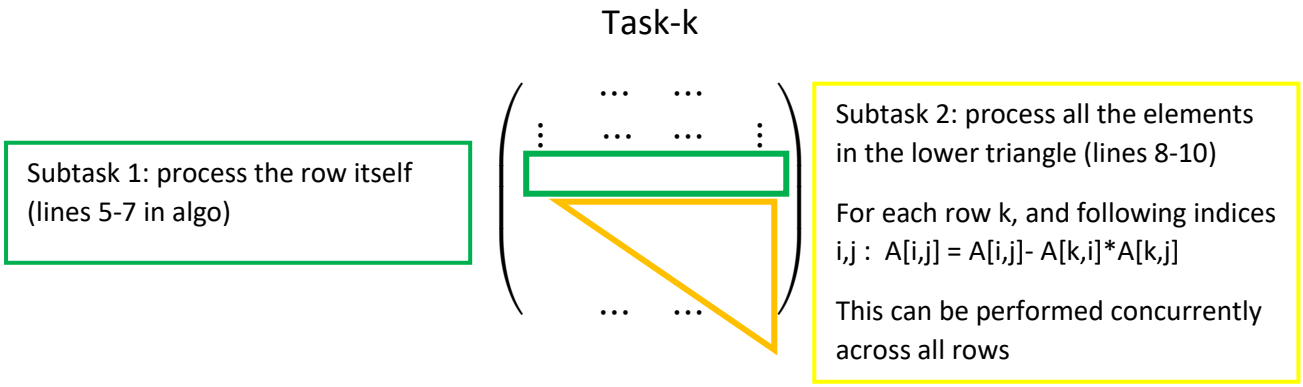
Key points observed:

1. The complete processing of any row is dependent on the completion of all rows above it.
2. After a certain row has been processed, all the rows below it use that row to modify each of its element. (line 8 - 10)

Using these 2 observations, we can see that we need to synchronize the processing of each row as well as parallelize the relevant sections.
Parallelizing lines 8-10 would be the most effective as after the k'th row has been processed, all the rows below it have no other dependencies to perform the required operations in lines 8-10.
Therefore, at some point in time, when the k'th row is being processed:

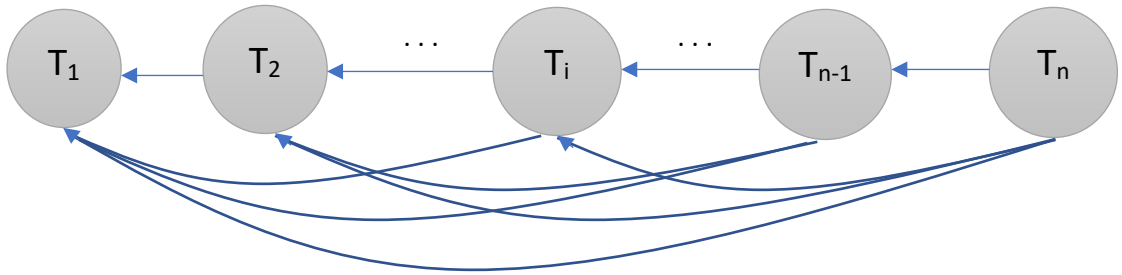


Thus, we can define the processing of each row as a task, consisting of two subtasks: first, processing the row itself and second, subtracting the relevant terms from elements in the remaining rows below.



Task Dependency Graph

Each task is dependent on its previous task to finish so that it can proceed. The task dependency graph will be as follows:



Each Task is also directly dependent on completion of all the previous tasks.

Task Interaction Graph

Each task will send its processed row to each task below it for processing. Thus, the task interaction graph will be the same as, or rather inverse of the task dependency graph.

Parallelism in Subtask 2

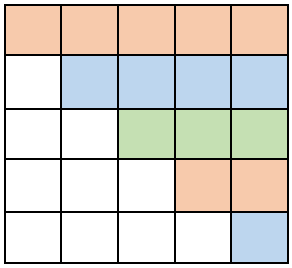
The actual part of the task that runs concurrently on different processes is sub-task 2. Since each process would have some rows, and there is no dependency among rows for the duration of subtask 2, each process can independently perform the required computation for all rows it is responsible after receiving the required information computed in subtask 1.

Process Mapping

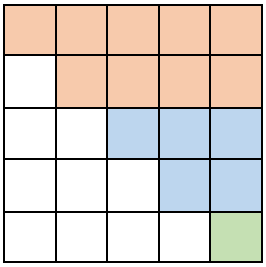
The mapping will be based on data-decomposition. The rows of the matrix will be partitioned in a round robin manner. The process that is responsible for row(k) will perform task(k). All processes that are responsible for rows(i>k) will parallelly perform subtask-2(k). Since each row must be computed one after the other, a round robin manner of distribution makes sense to equally divide tasks and processing time for each process.

The other way that the rows could be divided was block based. Each process would first compute all the rows it has and concurrently send the results to all following processes to work on. But this is less feasible as after a process is done with working all its rows, it would be idle for the rest of the duration. But the round robin way ensures that a process works on some part of the data for the maximum possible time.

The following example shows how dividing a 5x5 matrix among 3 processes would look like in the two cases. For our use case, we would go with the former. The un-coloured elements are not processed at all.



Division using round robin.



Division using blocks

We can also see that round-robin ensures for a more equal distribution of elements as compared to block division, thus further strengthening our decision in selecting it for our case.

Parallel Algorithm

The following pseudocode describes the parallel algorithm used:

```
1. procedure parallel-Cholesky(A):
2. begin
3.     allocate rows to all processes in a round robin manner
4.     for all rows:
5.         if (process owns current row to be processed):
6.             process the row and send result to all other processes
7.         else
8.             receive result of the current row and
               perform computation on remaining rows
9.     gather result from all processes
```

Performance Measurements

The time taken for the program to solve a 500x500 matrix in milliseconds is shown below:

1: 65ms	2: 35ms	3: 38ms	4: 34ms	5:28ms	6: 25ms

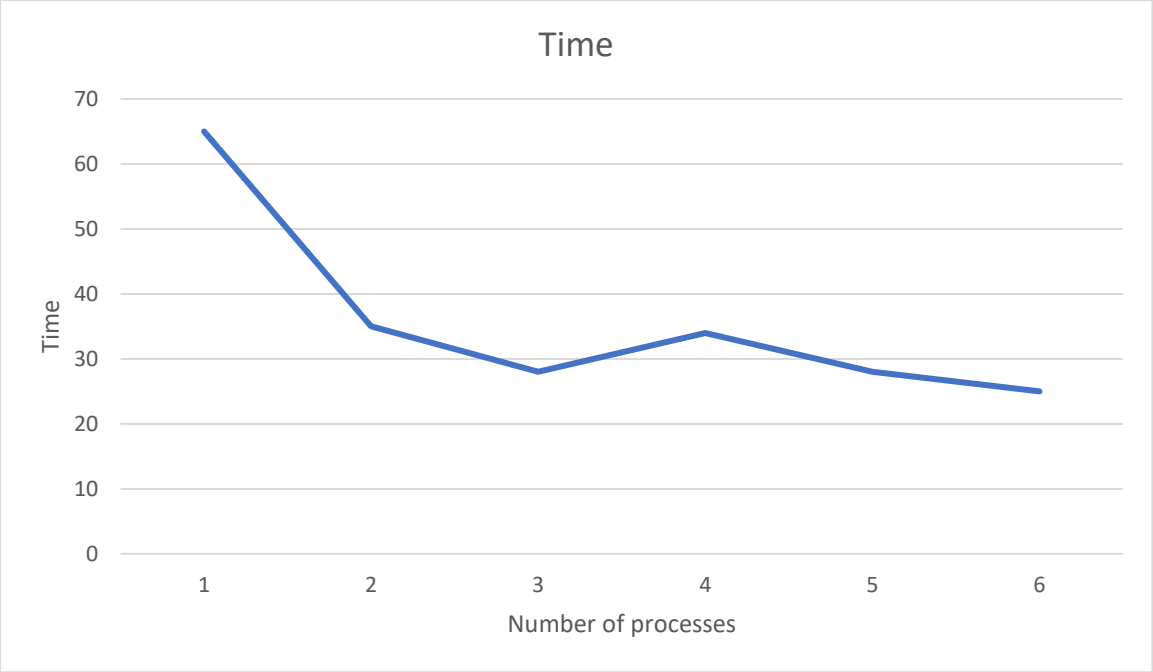
Speedup

2: 1.85	3:1.71	4:1.91	5:2.32	6: 2.6

Efficiency

2: 92.5%	3:57%	4:47%	5:46%	6: 43%

The trend is due to the message passing overhead incurred by our solution. This overhead becomes less significant as the size of the matrix starts increasing.



Scalability

E = $1/(1+T_o/T_s)$

Total communication cost $T_0 = n^2 * p$

$T_s = kn^3$

$T_0/T_s = \frac{p}{n}$

Hence the algorithm is scalable.