This homework is **ungraded/optional** and meant purely to help you be better prepared for the exams. Feel free to collaborate with your fellow students in the respective *Ed Discussions* post.

Question 1 (10 points)

Parallel Efficiency You have an algorithm, FOO, that you want to run on a distributed memory system. The algorithm takes $2n^{3/2}\tau$ time on a sequential processor, where τ is the time for a single computation. The distributed version is written for p processors and is listed below:

Algorithm: Distributed-FOO

```
for i in 1 ... $\sqrt{p}$:
  Sequential–Foo(...)
  MPI_Communicate(...)
```

The communication step takes time $\alpha + \frac{n}{p}\beta$ where α is the latency and β is the inverse bandwidth of the system. Data is evenly distributed across processors.

(a) (2 points) Is this algorithm work optimal? Explain.

Solution:
Yes. Each of the p processors does $\sqrt{p}$ calls to FOO on n/p data for each call, which works out to the same amount of work as the sequential algorithm.

(b) (2 points) What is the speedup of this algorithm on p processors?

Solution:
Speedup is sequential time T_1 over parallel time T_p so we can write it as

$$S = \frac{2n^{3/2}\tau}{\frac{2n^{3/2}}{p}\tau + \sqrt{p}(\alpha + \frac{n}{p}\beta)}$$

(c) (3 points) What is the parallel efficiency of this algorithm?

Solution:
Efficiency is S/p so we get

$$E = \frac{2n^{3/2}\tau}{2n^{3/2}\tau + p^{3/2}(\alpha + \frac{n}{p}\beta)} = \frac{1}{1 + \frac{\alpha}{\tau}\frac{p^{3/2}}{2n^{3/2}} + \frac{\beta}{\tau}\frac{\sqrt{p}}{2n}}$$

(d) (3 points) What is the isoefficiency function for this algorithm? (Remember: the isoefficiency function is a lower bound on how fast n must grow as a function of p to keep constant efficiency). Explain your answer.

Solution:
We can ignore the constants in the efficiency function above. We need n to grow as some function of p to keep both the α and β terms constant. From the α term, we see that $n = \Omega(p)$. And from the β term, we see that $n = \Omega(\sqrt{p})$. We choose the larger of these, so the isoefficiency function is $n = \Omega(p)$.

Question 2 (10 points)

Bumper cars. Researchers from the Sociology department are looking for some help doing a simulation. They want to simulate a large city with millions of people. All the people in the city are to ignore one another unless they bump into someone. If they bump into someone, they will then be polite and exchange pleasantries for a minute before resuming. If someone bumps into a conversation then it ends and begins with the two people and the third continues on in a random direction. The researchers want to model this and see how the crowd reacts over time. The modeled city is similar to New York with millions of people spread throughout the city to start.

(a) (4 points) You are tasked with implementing this on the cluster. What sort of software frameworks (OpenMP, MPI or CUDA) from this class would you use for this project? Be specific about why you want to use this in your design as opposed to something else. You might want to think about **Part B** before answering this.

Solution:
There is no right answer here. This is a twist on the nbody simulation as people. So you would want some sort of multi threading through cuda, threads or omp to update every person's movement. You would also want some sort of inter process communication to shift people from ”zones” to handle localized processing and hit detection.

(b) (6 points) Write pseudo code that models how you would implement this simulation if this was a real life task. Many things are going on here that should be considered. Think about how the data and workload will be distributed. This may then influence how you design hit detection to work. Keep in mind that one node may have more people to simulate than another, so how will time slices be handled?

Solution:
Again this will be somewhat dependent on part A:

Things they should probably include: - distributing workload across nodes / cpus - shifting people to different threads / processors based on movement - how do they model a person with a vector / quaternion? - how to handle a slice of time - - handling hit detection - this could be done through something like a neighbors list using their movement vectors - handling conversations - ending conversations - how do you keep different threads / processors in time step

Full credit should handle distributing tasks, person modeling, hit detection, conversing, and shifting people through different zones(threads / processors) of the city.

Question 3 (10 points)

Polynomial Evaluation With Horner's Rule

Evaluating a degree n polynomial is the task of solving

$$P(x) = a_{n-1} * x^{n-1} + \cdots + a_2 * x^2 + a_1 * x + a_0$$

for a given list of n coefficients: $A = [a_{n-1}, a_{n-2}, \cdots, a_1, a_0]$ and some specific input value of x.

The iterative method for evaluating is a serial for loop over all the coefficients that runs in $O(n)$ time.

```
# serial polynomial eval in python
def poly_eval(A : List[], x : int):
    x_to_power_degree = 1
    value = 0
    for coeff in A:
        value += x_to_power_degree * coeff
        x_to_power_degree *= x
    return value
```

As is evident, this method runs in $\Theta(n)$ time and is not intended to scale to multiple processors.

Horner's rule however, re-interprets the evaluating a polynomial is as follows:

$$P(x) = a_0 + x(a_1 + x(a_2 + \cdots x(a_{n-1})))$$

This way of "rephrasing" polynomial evaluation looking at the computation differently allows us to parallelize the evaluation to an arbitrary number of processors.

Construct a distributed memory parallel algorithm for P number of processors (that is, all communication between any two ranks must be explicit). Assume that number of coefficients $N >> P$ and that each processor is ranked starting at zero through $P - 1$. Further assume that each rank contains its block of coefficients at the start of the algorithm, i.e. rank 0 contains coefficients $A[0 : N/P]$, rank 1 contains coefficients $A[N/P : 2N/P]$ and that N is divisible by P.

Analyze the asymptotic complexity of your parallel algorithm. Please express your solution as pseudocode or short phrases for each step, do not write actual code for your algorithm.

Solution:

Notice that we can decompose this recurrence as a sum of two recurrences as shown below:

$$\begin{aligned} P(x) &= x^0(a_0 + x^2(a_2 + x^2(a_4 + \cdots x^2(a_{n-2})))) \\ &\quad + x^1(a_1 + x^2(a_3 + x^2(a_5 + \cdots x^2(a_{n-1})))) \end{aligned}$$

Notice that both of them do not have any dependent terms from each other, and therefore can be computed concurrently. We can further generalize this into a Horner's rule that allows for P way parallel evaluation with the following formula:

$$\begin{aligned}
P(x) = {} & x^0(a_0 + x^p(a_{0+p} + x^p(a_{0+2p} + \cdots x^p(a_{n-p+0})))) \\
& + x^1(a_1 + x^p(a_{1+p} + x^p(a_{1+2p} + \cdots x^p(a_{n-p+1})))) \\
& + x^2(a_2 + x^p(a_{2+p} + x^p(a_{2+2p} + \cdots x^p(a_{n-p+2})))) \\
& + x^3(a_3 + x^p(a_{3+p} + x^p(a_{3+2p} + \cdots x^p(a_{n-p+3})))) \\
& \vdots \\
& + x^{p-1}(a_{p-1} + x^p(a_{p-1+p} + x^p(a_{p-1+2p} + \cdots x^p(a_{n-1}))))
\end{aligned}$$

The solution that follows is based on this, with P total processors ranked starting at zero.

1. All P processors run an exclusive prefix sum over input value x with multiply as the operator. After this step, each processor holds x^p where p is the rank of that processor.
2. Last processor with rank $P-1$ multiplies its local value x^{P-1} with x again to obtain x^P.
3. Broadcast x^P to all processors.
4. Compute local evaluations using received value x^P and local x^{p-1} serially and store into local partial evaluation.
5. All reduce the local partial sum with addition as the operator.

Analysis:

1. Prefix sum for calculating powers of x takes $\Theta((a+b)\log P)$ time.
2. Broadcast of x^P takes another $\Theta((a+b)\log P)$ time.
3. Local evaluation takes $\Theta(N/P)$ time.
4. All reduce takes another $\Theta((a+b)\log P)$ time.

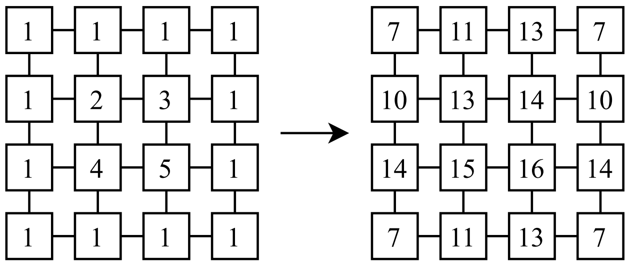
Summing all these up, our overall asymptotic runtime is $\Theta\left(\frac{N}{P} + (a+b)\log P\right)$
where a is the latency cost and b is the bandwidth cost of inter-rank communication.

Question 4 (10 points)
Cartesian Reduce

You are tasked with implementing a new kind of MPI reduce, called an *MPI_CartesianReduce*. This reduce is only defined for processors arranged in a d-dimensional grid, and is defined component-wise as follows:

$$\begin{aligned} B[i_1, i_2, ..., i_d] \leftarrow\ & A[i_1, i_2, ..., i_d] \\ & + \sum_{k=1, k\neq i_1}^{n} A[k, i_2, ..., i_d] \\ & + \sum_{k=1, k\neq i_2}^{n} A[i_1, k, ..., i_d] \\ & + ... \\ & + \sum_{k=1, k\neq i_d}^{n} A[i_1, i_2, ..., k] \end{aligned}$$

In words, this means that each processor ends up with the sum of the elements that vary in exactly one or zero indices. Let's take a look at an example in two dimensions:



We see that each processor ends up with the sum of all of the elements in its row, and all of the elements in its column.

(a) (4 points) Design a distributed memory algorithm to compute the *MPI_CartesianReduce* for a d-dimensional torus, where each dimension has length p, and each processor needs to send n words. Pseudo-code is not required, but you provide enough detail so that your answer to part (b) is clear.

Solution:
Do d rounds of allreduce, each allreduce is over p processors.

(b) (4 points) What is the time spent on communication, $T(n, p, d)$? Explain.

Solution:
A single allreduce is either $O(logp(\alpha + n\beta))$ for tree based or $O(p\alpha + n\beta)$ for the bandwidth optimal version. The final answer should be either of those, scaled by d.

(c) (2 points) Your friend has designed a new architecture that will allow each node to receive d messages and send d messages at once. How can you use their architecture to write a faster version of this algorithm? How does the communication time change?

Solution:
You can now do each allreduce in parallel, leading to factor of d improvement in the latency.

Question 5 (10 points)

Uh oh...something is wrong. One of the nice features of MPI is that it includes a lot of primitives that we can use. But what happens if you find out they aren't working correctly? In particular, suppose you're trying to implement an algorithm and find out that *MPI_Barrier()* no longer works.

For this problem, assuming the following pieces of information.

1. The number of nodes is P.
2. The communicator is $comm$.
3. The rank of a processor is $rank$.

(a) (6 points) Give detailed pseudocode for your own *MPI_Barrier()* using point-to-point operations. Your pseudocode should be reasonably detailed, that is, close to being "compileable."

Solution:
This is a basic solution to the problem (there are likely many)

```
procedure MPI_BARRIER(...)
    int magic = 454;
    int magic_recv[P];
    MPI_Request requests[P*2];
    for $i \leftarrow 0$ to $P$ do
        MPI_Isend(&magic, 1, MPI_INT, i, magic, comm, requests+i);
        MPI_Irecv(magic_recv+i, 1, MPI_INT, i, magic, comm, requests+i+P);
    MPI_Waitall(P*2, requests, MPI_STATUSES_IGNORE);
    return MPI_SUCCESS;
```

(b) (2 points) What do you think the asymptotic lower bound is?

Solution:
Since this could be implemented with something as simple as MPI_CHAR (which is one byte) we're looking for something that recognizes that this will be limited by the latency.
$T_{msg}(n) = O(\alpha)$

(c) (2 points) Why does your algorithm work and reach that bound?

Solution:
We're looking for a well reasoned argument here.

Question 6 (10 points)
Dense Matrix Algorithms

(a) (5 points) Consider the matrix-vector multiplication operation, $y = Ax$, where A is an $n \times n$ matrix and x and y are $n \times 1$ vectors. Each output y vector element may be computed as $y[i] = \sum_{j=0}^{n} A[i,j] \cdot x[j]$
Suppose that the matrix is partitioned with 1D *column-wise* partitioning, such that each processor stores $\frac{n}{p}$ complete columns of the matrix A and has matching $\frac{n}{p}$ elements of the vectors x and y, i.e., if a processor holds column j (i.e., $A[*,j]$), then it also owns $x[j]$ and $y[j]$. Design an efficient distributed memory parallel algorithm to compute matrix-vector multiplication in parallel. Analyze the run-time of this algorithm and specify the computation and communication time.

Solution:

- Each processor computes partial local dot products with local A and x values, and stores them in a local vector $y^{'}$
  Computation: $O(\frac{n^2}{p})$
- Using All-to-all reduction, the final distributed y vector is computed from local $y^{'}$ vectors.
  Computation: $O(n)$
  Communication: $O(\tau \log p + \mu \frac{n}{p} p)$ = $O(\tau \log p + \mu n)$

Total runtime = $O(\frac{n^2}{p} + \tau \log p + \mu n)$

(b) (5 points) Let A be an $n \times n$ matrix. The transpose of A, denoted A^T is defined as $A^T[i,j]$ = $A[j,i]$. Design a parallel algorithm to compute A^T from A using an $n \times n$ virtual torus topology. During the execution of the algorithm, each processor can hold at most a constant number of matrix elements at a time (can be greater than one). Specify the total runtime for your algorithm.

Solution:
In a $n \times n$ torus, the maximum distance $D_{\text{(i,j)}\rightarrow\text{(j,i)}}$ an element $A[i,j]$ has to travel to destination (j,i) is $D_{\text{(i,j)}\rightarrow\text{(j,i)}}$= $2 \times min((j-i) \mod n, (i-j) \mod n)$ To see this, note that with wraparound the distance $(j-i)$ (assume $j \geq i$ without loss of generality) can be represented two ways: $(j-i)$ or $n-(j-i)$. This distance must be covered twice, corresponding to rows and columns. It follows from the above equation that the number of steps to transpose any element is bounded by
by $2 \times \lfloor \frac{n}{2} \rfloor = O(n)$ Therefore, $D_{\text{(i,j)}\rightarrow\text{(j,i)}}$ = $O(n)$
We now propose an algorithm that achieves transposition within n step
Move each element $A[i,j]$

- North/South $(i-j) \mod n$ steps
- East/West $(j-i) \mod n$ steps

Note that, $(i-j) \mod n + (j-i) \mod n \leq n$ algorithm terminates in n steps
Furthermore, in this algorithm each processor receives at most 4 elements one from each direction before passing elements along, so each processor holds constant number of matrix elements max (i.e. 5).

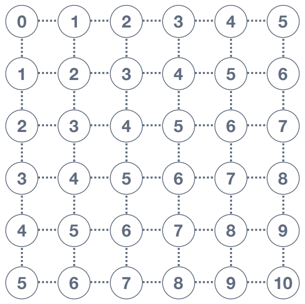


Figure 1: Results of a breadth-first search on a 2-D grid graph. The source is the upper-leftmost vertex. Each vertex is labeled by its level as computed by the BFS.

Question 7 (10 points)

BFS cache analysis. Suppose you run a sequential breadth-first search (BFS) on a two-dimensional (2-D) *grid graph* with n vertices. An example appears in fig. 1, where the $n = 36$ vertices are laid out on a 2-D grid with regular spacing, and each vertex is connected by edges to its north, south, east, and west neighbors. (Though the example is square, do **not** assume a square grid graph in this problem.) If you start from the upper-leftmost vertex and label each unseen vertex by its level during the search, you will get the labels shown in fig. 1.

In the following exercises, you will analyze the cache behavior of BFS on grid graphs for a machine with a two-level memory hierarchy. Let Z be the size of the cache and L be the line size. Furthermore, assume that the vertices in a row are laid out consecutively in memory, and *ignore* the storage cost of edges. That is, suppose we don't need to store edges at all and that you can determine location in memory of any neighbor of a vertex in $\mathcal{O}(1)$ time.

(a) (5 points) For what sizes n would you expect good cache behavior? "Good" means no thrashing of cache lines due to insufficient capacity. Do **not** assume anything about the order in which the algorithm visits the neighbors of a vertex. Explain your reasoning.

Solution:

The key insight derives from remembering that the working set of BFS is the *frontier* that it maintains. Since you are told you cannot make any assumptions about the visit order, in the worst case you might need to keep the entire frontier in cache.

For a 2-D grid graph, the *planar separator theorem*, which you saw in graph partitioning, tells you the frontier cannot be larger than $\mathcal{O}(\sqrt{n})$. (Can you figure out why?) Therefore, as long as you have enough cache lines to hold the frontier, i.e., $\sqrt{n} = \mathcal{O}(\frac{Z}{L})$ or $n = \mathcal{O}((\frac{Z}{L})^2)$, then there is enough capacity to avoid thrashing.

(b) (5 points) Suppose the graph is, instead, a **three-dimensional** (3-D) grid graph, again with n vertices. For what value of n would you expect to see cache thrashing due to a lack of capacity?

Solution:

Separators for 3-D grids scale like $\mathcal{O}(n^{2/3})$. Thus, by the same reasoning as above, $n = \mathcal{O}((\frac{Z}{L})^{3/2})$.

Question 8 (10 points)

Distributed 3D fast Fourier transforms (FFTs). Consider a supercomputer that can execute up to R_{max} operations per unit time. (That is, an embarrassingly parallel program that performs no communication and moves no data through the memory hierarchy will execute at this rate.) Further suppose that the architecture of this machine is a distributed memory network of P nodes connected by a 3D torus. Let the time to send a message of size m words between any two nodes, in the absence of congestion, be m/β_{link}, where β_{link} is the bandwidth (words per unit time) of a network link. (That is, we simplify our usual (α, β) model by *ignoring* any latency term, α.) Within each node, suppose there is a two-level memory hierarchy with a fast memory of size Z and main memory bandwidth of β_{mem} words per unit time. Lastly, assume that the node itself can execute up to R_0 operations per unit time, so that its balance (within the node) is $B_0 \equiv \frac{R_0}{\beta_{\text{mem}}}$ ops per word.

Suppose you wish to compute a 3D FFT. Here are its main performance characteristics.

- Let n^3 denote the size of the input. The output will be of the same size.
- The total work (e.g., arithmetic operations) is $\mathcal{O}(n^3 \log n)$.
- Let the parallel execution time be $T \geq \max\{T_{\text{comp}}, T_{\text{mem}}, T_{\text{net}}\}$, where T_{comp} is the time to perform just the arithmetic operations; $T_{\text{mem}} = T_{\text{mem}}(n; Z, \beta_{\text{mem}})$ is the time spent on local (intranode) communication between slow and fast memory; and $T_{\text{net}} = T_{\text{net}}(n; \beta_{\text{link}}, P)$ is the time spent on network communication. This equation says the running time is, in the best case, the largest of these three components, assuming arithmetic, local data movement, and network communication can all be completely overlapped.
- It is possible to show that $T_{\text{mem}} = \mathcal{O}\big(\frac{n^3 \log n}{P \beta_{\text{mem}} \log Z}\big)$ and $T_{\text{net}} = \mathcal{O}\big(\frac{n^3}{P^{2/3} \beta_{\text{link}}}\big)$.

With this background, answer the following questions.

(a) (1 point) Assume the arithmetic work is perfectly parallelizable. Calculate T_{comp}, in a big-O sense.

Solution:
The time is just the arithmetic work divided by the machine peak.

$$T_{\text{comp}} = \mathcal{O}\Big(\frac{n^3 \log n}{R_{\text{max}}}\Big) \tag{1}$$

(b) (2 points) Next, show that $T_{\text{mem}} = \mathcal{O}\big(\frac{B_0}{\log Z} \cdot T_{\text{comp}}\big)$.

Solution:
If the peak is R_{max}, then $P = \frac{R_{\text{max}}}{R_0}$. Thus,

$$\begin{aligned} T_{\text{mem}} &= \mathcal{O}\Big(\frac{n^3 \log n}{P \beta_{\text{mem}} \log Z}\Big) = \mathcal{O}\Big(\frac{n^3 \log n}{\frac{R_{\text{max}}}{R_0} \beta_{\text{mem}} \log Z}\Big) = \mathcal{O}\Big(\frac{B_0}{\log Z} \cdot \frac{n^3 \log n}{R_{\text{max}}}\Big) && (2) \\ &= \mathcal{O}\Big(\frac{B_0}{\log Z} \cdot T_{\text{comp}}\Big), && (3) \end{aligned}$$

since $B \equiv \frac{R_0}{\beta_{\text{mem}}}$ and using the result for T_{comp} from part (a).

(c) (2 points) Define "network node balance" for a 3D FFT as $B_{\text{net}} \equiv \frac{R_0^{2/3}}{\beta_{\text{link}}}$. Ignoring the $2/3$ exponent for the moment, briefly explain what this quantity tells you about a system.

Solution:
Ignoring the $2/3$ exponent, this ratio is like "operations per word," but for the node's peak arithmetic throughput compared to its network link bandwidth. Therefore, its a measure of balance with respect to the network, and tells you how many arithmetic operations a node can do in the time it takes to move a word of data into or out of the node.

(d) (2 points) Derive an expression for T_{net} in terms of B_{net}.

Solution:
With a little algebra, one can show that

$$T_{\text{net}} = \mathcal{O}\big(B_{\text{net}} \cdot \frac{n^3}{R_{\max}^{2/3}}\big). \tag{4}$$

(e) (3 points) Assume that communication dominates, i.e., $T_{\text{comp}} \ll T_{\text{mem}}$ and $T_{\text{comp}} \ll T_{\text{net}}$ for most supercomputers you can build from today's parts. Now suppose you have a choice of two supercomputers with the same $R_{\max}$. One uses high-end server-grade processors and the other uses processors more commonly found in mobile phones. Which system is likely to be faster, and why?

Solution:
Since communication dominates and both $T_{\text{mem}} \propto B_0$ and $T_{\text{net}} \propto B_{\text{net}}$, the system with the lower execution time will have smaller B_0 and B_{net} values. In today's systems, the typical B_0 and B_{net} values for server-grade processors is larger than those of mobile phones. Therefore, you should prefer the mobile phone system.

Question 9 (10 points)

Six Degrees of HPC. Last semester, you worked closely with the sociology department to model some behavior in New York City. People were randomly placed in the city and then set in motion. If they happened to collide with someone then the two people would stop and talk to each other and then move on after a few seconds. Since it is summer, the sociology department wants to do some low key additions to this project. When the two people bump and talk, they also get to know each other thus making a connection. Your colleagues in the sociology department are interested to see how this network grows over time. Specifically, they want to see how many people lie within the 6 degrees of separation from one another. It should be noted that this is all simulated using several nodes on the cluster communicating using MPI.

(a) (6 points) Write **pseudocode** that determines how many people in the simulation lie within six degrees of separation from each other.

Solution:
This should describe some distributed breadth first search. Every person is a graph vertex. The connections they develop are then used as edges. You can then iterate through that using the lecture method or the paper matrix method. The key though is stopping at six levels from any one node. The difference from the total people being the people who are not within the six degrees of separation.

(b) (2 points) What is the **work** and **span** for your proposed solution?

Solution:
Work: O(—M— + —N—) edges and vertices not N Span: $O(d * log^r(|M| + |N|)$

(c) (2 points) What other aspects of the design might influence performance?

Solution:
This should mainly go into the costs of the alpha and beta communication costs which are not necessarily included in a basic big O analysis of an algorithm but which will influence the ov

Question 10 (10 points)

Red vs Blue. Consider n balls distributed evenly across p processor nodes, each colored either ***RED*** or ***BLUE***. We want to ensure that each continuous sequence of r balls has at least $\underline{s}$ ***BLUE*** balls (assume $n > r > s$).

(a) (8 points) Design a parallel **algorithm** to check if the condition is *true* for every continuous sequence of length r.

Solution:

- step a: Create a size n array called A, distributed evenly across all processors. Let c_i be the color of ball i. Assign $A[i]$ as follows:

$$A[i] = \begin{cases} 1 & if\ c_i = blue \\ 0 & if\ c_i = red \end{cases}$$

  Runtime complexity: **Computation–** $O(\frac{n}{p})$

- step b: Apply parallel prefix sum over A and store the results in a new array, S.

  Runtime complexity: **Computation–** $O(\frac{n}{p}+\log p)$, **Communication–** $O((\tau+\mu)\log p)$

- step c: For all $i \leq n-r$, send $S[i]$ to the processor holding $S[i+r-1]$. Note that this implies each processor sends **at most** $\frac{n}{p}$ total elements. Furthermore, each processor sends to **at most** 2 other processors, so at most 2 rounds of communication are needed.

  Runtime complexity: **Communication–** $O(\tau + \mu\frac{n}{p})$

- step d: Create an array T such that–

$$T[i] = \begin{cases} 1 & if\ i < r-1\ or\ S[i] - S[i-r+1] \geq s \\ 0 & otherwise \end{cases}$$

- step e: Perform parallel reduction on T using the AND operator. Condition is true if the final result is 1, false otherwise.

  Runtime complexity: **Computation–** $O(\frac{n}{p}+\log p)$, **Communication–** $O((\tau+\mu)\log p)$

(b) (2 points) **Analyze** your algorithm separately in terms of both **work** and **span** complexities.

Solution:

Based on the runtime complexity of each of the steps above, the final complexity analysis is bounded by and is as follows:

- **Computation–** $O(\frac{n}{p} + \log p)$
- **Communication–** $O((\tau + \mu)\log p + \mu\frac{n}{p})$

Question 11 (10 points)

VeryOP. Let's start by defining a new reduction operation primitive and calling it op.

Now imagine we have a p processor system where each processor node has an array of size p stored local to itself. If we perform our reduction operation primitive, op, on our p processor system, the processor node at the i^{th} rank will have the reduction of all the i^{th} elements of all the processor nodes' arrays.

Design an efficient parallel algorithm for this reduction operation primitive and then compute its computation cost/runtime complexity.

(a) (8 points) Provide your parallel algorithm design.

Solution:
Let the array on each processor be A_i. This problem can be solved in $log\ p$ iterations as follows:

- step 1: Divide the p processors into two groups such that $P_0\ to\ P_{p/2-1}$ belongs to the left group and $P_{p/2}\ to\ P_{p-1}$ belongs to the right group. (2 points)
- step 2: Every P_i in the right group sends $A_i[0,\ \frac{p}{2}-1]$ to P_j in the left group, where $j = i - \frac{p}{2}$. (1 points)
- step 3: Similarly, every P_j in the left group sends $A_i[\frac{p}{2},\ p-1]$ to P_i in the right group, where $i = j + \frac{p}{2}$. (1 points)
- step 4: Then every processor executes and computes the reduction of the received elements with its own local A_i using the reduction operator, op. (2 points)
- step 5: The left and right groups represent subproblems on $p/2$ processors using their respective parts of A(i.e., $A_i[0,\ \frac{p}{2}-1]$ and $A_i[\frac{p}{2},\ p-1]$). Lastly, repeat the above steps on each group separately until each processor has one element. (2 points)

(b) (2 points) Find the **work** and **span** of the algorithm.

Solution:
The run time complexity is given by:

- Computation cost = $O(p/2 + p/4 + ...1) = O(p)$.

  (2 points for getting the Computation cost correctly.)

Question 12 (10 points)
RANKrank. Let's start with our parallel **SampleSort** algorithm.

We intend to feed our algorithm n elements such that all of these elements are distinct from one another and they are *globally* in decreasing order with respect to the order of our processor nodes. Our algorithm is going to utilize p processor nodes and each processor will initially have $\frac{n}{p}$ elements locally stored in them (assume n is divisible by p).

Our **SampleSort** is run to sort our input elements in *ascending* (increasing) order. Now Compute the number of input elements in each processor, after running our algorithm, as a function of the processor rank (i.e. i).

(a) (10 points) As mentioned, **compute** the number of input elements per processor (as a function of/based-on the processor rank/number) and provide **detailed** steps for doing so.

Solution:
The steps for finding the number of elements per processor are as follows:

- step 1: First each processor will sort its local array and choose $p-1$ local splitters. These local splitters will be evenly spaced out in the sorted local array at local indices $k \cdot \frac{n}{p^2} - 1$ for k from 1 to $p-1$. Since sorting the our input list (feed) is equivalent to reversing it (since they are initially in globally *descending*/decreasing order), these splitters have indices $\frac{n}{p} - 1 - (k \cdot \frac{n}{p^2} - 1) = \frac{n}{p} - k \cdot \frac{n}{p^2}$ in the original unsorted local array.

  Also the last local splitter has $k = p-1$ and has index $\frac{n}{p} - (p-1) \cdot \frac{n}{p^2} = \frac{n}{p^2}$ in the original local array, which corresponded to index $i \cdot \frac{n}{p} + \frac{n}{p^2}$ in the unsorted global array for processor i.

- step 2: Following the sample sort process, these local splitters are globally sorted, which amounts to globally reversing the sets of candidate splitters from each process. That is, processor 0 will have the $p-1$ smallest global splitter candidates which came from the last processor; the second process will have the next smallest $p-1$ candidates which came from the second to last processor and so on.

- step 3: When sample sort chooses the last candidate splitter on each processor to be a global splitter, this corresponds to the last local splitter from the opposite process. I.e., processor j will choose the last local splitter from processor $i = p-1-j$ to be a global splitter. This is while the last process does not choose a global splitter, so the last local splitter from process 0 is never used.

- step 4: Therefore the global splitters occur at global indices $i \cdot \frac{n}{p} + \frac{n}{p^2}$ in the original array for i from 1 to $p-1$. As mentioned, sorting the global array amounts to reversing these splitters. This makes the splitter on processor j be at index $n - 1 - ((p-1-j) \cdot \frac{n}{p} + \frac{n}{p^2})$ for j from 0 to $p-2$. This simplifies to $(j+1) \cdot \frac{n}{p} - \frac{n}{p^2} - 1$.

- step 5: Processor 0 will end up with all the elements up until the first splitter. Processor j will end up with all the elements up until the splitter at $j-1$. Processor $p-1$ will additionally have the elements up to the end of the array.

Therefore the number of input elements per processor after the conclusion of our Sample sort algorithm, as a function of the processor rank (where it's denoted as i) is given by:

$$f(i) = \begin{cases} \frac{n}{p} & 0 < i < p-1 \\ \frac{n}{p} - \frac{n}{p^2} & i = 0 \\ \frac{n}{p} + \frac{n}{p^2} & i = p-1 \end{cases}$$

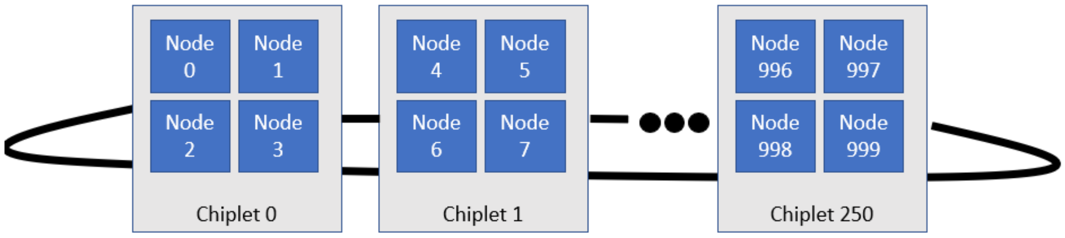
Grading:
In terms of points allocation and grading, following the exact steps outlined in the solution is definitely not necessary. Getting to the final answer for the number of elements per processor is what matters for getting the question correct and all the points, unless the approach is completely and utterly wrong. Also not getting any of the answers correct does not mean a 0 for the question– there will be partial credit reflected on the rubric.
For the final answer the following point allocations stand:

- +4 points –the intermediate processor nodes' element count.
- +3 points –first node's element count.
- +3 points –last node's (processor $p-1$) element count.

Question 13 (10 points)

Communication Madness. We studied several distributed memory topologies such as the ring, 2-D mesh, hypercube, and fully connected. Of course in real world supercomputers (like our PACE cluster), the actual topology is hybrid, complicated, and messy.

Consider a machine with P=1000 nodes, where each a node is a processor on a chiplet of four CPUs. Within any one chiplet, the four processors can communicate with extremely low latency and high bandwidth, call it α_{chip} and β_{chip}. However, all the four node chiplets are connected in a ring network (250 total chiplets) and those inter-chiplet connections have regular $\alpha + \beta n$ cost.



Your task: Outline, in high-level pseudocode, a personalized all-to-all messaging algorithm that is efficient for this topology.

(a) (6 points) **Describe your algorithm** using high-level **pseudocode**. That means, no variable declarations or C syntax; just provide sufficient detail to understand your approach.

Solution:
There are a couple ways to approach, but in the end an efficient solution will split into two problems: the on-chiplet and then the ring communication. The chiplet is fully-interconnected with very low cost. Ring all-to-all is discussed in the lectures with a shift approach. So an approach could be:

```
point-to-point (effectively instant) on all chiplets
for i = 1 to P/4
  shift pass data from node 0->4, 4->8, etc.
point-to-point all within all chiplets
```

The key fact is P/4 rounds.
Note: Attempts to use broadcast or scatter are doomed. Nodes can't hold 1000x their base data, and it result in horrible congestion on a ring,

(b) (4 points) **Analyze** the efficiency of your algorithm in terms of alpha and beta model, but using α_{chip} and β_{chip} to note intra-chiplet communications. Remember that:
$\beta_{chip} << \alpha_{chip} << \beta << \alpha$
Hint: you can just write AlphaChip, BetaChip, Alpha etc. in Canvas as text.

Solution:
The base communications as described in book and lecture is $O(\alpha P + \beta n P)$. Then add a constant times $\alpha_{chip} + \beta_{chip}$ if point to point (can use tree or other).

Question 14 (10 points)

Projection. You have learned about the 2D distributed Matrix Multiplication algorithm in this class. In practice, instead of computing and forming new matrices explicitly, the matrix as an operator could be applied to data in multiple steps.

Consider the operation AA^Tx, where matrix A is of size $n \times k$, and x a vector of size n, and A^T denotes A's transpose.

In the real world $n >> k$ so we will assume k is a constant, and the problem size is n.

(a) (1 point) What is the **optimal** amount of floating point operations required to complete this computation in serial? Give the precise number in terms of n and k, **do not use big-O notation**.

Solution:
$4nk - n - k$

(b) (6 points) Now we have p distributed processors and p is a perfect square. **Describe a parallel algorithm** to compute AA^Tx.

You may arrange p processors in an arbitrary grid/topology if you would like, and partition/store A in the processes in any way you want. Do not sweat about optimality, but clearly state the partition of matrix, process network layout, and use communication primitives like *send, recv, broadcast*, etc. you learned from MPI.

Analyze runtime of your algorithm $T(n, p)$ with respect to problem size n and available workers p.

Solution:
No standard solution, focus on correctness of students' algorithm and runtime analysis, as long as the algorithm itself is reasonably parallel, and all presumptions clearly stated.

(c) (3 points) Now suppose we have an algorithm that runs in $T(n, p) = \Theta(\frac{nk}{p} + k \log p)$, what is the largest value of p that maintains the optimal $\Theta(1)$ efficiency?

Solution:
$\frac{T(n,1)}{pT(n,p)} = \Theta(1) \Rightarrow \frac{\Theta(nk)}{\Theta(nk+kp\log p)} = \Theta(1) \Rightarrow \Theta(p \log p) \in \Theta(n) \Rightarrow p \in \Theta(\frac{n}{\log n})$

Question 15 (10 points)

SortAndSolve. This question has two parts. The first is meant to serve as motivation and help for the second.

(a) (4 points) **Show the steps** for bitonic sorting the following 8-elements in non-increasing order. You can for example use lines to denote comparison operations- but there are no hard requirements as long as the correct comparisons are mentioned. You can also for example label the comparison operations as ↑ or ↓ where the direction of your arrow indicates the destination of the smaller element (or bigger element but make sure to mention if not following the default assignment). Illustrate how the following input is sorted using a text diagram:

$$(10, 14, 9, 6, 17, 3, 8, 4)$$

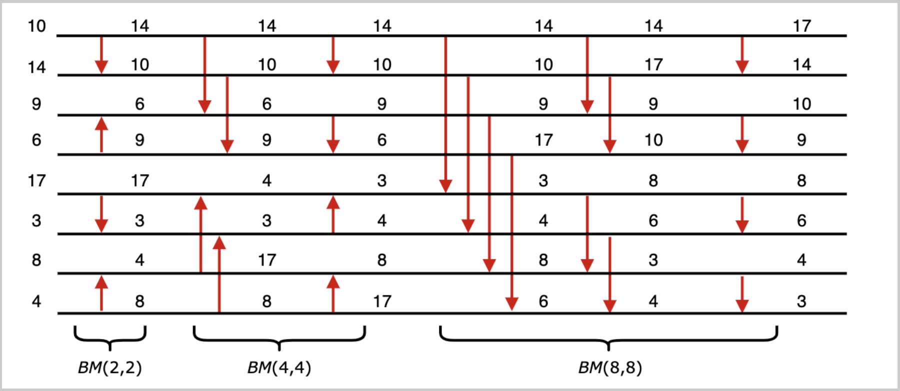
NOTE: Here's an example of what to write and what to mention (position of the comparisons in this example are totally random) given the constraints of *Canvas*.

Step 0: $5, 6, 9, 23, 1, 2$

Step 1: $5, 23, 9, 6, 2, 1$ (23 compared with 6, 2 with 1) (or you can use index starting with 0)

Step 2: $23, 5, 9, 6, 2, 1$ (23 compared with 5)

Solution:



(b) (4 points) Now **provide an efficient parallel algorithm** to merge two sorted sequences of lengths m and n, respectively. You may assume that the input is an array of length $m + n$ with one sequence followed by the other, distributed across processors such that each processor has a sub-list of size $\frac{m+n}{p}$.

Solution:
Without any loss of generality, we can assume that $m >= n$. Reversing the second sorted sequence results in a bitonic sequence of length $m + n$. To reverse the second sorted sequence, each processor need to communicate with at most 2 processors and the communication time will be $O(\tau + \mu(m + n)/p)$.

(c) (2 points) What is the **work** and **span** of your algorithm? (*You may assume the use of any permutation style communication, not necessarily hyper-cubic for example.*)

Solution:
Bitonic merging of a sequence of length $m + n$ using p processors takes $O((\tau + \mu(m + n)/p)logp)$ communication time (span) and $O((n + m)/plogp)$ computation time (work).

Question 16 (10 points)

Parallel Distributed Scan. As we have seen, the general running time $T_P(n)$ for a parallel scan is $O(\frac{n}{P} + \log^2 P)$ if we have fewer than $O(n)$ processors. This algorithm exhibits strong scalability - whenever $\frac{n}{P} \geq \log^2 P$, the speedup is linear in P. However, there are practical limitations to the number of processors we can fit on a single node. If we want to get a 500x or 1000x speedup, we have to expand to more than one node to do so. How does parallel distributed scan perform on multiple nodes?

(a) (4 points) **Design a parallel add-scan algorithm** to be run on a system with P_d *distributed* nodes and P_l processors *local* to each node. The system is connected via a 2D grid interconnect. **Do not provide pseudocode**.

Solution:

We can augment the addScan procedure from lecture as follows. First, on each node, we locally partition the data over the P_l processors. Each processor does a linear scan of its data. Next, each node performs the recursive scan, accumulating partial sums until we hit the base case. At this point, one processor on each node contains the accumulated sum, σ_i of all its local data elements.

Next, we perform a scan across the P_d nodes. At the end of this scan, each node will have the $+$-scan, S_i, up to and including itself. $S_i - \sigma_i$ is the prefix sum of the values in the nodes prior to node i. We save this sum for later, and in the meantime we proceed back up the recursive stack on each node, applying the second half of the local scan. Lastly, in parallel, each processor applies the node-wise prefix sum to its partition of the local data.

(b) (3 points) What are the **computation** and **communication** times of your algorithm? Provide your answers in terms of n, P_d, P_l, and, where appropriate, α and β.

Solution:

On each node, the addScan takes time $T_{P_l}(\frac{n}{P_d}) = O(\frac{n}{P_d P_l} + \log^2 P_l)$. The node-wise scan takes time $O(\log^2 P_d)$, so the time taken overall is $T_{comp}(n, P_d, P_l) = \tau \cdot O(\frac{n}{P_d P_l} + \log^2 P_d + \log^2 P_l)$.

Communication only happens during the node-wise scan in the middle. Each round, half of the nodes send a message to, and later receive a message from, the other half of the nodes. Then the overall maximum number of messages sent (only two nodes make it to the final round) is $2 \log P_d$. Similarly, each of these messages is a single number, so the number of words sent is $2 \log P_d$. The overall communication time is then $T_{comm}(n, P_d, P_l) = 2\alpha \log P_d + 2\beta \log P_d$.

(c) (2 points) **What is the isoefficiency function** of your algorithm? How does it compare to **scalability** of the PRAM add-scan algorithm mentioned above? For what relationship(s) between P_d and P_l does it approach the PRAM add-scan? *Helpful hint: when computing efficiency from speedup, note that your system has $P_d \cdot P_l$ total processors.*

Solution:

Since we don't know whether the communication or computation time takes longer (the distributed add-scan can be overlapped with the second half of the local add-scan which proceeds back up the local recursive stacks), the overall running time is the initial linear scan, $O\left(\frac{n}{P_d P_l}\right)$, the max of the communication time and the parallel and distributed scans $O\left(\max\left(2(\alpha+\beta)\log P_d, \log^2 P_d + \log^2 P_l\right)\right)$, and the final parfor application of the prefix sum $O\left(\frac{n}{P_d P_l}\right)$. The parallel speedup is then given by

$$S(n, P_d, P_l) = \frac{n}{\frac{n}{P_d P_l} + \max\left(2(\alpha+\beta)\log P_d, \log^2 P_d + \log^2 P_l\right)}$$

This simplifies to

$$S(n, P_d, P_l) = \frac{1}{\frac{1}{P_d P_l} + \max\left(\frac{2(\alpha+\beta)\log P_d}{n}, \frac{\log^2 P_d + \log^2 P_l}{n}\right)}$$

The efficiency, $S(n, P_d, P_l)/P_d P_l$, is then

$$E(n, P_d, P_l) = \frac{1}{1 + \max\left(\frac{2P_d P_l(\alpha+\beta)\log P_d}{n}, \frac{P_d P_l(\log^2 P_d + \log^2 P_l)}{n}\right)}$$

In order to make each term in the denominator constant, we must have that

$$n = \Omega(P_d P_l \log P_d)$$

and

$$n = \Omega(P_d P_l(\log^2 P_d + \log^2 P_l))$$

respectively. Since $P_d P_l(\log^2 P_d + \log^2 P_l) = \Omega(P_d P_l \log P_d)$, then our isoefficiency function is $n = \Omega(P_d P_l(\log^2 P_d + \log^2 P_l)$.

This is not too far off from the original scalability of PRAM addScan, which is $n = \Omega(P \log^2 P)$. Indeed, if $P_l >> P_d$, then we see the isoefficiency starts to look closer to the original PRAM - which makes sense, given that when $P_l >> P_d$, the system looks more and more like a PRAM in the first place. As P_d starts to dominate, then we see a slightly less advantageous relationship form, as the communication time begins to dominate the other smaller parallel scan factors. This also makes sense, as $P_d >> P_l$ starts to look like a parallel scan over d nodes in a network.

(d) (1 point) What if your algorithm needed to be run on a linear network? How would it change?

Solution:
Since a linear network can be logically embedded in a 2D grid, the algorithm - and its communication time - does not change.

Question 17 (10 points)

BeastMode Returns! You're working for a big social networking company, *Squawker*. It has an enormous dataset spread across a thousand processor nodes consisting of *unordered* messages of the form: keyword, squawk content. Example:

"#OMSCS","HPC is the best class!"
"#MOVIES","I wish Indiana Jones stopped at three"
"#ELON","I hope he doesn't buy Squawker"
"#OMSCS","I really hated CN"

In this example, "#OMSCS" is the most common keyword.

You have been told to efficiently find the most common keywords(s) distributed across the huge dataset over P nodes. There is only one keyword per message, but there are at least 10 million different keywords in use. There may turn out to be more than one "most common value".

(a) (6 points) Using **pseudocode** and referencing any MPI functions you wish, describe an **efficient** solution. The output should be a list of one or more most common keywords.

Solution:
A *key* takeaway (pun intended) is that you don't need to sort and communicate everything – just the keywords with counts. That takes the problem from enormous to just being a distributed sort of 10M elements – not an insurmountable problem.
A most-efficient solution will collect keyword counts locally, then do bucketsort or samplesort (like Lab3) to distribute keywords with their counts across the P nodes (accumulating the counts).
Then local "scan" within each node to find the most common value(s) This can be a MPI_Gather of each node's most common value(s), which can be scanned on the root.
An alternative solution is to do a tree-like accumulation of keyword/accounts, recognizing that there may be different numbers of keywords from each node. It's not MPI_Reduce!
Non-workable: accumulating all keywords on root (P * #keywords is too much data).

(b) (4 points) Assuming a fully-connected network, analyze the communications complexity of your approach in terms of α and β.

Solution:
Communications is dominated by the sort, but for size k data, [number #/size of keywords] (k is much smaller than n, and it is important to make a distinction). β term does not depend on n (the overall problem size). There are a couple variations on how this can be done, but a good answer is $\alpha * P + \beta(P * k)$

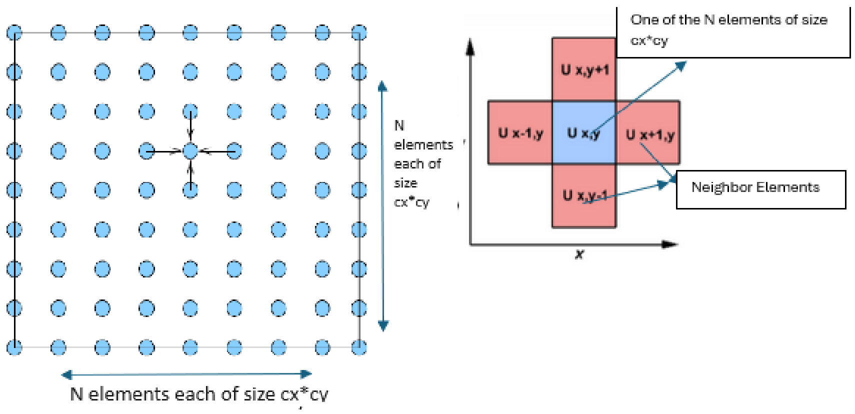

N elements each of size cx*cy
N elements each of size cx*cy
One of the N elements of size cx*cy
U x,y+1
U x-1,y
U x,y
U x+1,y
U x,y-1
Neighbor Elements
x

Figure 2: Heat Diffusion Calculation Setup - $U(x, y)$ represents temperature of element x, y

Question 18 (10 points)

Analyzing a distributed memory problem. Let us consider the problem of simulating heat diffusion on a 2-D surface of N elements in each dimension. Lets say cx (constant) is the size of each element in x-dimension and cy (constant) is the size of the element in y-dimension. The heat equation calculates the change in temperature over a period of time. The initial temperature and boundaries are given. Lets assume that $t1$ & $t2$ are two consecutive time steps having temperatures $U1$ & $U2$. The resulting heat equation is given as:

$$U2_{x,y} = U1_{x,y} + cx * (U1_{x+1,y} + U1_{x-1,y} - 2 * U1_{x,y}) + cy * (U1_{x,y+1} + U1_{x,y-1} - 2 * U1_{x,y})$$

The calculation of temperature of an element is dependent upon the neighbor element temperature values at the prior time step.

Let us consider analyzing a distributed memory algorithm to solve this problem.

(a) (2 points) Lets start with partitioning the data (elements of the 2-D surface) across the processors of the distributed memory setup. Assuming that the size of the element is 1 (cx, cy are both 1) and there are N elements in each dimension (x & y) and there are P processors to do the heat diffusion simulation, can you **provide the number of elements in each processor** in a simple 1-D and 2-D partitioning of data?

Solution:
1-D: P elements of size $\frac{N}{P} * N$
2-D: P elements of size $\frac{N}{\sqrt{P}} * \frac{N}{\sqrt{P}}$

(b) (5 points) Assuming the same setup as **part (a)**, Can you come up with an **estimate of the computation and communication times for one time step** of temperature calculation of all $N * N$ elements for both the partitioning schemes? Explain the size of the communication block that you are using for each partitioning scheme.
Hint: You need to consider the number of elements that have to communicated per process to other processes given the partitioning scheme and the heat diffusion equation

Solution:
Let's ignore the boundary elements because they won't communicate with all neighbors and hence won't dominate the communication times. Say, a & b are the startup time and reverse bandwidth respectively, For the 1-D partitioning, we will send N element blocks to the neighbor (either side). So, the comm time will be 2.(a + b.N). For the 2-D partitioning, $\frac{N}{\sqrt{P}}$ block needs to be communicated to each of 4 blocks. So the total comm time will be 4.(a+b.$\frac{N}{\sqrt{P}}$). The computation time will be the same for both the partitioning scheme which is basicaly the number of elements per process assuming constant times for division and multiplications: $\frac{N^2}{P}$

(c) (3 points) What will be the **isoefficiency** of the 1-D partitioning scheme and what can you conclude from it?
Hint: In the serial solution of the heat diffusion equation stated above, we will just have two outer loops - one across x axis and one across y - axis

Solution:
Isoefficiency is 1D Speedup/P. n = $\Omega(p)$. So n has to grow as much as p for the algorithm to remain scalable.