1. (**20 points**) Store-and-forward Routing

   Consider the routing of messages in a parallel computer that uses store-and-forward routing. In such a network, the cost of sending a single message of size m from P_{source} to $P_{destination}$ via a path of length d is $t_s + t_w \times d \times m$. An alternate way of sending a message of size m is as follows. The user breaks the message into k parts each of size m/k, and then sends these k distinct messages one by one from P_{source} to $P_{destination}$. For this new method, derive the expression for time to transfer a message of size m to a node d hops away under the following two cases. For each case, comment on the value of this expression as the value of k varies between 1 and m. Also, what is the optimal value of k if ts is very large, or if ts=0?

   A. Assume that another message can be sent from P_{source} as soon as the previous message has reached the next node in the path.

   $$\begin{aligned} Time &= k \times t_s + \frac{m}{k} \times t_w \times (d + k - 1) \\ &= k \times t_s + m \times t_w \times (\frac{d}{k} + 1 - \frac{1}{k}) \\ &= k \times t_s + m \times t_w \times (\frac{d-1}{k} + 1) \end{aligned}$$

   Note that as the value of k increases, the $k \times t_s$ term in above expression increases, while the $m \times t_w \times (\frac{d}{k} + 1 - \frac{1}{k})$ term decreases. This means that the optimal value of k (so as to minimize the transfer time) is dependent on the transfer startup cost t_s. If $t_s = 0$ then the optimal k should be as large as possible (i.e. $k = m$) to minimize the $m \times t_w \times (\frac{d-1}{k} + 1)$ term. However, if t_s is very large, then the $k \times t_s$ term might come to dominate the transfer time expression, making a small value k optimal ($k = 1$ in the limit).

   B. Assume that another message can be sent from P_{source} only after the previous message has reached $P_{destination}$.

   $$\begin{aligned} Time &= k \times t_s + \frac{m}{k} \times t_w \times d \times k \\ &= k \times t_s + m \times t_w \times d \end{aligned}$$

   As evidenced by the above expression, there is no value gained in splitting up the message into k parts when a message can only be sent after the previous message has reached its destination. In fact, under such restrictions, splitting the message up only increases the message transfer time since the startup cost t_s must be paid for each of the k messages sent. As such, the optimal value of k is 1 if $t_s \neq 0$. If $t_s = 0$, then any value of k will result in optimal transfer times as the first term in the time expression becomes zero and the second term is independent of k.

2. (**10 points**) Embedding a Ring Network in a Tree Network

   Discuss how one can map/embed the Ring to the Tree. Be detailed, perhaps annotating the diagrams to show the specific mapping. Give the numeric values for Congestion and Dilation in this mapping.

   Generalize your answer for a 2^D node Ring mapped to a 2^D node Tree. Provide an analytic expression for Congestion and Dilation. Informally justify your answers (no need for a formal proof).

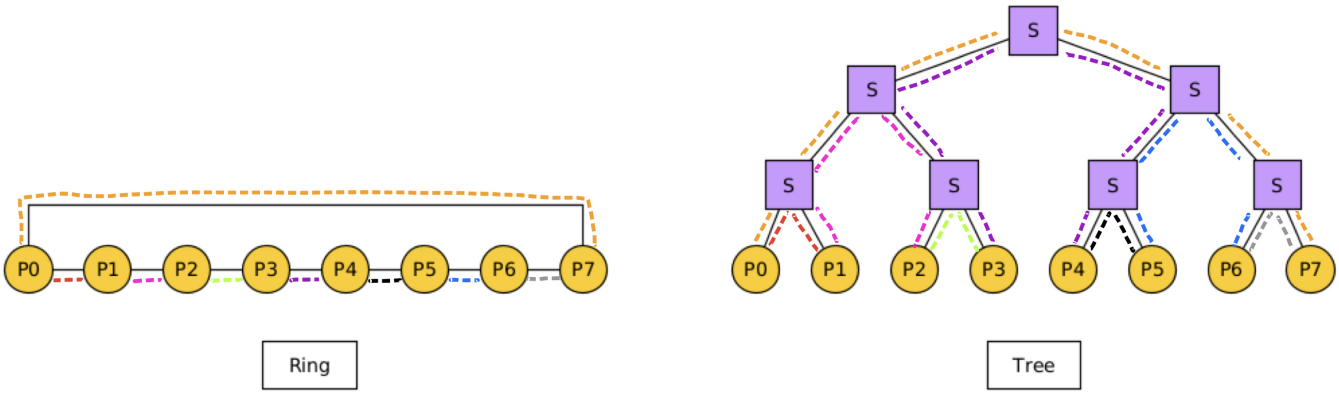


Figure 1: Annotated diagrams showing node and link mapping from 8-node ring network to 8-node binary tree network.

As shown in figure 1, there exists a simple mapping between the 8-node ring network and an 8-node binary tree network. This simple mapping has a congestion of 2 since each link in the destination network has exactly 2 links from the source network mapped to it, and a dilation of 6 with both the link between `P3`, `P4` (purple) and the link between `P0`, `P7` (orange) requiring a traversal of 6 links in the destination network. It is easy to see that no mapping could have a smaller congestion than the current mapping's congestion of 2. This is because each node in a ring network has exactly 2 links which are connected to it, whereas each node in a binary tree network has only 1 link connected to it. It follows that at least 2 links in the source ring network must be mapped to the single link which connects to one of the nodes in the destination binary tree network, forcing the congestion of the latter to be ≥ 2 (note that this reasoning is independent of the specific number of nodes in the network and so generalizes to a 2^D node ring mapped to a 2^D node tree).

With a little bit of reasoning, it is also pretty easy to show that no mapping can improve upon the dilation of this simple mapping. First, divide the binary tree network in half (break apart at the topmost intermediary switch). Notice that any two nodes which share a link in the source network, but are mapped to nodes contained in different halves of the destination network, will require a traversal of exactly 6 links in the destination network to represent the shared link–regardless of the ordering of the nodes in their respective halves–since $D = 3$ links are needed to go from a leaf in the tree to the root of the tree, and another $D = 3$ links are needed to go back down from the root of the tree to a leaf in the other half of the tree. This means that the only way to get a dilation less than the current dilation of $2 \times D = 6$ would be to ensure that any two nodes which share a link in the source network be mapped to nodes in the same half of the destination network. This, however, is not possible as such a mapping would imply that you can divide the cyclic ring network into two disjoint groups (i.e. no link exists which goes from a node in one of the groups to a node in the other group).

Hence, for a general 2^D node ring mapped to a 2^D node tree, the following expressions describe the best case congestion and dilation values which are realized by any mapping which preserves the ordering of the nodes from the ring network:

- Congestion = 2
- Dilation = 2×D

3. (**10 points**) Task-dependence Graphs

   For the task graphs given in figure below, determine the following:

   A. Maximum degree of concurrency.

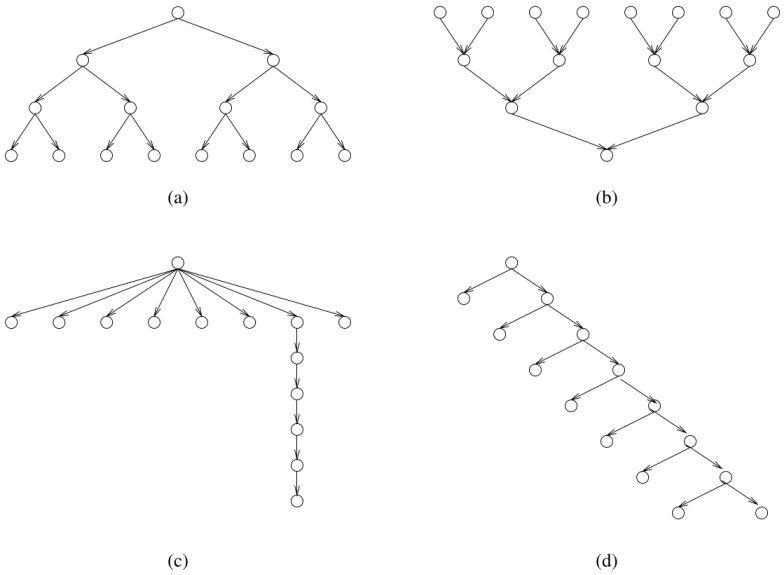
   B. Critical path length.

   C. Maximum achievable speedup over one process assuming that an arbitrarily large number of processes is available.

   D. The minimum number of processes needed to obtain the maximum possible speedup.

   E. The maximum achievable speedup if the number of processes is limited to 2, 4, and 8.

   Assume that each task node takes an equal amount of time to execute.



Task (a)
- A. 8
- B. 4
- C. $15/4 = 3.75\times$ speedup
- D. 8
- E. $2 \mapsto 15/8 = 1.875\times$ speedup
  $4 \mapsto 15/5 = 3.0\times$ speedup
  $8 \mapsto 15/4 = 3.75\times$ speedup

Task (b)
- A. 8
- B. 4
- C. $15/4 = 3.75\times$ speedup
- D. 8
- E. $2 \mapsto 15/8 = 1.875\times$ speedup
  $4 \mapsto 15/5 = 3.0\times$ speedup
  $8 \mapsto 15/4 = 3.75\times$ speedup

Task (c)
- A. 8
- B. 7
- C. $14/7 = 2.0\times$ speedup
- D. 8
- E. $2 \mapsto 14/8 = 1.75\times$ speedup
  $4 \mapsto 14/5 = 2.8\times$ speedup
  $8 \mapsto 14/4 = 3.5\times$ speedup

Task (d)
- A. 2
- B. 8
- C. $15/8 = 1.875\times$ speedup
- D. 2
- E. $2 \mapsto 15/8 = 1.875\times$ speedup
  $4 \mapsto 15/8 = 1.875\times$ speedup
  $8 \mapsto 15/8 = 1.875\times$ speedup

4. (**20 points**) Parallel Histogram

   Consider parallelizing a simple histogram algorithm on a distributed memory computer.

   - Input consists of an array `A[]` of N random integers in the range `{0...(R-1)}`.
   - The algorithm should compute the array of counts `H[]` where element `H[i]` is the total number of times integer i appeared in input array `A[]`.
   - All P processors on the parallel computer are assumed to have access to the entire array `A[]` which can be loaded from permanent storage. However if the size N gets large enough, only parts of `A[]` can be held in memory for an individual processor.
   - At the end of the algorithm execution, at least one processor must contain the entire array `H[]` which may involve some communication.

   A. Describe a decomposition based on partitioning the input data (i.e., the array `A[]`) and an appropriate mapping onto p processes. Describe briefly how the resulting parallel algorithm would work.

   First, partition `A[]` into p roughly equal contiguous sub arrays (i.e. p sub arrays each with a size of $\approx \lfloor \frac{N}{p} \rfloor$). Next, map each of the p sub arrays to one of the p processes which will then compute the local `H[]` for its assigned sub array. This computation can be done in a single pass through the sub array as shown in the following pseudocode:

```
for element in sub_array:
  H[element] += 1
```

   The p partial histogram results from each process can then be combined/reduced (all-to-one reduction) into the final, complete `H[]` by performing element-wise sums.

   B. Describe a decomposition based on partitioning the output data (i.e., the array `H[]`) and an appropriate mapping onto p processes. Describe briefly how the resulting parallel algorithm would work.

   First, partition the range `{0...(R-1)}` into p roughly equal contiguous sub ranges (i.e. p sub ranges each with a size of $\approx \lfloor \frac{R}{p} \rfloor$). Next, map each of the p sub ranges to one of the p processes which will then compute the array of counts `H[]` for its assigned sub range. This computation can be done in a single pass through the sub array as shown in the following pseudocode:

```
for element in A:
  if sub_range_start <= element < sub_range_end):
    H[element - sub_range_start] += 1
```

   Note that `H[i]` in the array of counts computed by some process is the number of occurrences of `i + sub_range_start` throughout `A[]`, where `sub_range_start` is the smallest number in its assigned range. It follows that the size of the `H[]` computed by each process is equal to the size of its assigned sub range. After each process has completed its assigned computation, the p partitions of `H[]` can be concatenated together in order of increasing range partitions to form the final result.

Continued on next page ⟶

C. Discuss the advantages and disadvantages of these two parallel formulations. Describe circumstances under which each is preferred. Such circumstances should consider

- P the number of processors
- R the number of different data possibilities (size of `H[]`)
- `A[]` the input array and n its size
- Potentially also the cost to communicate between the processors

In situations where there are a large number of different data possibilities, the decomposition based on partitioning the output described above offers notable memory advantages over the partitioning of input data, particularly in the event of a large number of available processors. This is because each process only maintains an array of counts of size equal to its assigned range partition–which is inversely proportional to P. However, this memory advantage comes at the cost of decreased parallelism and increased complexity in combining the partial outputs from each process into the final, complete array of counts since the order in which the arrays are concatenated is essential to producing correct output. Critically, the output partitioning based decomposition does not actually offer any speed advantages over a sequential, non-parallel implementation since each process must traverse the entire input array once to obtain its portion of the result. This means that the computational/communication overhead associated with the output partitioning is only justified in the event of memory limititations which might be especially apparent when R is large.

In situations where the size of the input array `A[]` is large, the described decomposition based on partitioning the input data offers both memory and speed advantages over the output partitioning decomposition. This is because each process only has to hold its assigned partition of the input array in memory, and still only performs a single pass over this smaller array to obtain its partial result. It follows that the resulting speedup from an input partitioning based decomposition is directly proportional to the number of available processors (P)–a major advantage over the output partitioning decomposition. However, if the value of R is very large, a signficant amount of memory must be allocated by each processor to hold its array of counts. For this reason, if memory limitations are a concern for either the input and output data (i.e. large R and large n) a mixed decomposition based on a partitioning of the both the input and output data might be best. Another advantage of the input partitioning based decomposition is that the order in which the partial results are combined to form the final `H[]` array of counts does not matter insofar as no partial result is missed or counted multiple times, thereby allowing a highly parallelized solution reduction.

5. (**20 points**) Parallel One-to-All Broadcast in 2D Torus

   Assume processor 0 has a message that must be sent to all other processors in a distributed memory parallel computer. Describe an efficient algorithm to do this in a 2D Torus (wrap-around mesh) with R rows and C columns.

   Account for the following in your answers.

   - A processor can send a message to any other processor. Any processor between the source and destination does not receive the message, only forwards it along the path. However, once a processor has actually received the message, it can subsequently be a source and send it to other processors.
   - Processors can be indexed with 2D coordinates as in processor (0,2) sends a message to processor (5,4)
   - Give rough pseudocode for how this algorithm will look
   - Give a cost estimate of this communication in terms of the number or rows R and columns C in the network. The cost of a single communication between nodes that are d hops away is

   $$t_{single} = t_s + t_w \times m \times d$$

   You may assume that R and C are powers of two in your analysis to simplify the expression for the total communication time t_{comm}.

   The basic idea behind the one-to-all broadcast algorithm that follows is that we first broadcast the message to each processor along one dimension of the 2D Torus using a repeated doubling scheme. Once an entire row of processors has received the message, the message is then broadcast down each of the columns of the 2D Torus, using the same repeated doubling scheme. For example, in a 4 by 4 2D Torus, processor (0,0) first sends the message to processor (0,2) in the first time step. In the second time step, then, processor (0,0) sends the message to processor (0,1) while processor (0,2) sends the message to processor (0,3). At this point, the entire first row has received the message, so the previous two steps are repeated down each of Torus's four columns, using the first element (from row 0) as the columns initial source.

   Observe the following pseudocode for a repeated doubling one-to-all broadcast:

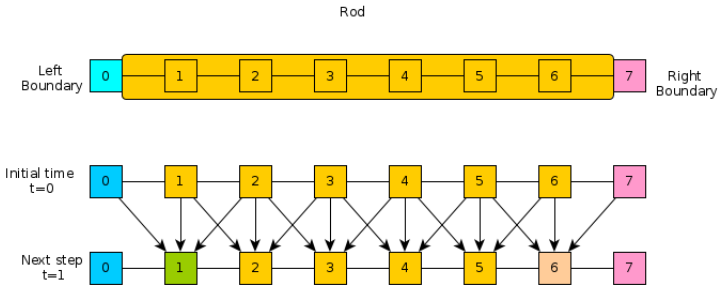
```
msg = recieve message from a source;

if (my_coord[0] == 0) { // broadcast along 1st dimension
  for (int i = (C - my_coord[1]) / 2; i > 0;) {
    send msg to processor (my_coord[0], my_coord[1] + i);
    i /= 2;
  }
}
for (int i = (R - my_coord[0]) / 2; i > 0;) { // broadcast down 2nd dimension
  send msg to processor (my_coord[0] + i, my_coord[1]);
  i /= 2;
}
```

   As implemented above, an estimate for the total communication time of the one-to-all broadcast is as follows:

$$t_{comm} = (\log_2(C) + \log_2(R)) \times t_s + t_w \times m \times \left(\sum_{i=1}^{\log_2(C)} \left(\frac{C}{2^i} \right) + \sum_{i=1}^{\log_2(R)} \left(\frac{R}{2^i} \right) \right)$$

6. (**20 points**) Bring the Heat

Consider a simple physical simulation of heat transfer. A rod of material is insulated except at its ends to which are attached two constant sources of heat/cold.



Answer the following questions about how to parallelize this code.

A. A typical approach to parallelizing programs is to select a loop and split iterations of work between available processors. Describe how one might do this for the heat program. Make sure to indicate any loops for which this approach is not feasible and any which seem more viable to you.

As a result of the dependency that exists between the rod temperature at time $t+1$ and time t (i.e. to determine `H[t+1]`, you must first determine `H[t]`), it is not feasible to perform the outer loop of the simulation portion of the program in parallel. It is possible, however, to parallelize the inner loop of the simulation portion:

```
for(p=1; p<width-1; p++){
  double left_diff  = H[t][p] - H[t][p-1];
  double right_diff = H[t][p] - H[t][p+1];
  double delta = -k*( left_diff + right_diff );
  H[t+1][p] = H[t][p] + delta;
}
```

This might be accomplished by splitting the `width - 2` iterations evenly (or as evenly as possible) between available processors. For example, if we assume `n` processors, processor 1 would execute the loop body for `p = 1` to `p = floor((width - 2) / n)` (inclusive), while processor 2 executes, in parallel, the loop body for `p = floor((width - 2) / n) + 1` to `p = 2 * floor((width - 2) / n) + 1` (inclusive), ... etc. Note that some care must be taken to ensure that all loop iterations get executed for matrix widths which are not exactly divisible by the number of available processors (ex. processor n executes any leftover loop iterations).

Similarily, the memory allocation loop, the initialize temperatures at time 0, and the initialization of the constant left/right boundary temperatures can be parallelized in the same way since, in each case, no dependency exists between loop iterations. Importantly, however, in all of these situations, each processor must initialize and update their own local variable used to track the current iteration (i.e. variables `p` or `t` depending on the loop in question).

Finally, all of the loops in the print results portion of the program cannot be parallelized since the results must be sent to `stdout` serially for meaningful output matching the described format of time steps increasing going down rows, and rod position changes going across columns.

Continued on next page ⟶

B. Describe how you would divide the data for the heat transfer problem among many processors in a distributed memory implementation to facilitate efficient communication and processing. Describe a network architecture that seems to fit this problem well and balances the cost of the network well.

Each processor would hold the columns of matrix `H` which correspond to the rod cells that are updated in its assigned partition of the simulation loop. This means that every processor (except the one that assinged to the first or last rod partition) will need to communicate with two other processors each iteration of the simulation's outer loop. As such, a simple linear array with no wraparound link network architecture would suit this problem. This division of data and network architecture would minimize the frequency of interactions between processing elements since all but two temperature values needed for each iteration of the loop body will be available locally, and these two externally located values will only be a single link away in the network.