1. (35%). Consider a problem with serial execution time $T_1 = n^2$ and parallel execution time with $p$ processors is

$$T_p = \frac{n^2}{p} + \frac{n}{p} \cdot \log(p)$$

/* you can view the second term as the cost of communication

a) Derive speedup and efficiency

b) Derive Iso-efficiency function

c) Derive scaled speedup based on Gustavson’s law – keep the parallel execution equal to $T_1$

d) Derive scaled speedup if the problem size is increased proportional to $p$

Key.

a). Speedup = $T_1/T_p = \frac{(n^2)/[(n^2/p) + (n/p) \cdot \log(p)\]}{\text{you can view the second term as the cost of communication}}$

$$= \frac{n^2}{n^2 + n \cdot \log p} = \frac{p}{1 + (\log p)/n}$$

Efficiency = Speedup/p = $1/[1+ (\log p)/n]$

Note actually this is not a bad parallel algorithm since for very large $n$, $(\log p)/n$ will be very small.

b. Iso-efficiency = $K \cdot T_o$

$$T_p = \frac{n^2}{p} + \frac{n}{p} \cdot \log(p) = (n^2 + n \cdot \log p)/p = (W + T_o(W,p))/p$$

$$T_o = [n \cdot \log p]$$

So iso-efficiency function = $K \cdot n \cdot \log p$ or $K \cdot (W^{1/2}) \cdot \log p$

c. We need set $T_1 = T_p$

$$n^2 = \frac{x^2}{p} + \frac{x}{p} \cdot \log p$$

Here $x$ is the new work we are trying to solve

Rearranging, we get

$$x^2 + x \log p - p n^2 = 0$$

We can solve for $x$ using quadratic equations
\[ x = \{- \log(p) + (\text{or}) \left[(\log(p))^2 - 4(-pn^2)\right]^{1/2}\}/2 \]

If you ignore \((\log(p))^2\)
we get \(x = -\log(p) + (\text{or}) 2p^{1/2}n\) and you can further simplify this as \(O(p^{1/2} \times n)\)

Since problem size is defined by \(n^2\), for our purpose \(x^2\) is \(O(p \times n^2)\)

\[ d. \text{Note that } W = O(n^2). \text{So when we scale the work proportional to } p, \text{the new work is } x^2 = n^2 \times p \]

Now \(T_p = (n^2 \times p)/p + \{(n^2 \times p)^{1/2}/p\} \times \log(p) \]
\[ = n^2 + [n \times \log(p)]/p^{1/2} \]

Scaled speedup = \(T_1/T_p\) (note \(T_1\) is still \(n2\))
\[ = [n^2]/[n^2 + [n \times \log(p)]/p^{1/2}] \text{ which is less than 1} \]

2. (30%). By now you know how matrix*vector product works \((A \times b = c)\). The following shows pseudo code for sequential version

\[
\text{for (i=0; i<n; i+){}
    c[i] = 0;
    \text{for (j=0; j<n; j++)}
        c[i] = c[i] + A[i][j]*b[j];
\}
\]

Assume 2-D (checkerboard) partitioning of \(A\). The vector \(b\) is broadcast to all processors

Results (vector \(c\)) will be computed by processors in the first column

\[
\begin{array}{cccc}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33} \\
\end{array}
\begin{array}{c}
B \\
= \\
C
\end{array}
\]

a). In a narrative form, describe what each processor needs and thus the communication needed

**Key**: Since \(B\) is broadcast to all processors, each processor can perform multiplications and accumulation of values locally. But to compute the final result, processors in a row must send their results to the first column processors. So we need a reduction.
b). Using 2D Mesh, estimate total parallel execution time (ignoring initial cost for data distribution, but include all the communication needed after that).

**Key:** Each processor receives \((n/p^{1/2}) \times (n/p^{1/2})\) A elements and all n elements of B

Each processor performs \((n^2/p)\) multiplications and additions and let us assume the cost of a multiplication plus addition is \(t_c\)

We then perform reduction across \((p^{1/2})\) processors in each row, and each processors contains \((n/p^{1/2})\) results for the reduction.

From textbook page 156, the cost of all-to-one reduction is

\[(t_s + t_w \times m) \log p\]

when there are \(p\) processors and size of message is \(m\)

In our case:

\[(t_s + t_w \times ((n/p^{1/2})) \log (p^{1/2}))\]

Total cost = \(t_c \times (n^2/p) + (t_s + t_w \times ((n/p^{1/2})) \log (p^{1/2}))\)

3 (15%). Consider a MPI configuration with 4 processors a, b, c, d and oldrank of 0,1,2,3. Let color= oldrank\%2 and corresponding keys set to 7,1,0,3. Identify newgroups sorted by newranks when you execute

\[
\text{MPI_COMM_SPLIT (comm, color, key, newcomm);} \\
\]

**Key:** Since we are dealing with modulo 2, the only values for color are 0 and 1

<table>
<thead>
<tr>
<th>Original Rank</th>
<th>Color</th>
<th>Key</th>
<th>New Group</th>
<th>New Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Thus processors 0 and 2 form a new group and they will be ordered with 2 as 0 and 0 as 1

Processors 1 and 3 from the second group and their new ranks with 1 as 0 and 3 as 1

4 (20%). A matrix is symmetric if \(a_{ij} = a_{ji}\) (or the transpose of a matrix is the matrix itself).

Develop an algorithm to check if a n\(\times n\) matrix is symmetric assuming CREW PRAM. What is the complexity of your algorithm.

Develop an algorithm for this problem assuming CRCW PRAM.

**Key:**

We can use either row striping or checker board.
If we use row striping, each thread compares n/p rows to n/p columns to see if they are equal. If so, we can increment a counter (if not add a zero to the counter).

If we use checker-board, each thread compares a submatrix with another submatrix that is diagonally opposite. Again add 1 to a counter if they are equal or zero otherwise.

a). Using CREW. Let us assume you have p threads, each thread is responsible for comparing n/p rows with n/p columns. So we need to compare n^2/p elements. (or each thread compares (n/p^{1/2})^2 (n/p^{1/2}) submatrices.

Since we can increment the counters (shared) in CREW, we need perform the counter increments sequentially. Total complexity = n^2/p + p

b). Using CRCW. Now we can perform the counter updates concurrently and this will take one time unit

Total complexity = n^2/p + 1