1 (20%). The value of $\pi$ can be computed as an integral of the following function

$$\int_0^1 f(x) \, dx = \int_0^1 \frac{4}{1 + x^2} \, dx$$

We use summation to replace integration. We use very small slices of $x$ and compute the integral. For example consider the following code

```c
float x, delta, sum;
int slices = 100000;
sum = 0.0; delta = 1.0/slices;
for (i=0, i<slices; i++){
    x = (i+0.5)*delta; sum = sum + 4.0/(1.0+x*x);
}
pi = sum * delta;
```

Write an MPI program to parallelize this function. Assume that the number of processors $p$ is much less than the number of slices.

**Key.** Each processor computes slices/p iterations. Process with rank $k$ will start its $i$ at $r*n/p$ and goes to $(r+1)*n/p$.

After MPI initialization etc

```c
sum = 0.0; delta = 1.0/slices;
MPI_Comm_rank(MPI_COMM_WORLD, &rank)
start = r*n/p; end = (r+1)*n/p;
for (i= start; i<end; i++) {
    x = (i+0.5)*delta; sum = sum+4.0/(1.0+x*x);
}
MPI_Reduce(&sum, total_sum, 1, real, MPI_sum, 0, MPI_COMM_WORLD);
pi = total_sum * delta; /* only in root or use AllReduce */
```

2 (25%). Consider sorting $n$ elements using $n (=2^d)$ processor hypercube. Processor $P_0$ sends its element to $P_1$; $P_1$ keeps the larger of the value received and the value contained at $P_1$, and returns the smaller value to $P_0$. $P_0$ then communicates with $P_1$ and receives the smaller of values, and so on. At the end of this step, $P_0$ has the smallest value. Then $P_0$ starts a similar iteration and communicates with processor $P_2$, $P_3$, ..., $P_{n-1}$; and at the end keeps the second smallest value at $P_0$. We continue in this manner with processors $P_0$, ..., $P_{n-1}$.

What is the complexity of this algorithm?

**Key.** Consider the communication steps involved in obtaining the smallest at $P_0$. 
P, communicates with of the n-1 other processors, and may involve exchange of 2 words at each step (sending and receiving). But, note the distance that a message ranges from 1, 2, ... d where d is the number of dimensions. And there are d processors at a distance of 1, there are d*(d-1)/2 processors at a distance of 2, .. and there is exactly one processor at a distance 1.

But I will simplify this and assume that the average distance is d/2
So to get the smallest value to P, we have (n-1)*2*(d/2) = (n-1)*d

To get the next smallest value to P,, we have (n-2)*2*(d/2) = (n-2)*d

Total communication cost = d*{(n-1)+(n-2)+...+1} = d*{(n-1)(n-2)/2} = O(n^2 log n)

The computation time is n-1 comparisons in the first step, n-2 in the second and so on

= O(n^2)

Total complexity = O(n^2 log n)

Not very good. The main problem is that at each step only 2 processors are active.

3 (20%). A matrix is symmetric if a_{ij} = a_{ji} (or the transpose of a matrix is the matrix itself).

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

b). Develop an algorithm to check if the matrix is symmetric using EREW PRAM. What is the complexity of your algorithm.

Key
a). Let us assume we have n threads. Each thread needs to read two values: a_{ij} and a_{ji} and compare them.
   If we have CREW, all threads can read the values in parallel
   So, we can assume that we have 2 read operations and one compare for a total of 3 operations.

   Complexity = 3 or O(1). Note sequential complexity = O(n)

Note, if you assume that the result of the test to see if the column and row elements match need to be collected at some place, we can use “reduction” to add how many time the test failed (and zero means the matrix is symmetric). If you require this, since we cannot write concurrently, we need O(p) for the write and if you have n processors, it takes O(n)

If we have n threads, each thread will read one row and one column and performs n comparisons
Here we have a total of 2n reads and n comparisons, for a complexity of 3n.

Again, if we need to collect the count of successes and failures, and if we assume each thread computes locally the number of its row-column failures, and we use a global variable for total count (using reduction), this step will take O(n).

b). Develop an algorithm to check if the matrix is symmetric using EREW PRAM. What is the complexity of your algorithm.
b). Let us once again assume \(n^2\) threads. Again each thread needs to read two elements and perform one comparison.

If we use EREW, only one thread can read a value at a time. So a thread may have to wait for \(n-1\) other threads to complete reading. So reads will take \(2*(n-1)\). Total complexity = \(2*(n-1)+1\).

Or \(O(n^2)\).

What happens if we have \(n\) threads, each thread reads \(n\) row and \(n\) column values (or total of \(2n\) values). A thread may have to wait for \(n-1\) other threads to read first. So the reads will (in the worst case) take \(2n \times (n-1) = O(n^2)\)

4 (20%) For the following code segments, show how you can parallelize in OpenMP or indicate why you cannot parallelize the code using OpenMP.

**Key**

a). 
\[
k = (\text{int}) \sqrt{x};
\]

```
#pragma omp parallel for
for (j=0; j ++; j < k {
    a[j] = 2.3 *j;
    if (j < 10) b[j] = a[j]; }
```

b).
```
for (j 0; j <n; j++) {
    a[j] = foo(j);
    if (a[j] <b[j]) break }
```

OpenMP does not permit for breaking out of a loop; since this implies that we need to kill remaining threads. If you use Pthreads, then you can parallelize this.

c). for (j=k; j<n; j++)
\[
a[j] = b *a[j-k];
\]

Here we have a dependency between loop iterations. So if you parallelize this we need to force threads to wait until previous threads have computed their values. We can use “Ordered” clause with omp parallel so that the threads execute sequentially.

```
#pragma omp parallel for
for (j=k; j<n; j++)
    #pragma omp ordered
    a[j] = b *a[j-k];
```

But you do not get any parallelism since the threads will wait for previous threads to complete.
5 (15%). Prove that if $T \leq \Theta (p)$ for a given problem size, then the parallel execution time will continue to decrease as $p$ is increased and will asymptotically approach a constant value. Likewise if $T \geq \Theta (p)$, then $T$, first decreases and then increases with $p$, hence it has a distinct minimum.

**Key.**

Note, $T = T_s/p + T_0$.

If $T \leq \Theta (p)$, then $T$, continuously decreases since $T/p$ decreases while $T_0$ grows slower than $p$.

Likewise if $T \geq \Theta (p)$, $T$, decreases first since $T/p$ decreases with $p$ more rapidly than $T_s$, and for large $p$, $T_0$ is growing faster than $T/p$ is decreasing.
1 (35%). Consider an array A of elements a₀, a₁, …, aₙ. Write an algorithm that creates a new array where
aₖ will be in position k such that
aᵢ <= aₖ for all i < k
aₖ <= aⱼ for all k < j
Note that I am not asking you to sort the list, only that use aₖ as a pivot to create two lists, where the first
lists consists of elements smaller than the pivot and the second list consists of elements greater than the
pivot.

a). What is the complexity of the algorithm?

b). Describe how you would parallelize this algorithm using OpenMP. You do not have to be exact with
your syntax, but show what OpenMP features you would be using.

Key. (a) The serial complexity is n since you have to examine each element to determine in which list it
falls. Note this is a part of Quick Sort algorithm. So, we can use the same technique to implement the
algorithm. Consider having two pointers, one starts at the top and moves down and the other starts at the
bottom and moves up. In each iteration, we compare the elements indexed by the top and bottom pointers,
and swap the elements if bottom element is less than the pivot, and the top element is greater than the
pivot. Then we move the pointers. If, say both elements are smaller than the pivot, we only move the top
pointer and compare the new elements. If both are larger than the pivot, then we only move the bottom
pointer and compare new elements.

b). If we assume that we can use two new arrays, in the parallel version, we can assign n/p elements to
each thread. Each thread places its elements into the appropriate list. The main concern is the maintenance
of the indexes for the two lists. We need to use critical sections, locks or atomic actions to do this.
Consider the simpler case where we have n threads, each dealing with one element

```c
int top=0, bottom=0, list_1[n]; list_2[n]; /* these are used to create two lists
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] < a[0])
        #omp omp critical
        {list_1[top]=a[i]; top++;}
    else
        #omp omp critical
        {list_2[bottom]=a[i]; bottom++;}
}
```

I only outlined a solution here. This is not necessarily the best, particularly when we are using each thread
to manage n/p elements. In that case, you want to have each thread create two lists, and then use critical
section to add its elements form the two lists into global lists.
Consider the matrix * vector product problem (A*x = y). The following shows pseudo code for sequential version:

```c
void function MAT_VECT (A, x, y)
{ for (i=0; i<n; i+)
  y[i] = 0;
  for (j=0; j<n; j++)
    y[i] = y[i] + A[i][j]*x[j];
}
```

Assume the matrix A is distributed using row striping and n/p rows distributed to each processor. Likewise, n/p elements of the vector are distributed to each of the p processors.

When using MPI, we need to scatter A and x to p processors. To complete computation, we need to perform all-to-all broadcast where each processor broadcasts its n/p elements of x to other processors. Finally, we need to perform a gather operation to collect the result vector y elements.

a). Write an outline of the MPI code to perform the above matrix * vector product. Again, you do not have to be exact with your syntax, but show what MPI features you would be using.

b). Assuming that you have a ring network, compute the execution time of the algorithm that includes both communication and computation costs.

**Key** (a). You have already seen the MPI version since it does not matter what the underlying network is. Also, I outlined the program; root scatters A and x elements. All processors perform all to all broadcast of their b elements, and then you complete the product computation. I will outline the functions need

```c
...  
MPI_Int..
MPI_Comm_rank..
MPI_Comm_size..
...
Read_input;
MPI_Scatter... /*scatter A
MPI_Scatter... /*scatter x
....
MPI_All_Gather.... /*this does all-to-all broadcast and each process collects data from
/* all other processes

Perform_matrix_vector_product
MPI_gather... /*collect results of y from individual processes
```

b). Note that in the serial version, the computations complexity is O(n^2)
If we use t_c for computation, then we have t_c*n^2 as the execution time.
In the parallel version, is process computes n/p result (y) elements. So the computation time of the parallel version is t_c*(n/p)

Now let us look at communication using ring network. We can view the scatter in a ring can be viewed as p-1 one-to-one communication (each message sending n/p rows of A or n/p x elements to a different process); or think of sending the entire A (or x) to the farthest node and intermediate nodes keep their portion and send the remaining elements to the other nodes.
If you see page 170 of the text, it shows that the cost of scatter (also the same as gather) in a ring as
\[ t \log p + t \cdot m \cdot (p - 1) \]

For scattering \( A \ m = n/p \) and for scattering \( x \), \( m = n/p \)
So the cost of scattering \( = 2 \cdot t \cdot (\log p) + t \cdot (n + n) \) -- ignoring -1 in p-1.

Now we have to compute the cost of all to all broadcast (or all gather). This is available in the book on page 164
\[ (t + t \cdot m)(p - 1) \] and since \( m = n/p \), we have
\[ (t + t \cdot n) \]
Finally we have a all to one gather operation to collect results. See page 170, which shows that scatter and gather can be done in the same amount of time in a ring
\[ t \log p + t \cdot m \cdot (p - 1) \]
Here \( m = n/p \) (result is a vector). So the communication cost = \( t \cdot \log p + t \cdot n \)
So, total communication cost is \( t \cdot (2 \log p + 1) + t \cdot (n + 3n) \)

3 (15%). As we have seen, scalability using Gustafson’s formula indicates how much more work can be completed using \( p \) processors in a fixed time. Consider that a single computer can solve a problem of size \( W = 100,000 \) in 15 hours.

Ignoring overhead costs, how large a problem can be solved in 15 hours (same time) if there are 64 processors if the complexity of the problem is

i). \( O(n) \)
ii). \( O(n \cdot \log n) \)
iii). \( O(n^3) \)

Key. Note we are completely ignoring communication costs, serialization etc. So the \( W \) should scale linearly. But you need to remember work \( W \) is the sequential complexity.

a) \( 64 \cdot W = 64 \cdot 100,000 \)
b) \( 64 \cdot w = 64 \cdot (100000 \cdot \log 100000) = 106,301,699 \)
c) \( 64 \cdot W = 64 \cdot (100000)^3 \)

4 (15%). Looking at the costs for all-to-all broadcast on 2-D mesh and a Hypercube with equal number of processors – see page 165 of the textbook, equations 4.3 and 4.4; also reproduced below -- show which of the two interconnection networks are better for the following values.

\[ t = 100, t_s = 1; p = 256 \]
\[ t = 10, t_s = 10; p = 1024. \]

Do you think that 2-D mesh will ever outperform a Hypercube? If not why? If yes, under what conditions?

Here are communication costs

2-D Mesh: \( 2t(p^{1/2} - 1) + t \cdot m \cdot (p - 1) \) \hspace{1cm} (4.3)
Hypercube: \( t \log (p) + t \cdot m \cdot (p-1) \) \hspace{1cm} (4.4.)

**Key.**
Using \( t = 100, t = 1; \ p = 256 \), communication costs are
- 2D Mesh = 3000+255\(m\)
- Hypercube = 799.4+255\(m\)

So, hypercube performs better for all message sizes

Using \( t = 10, t = 10; \ p = 1024 \).
- 2D Mesh = 620+10,230\(m\)
- Hypercube = 100 +10,230\(m\)

Again hypercube outperforms 2D mesh for all message sizes

2D mesh can only outperform Hypercube only if
\[ 2t(p^{1/2} - 1) < t \log (p) \]
This can never be true for any \( p > 1 \)