HPC | HW3 | Alireza Rafiei

1. Network properties (10pt)

1.1 Consider a k-ary d-cube without wrap-around. Show its degree, cost, diameter, bisection bandwidth, arc connectivity in general, and for k=16 and d = 5.

- **Degree**: Each node in a k-ary d-cube is connected to $2\times d$ neighbors. This is because there are d dimensions, and for each dimension, a node has two neighbors: one in the positive direction and one in the negative direction.
- **Cost**: The cost is the total number of edges in the network. With the assumption of $p = k^d$, the cost would be $d^2\times k^d/2$.
- **Diameter**: The diameter of a k-ary d-cube is $d\times(k-1)$. This is the maximum distance between any two nodes in the network. In each dimension, the maximum distance between two nodes is $k-1$, and since there are d dimensions, the diameter becomes $d\times(k-1)$.
- **Bisection Bandwidth**: This is the number of edges that need to be removed to partition the network into two equal halves. In a k-ary d-cube, this would be $k^d/2$.
- **Arc Connectivity**: Arc connectivity is the minimum number of links that must be removed to disconnect the network. For a k-ary d-cube, the arc connectivity is $2\times d$, which is also the degree of each node.

Now, if we substitute $k=16$ and $d=5$:
- **Degree**: $2\times5=10$
- **Cost**: $5^2\times16^5/2 = 13{,}107{,}200$
- **Diameter**: $5\times(16-1) = 75$
- **Bisection Bandwidth**: $16^5/2=524{,}288$
- **Arc Connectivity**: $2\times5=10$

1.2 Show that a hypercube is a k-ary d-cube without wrap-around and its degree, cost, diameter, bisection bandwidth, arc connectivity. Justify that wrap-around is not needed.

A hypercube is indeed a special case of a k-ary d-cube where $k=2$. In this context, "without wrap-around" means that the network does not form loops along each dimension, which is naturally the case for a standard hypercube topology. In a hypercube, nodes are connected in a hierarchical, recursive fashion, and there are no cyclic paths that wrap around the topology.

- **Degree**: A d-dimensional hypercube has a degree of d, which coincides with the general formula for a k-ary d-cube where $k=2$ for considering one directional connection for each node.
- **Cost**: In a d-dimensional hypercube, there are $2d$ nodes. Each node has d edges emanating from it, resulting in $d\times2^d$ edges. However, since each edge is shared between two nodes, the cost (total number of unique edges) becomes $(d^2\times2^d)/2$.
- **Diameter**: The diameter of a d-dimensional hypercube is d, as the farthest node is d hops away. This also aligns with the general $d\times(k-1)$ formula for a k-ary d-cube when $k=2$, which means d.

- **Bisection Bandwidth**: In a d-dimensional hypercube, 2^{d-1} edges need to be cut to bisect the network into two equal sub-networks. This also aligns with the general formula for a k-ary d-cube when k=2, which means 2^{d-1}.
- **Arc Connectivity**: The minimum number of edges that need to be removed to disconnect the network is d. This is consistent with the general property of a k-ary d-cube for considering one directional connection for each node.

Wrap-around is generally useful for reducing the diameter of a network and thus improving latency. However, in a hypercube, the diameter is already quite small (equal to d) relative to the number of nodes. Adding wrap-around would create cycles but wouldn't necessarily improve the existing efficient properties of the hypercube, such as its relatively low diameter and high bisection bandwidth. Furthermore, the hypercube is already a highly symmetric and interconnected topology. Each node has the same properties, and the network is robust in terms of connectivity. Wrap-around would not enhance these properties. Therefor, adding wrap-around could complicate routing algorithms and might not add significant benefits to fault tolerance or other network properties.

2. (10 pt) Design an algorithm for all-gather for the 2-D mesh topology, and write the pseudocode for the algorithm. Analyze its complexity (in terms of latency Ts, bandwidth Tw, total processor count p, and message size m.) Assume each node has 1 message of size m.

In a 2-D mesh topology, nodes are arranged in a grid with rows and columns. I assume that the size of the 2-D mesh is n1×n2 and it is not squared. An all-gather operation requires every node to collect data from every other node in the mesh.

One way to achieve this in a 2-D mesh topology is to use a two-step process:

1. Phase 1: Row-wise gather and broadcast
    - Each row of nodes will perform a local gather and then a broadcast operation. This means each node gathers data from all other nodes in the same row and then broadcasts the gathered data back to all nodes in the same row.
2. Phase 2: Column-wise gather and broadcast
    - After the row-wise operations are complete, each node will perform a similar operation but now along the columns. Each node will gather data from all other nodes in the same column and then broadcast the gathered data back to all nodes in the same column.

Here's the pseudocode to illustrate this algorithm:

Algorithm AllGather2DMesh:

```
Input: my_data of size m at each node and size of mesh (my_data, my_row, my_col, n1, n2)
Output: all_data, a collection of data from all nodes

gathered_row_data = empty_list_of_size(n1 * m)
gathered_full_data = empty_list_of_size(n1 * n2 * m)

// Phase 1: Row-wise gather and broadcast
for i from 0 to n2-1:
    if i == my_col:
        send my_data to all nodes in the same row (except myself)
        gathered_row_data[(i * m) : ((i+1) * m)] = my_data
    else:
        received_data = receive data from node at (my_row, i)
        gathered_row_data[(i * m) : ((i+1) * m)] = received_data

// Broadcast the gathered_row_data to all nodes in the same row
send gathered_row_data to all nodes in the same row (including myself)

// Phase 2: Column-wise gather and broadcast
for j from 0 to n1-1:
    if j == my_row:
        send gathered_row_data to all nodes in the same column (except myself)
        all_data [(j * n2 * m) : ((j+1) * n2 * m)] = gathered_row_data
    else:
        received_data = receive data from node at (j, my_col)
        all_data [(j * n2 * m) : ((j+1) * n2 * m)] = received_data

// Broadcast the gathered_full_data to all nodes in the same column
send all_data to all nodes in the same column (including myself)

return all_data
```

Complexity Analysis

- **Latency (*Ts*):** Each node sends and receives data from $n1-1$ other nodes in the same column and $n2-1$ other nodes in the same row. So, the latency would be $(n1-1)Ts+(n2-1)Ts=(n1+n2-2)\times Ts$.
- **Bandwidth (*Tw*):** In Phase 1, each node sends and receives m bytes to/from $n2-1$ other nodes, and in Phase 2, it sends and receives $n2\times m$ bytes to/from $n1-1$ other nodes. So, the bandwidth term would be $(n2-1)\times m\times Tw+(n1-1)n2\times m\times Tw$.
- **Total Processor Count (p):** In the 2-D mesh, p = n1 × n2.
- **Message Size (m):** Each node starts with 1 message of size “m”.

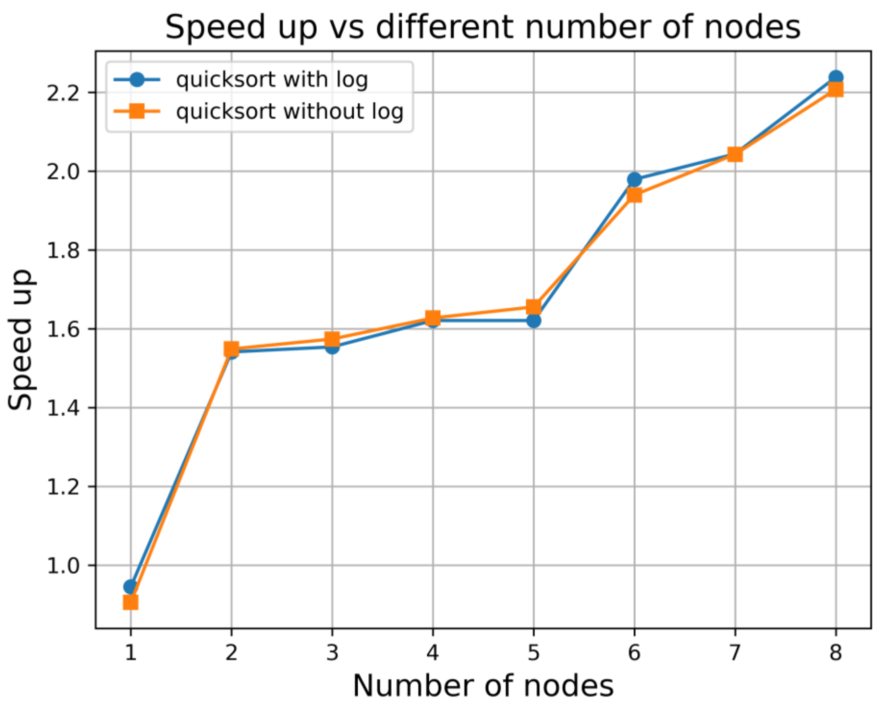
The overall complexity in terms of latency and bandwidth can be considered as additive components of row-wise and column-wise operations: $(n1+n2-2)\times Ts + [(n2-1)\times m\times Tw + (n1-1)\times n2\times m\times Tw]$

3. MPI Quicksort (10pts, 5 pts for write up)

Implement the quicksort algorithm described in class as an MPI program. Do not use SIMD or OpenMP. Evaluate its strong scaling speedup and parallel efficiency for an array of 1000000 randomly generated double precision floating point numbers, for nodes from 1 to 8. To increase the computational intensity, re-run the evaluation by comparing log(x_i + 1.0), where x_i is the array element value at position i.

The mpi implementation for the quicksort algorithm without and with adding a log is available in mpi_quicksort_no_log.c and mpi_quicksort_log.c script respectively (seq_quicksort_no_log.c and seq_quicksort_log.c are the sequential scripts). The following is the strang scaling speed up and parallel efficiency for an array of 1000000 randomly generated numbers for nodes from 1 to 8 in a table and figure format:

p	Scaler	Scaler_log	Runtime	Runtime_log	Speedup	Speedup_log	Parallel efficiency	Parallel efficiency_log
1	0.188	0.192	0.199	0.212	0.944724	0.90566	0.944724	0.90566
2	0.188	0.192	0.122	0.124	1.540984	1.548387	0.770492	0.7741935
3	0.188	0.192	0.121	0.122	1.553719	1.57377	0.5179	0.52459
4	0.188	0.192	0.116	0.118	1.62069	1.627119	0.40517	0.40678
5	0.188	0.192	0.116	0.116	1.62069	1.655172	0.324138	0.33103
6	0.188	0.192	0.095	0.099	1.978947	1.939394	0.3298	0.32323
7	0.188	0.192	0.092	0.094	2.043478	2.042553	0.2919	0.29179
8	0.188	0.192	0.084	0.087	2.238095	2.206897	0.2798	0.2759



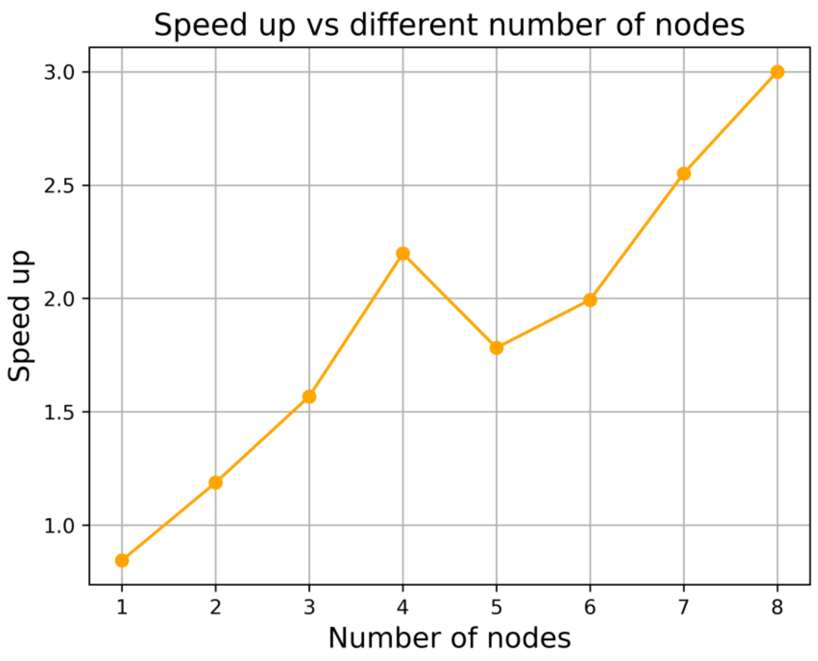
4. MPI Simple matrix-vector multiplication (C[N] = A[N][N] x B[N]) (10points, 5 points for write-up)

Starting with the simple matrix multiplication code from HW 0, re-implement using MPI. Do not use SIMD or OpenMP. Evaluate its strong scaling speedup and parallel efficiency for sizes of A = 4000x4000 and B = 4000x1, both as randomly generated double precision floating point numbers, for nodes from 1 to 8. Evaluate weak scaling performance sizes of A = (p * 1000) x 4000 and B = 4000x1. Present a table as well as a plot for the scaling results. Describe your algorithm in the write up, including partitioning and data movement/communication strategy.

The mpi implementation for the matrix-vector multiplication algorithm for strong and weak scaling is available in mpi_mat_multiply_st.c and mpi_mat_multiply_we.c script respectively (seq_mat_multiply_we.c and seq_mat_multiply_st.c are the sequential scripts). The following are the table and figure for nodes from 1 to 8:

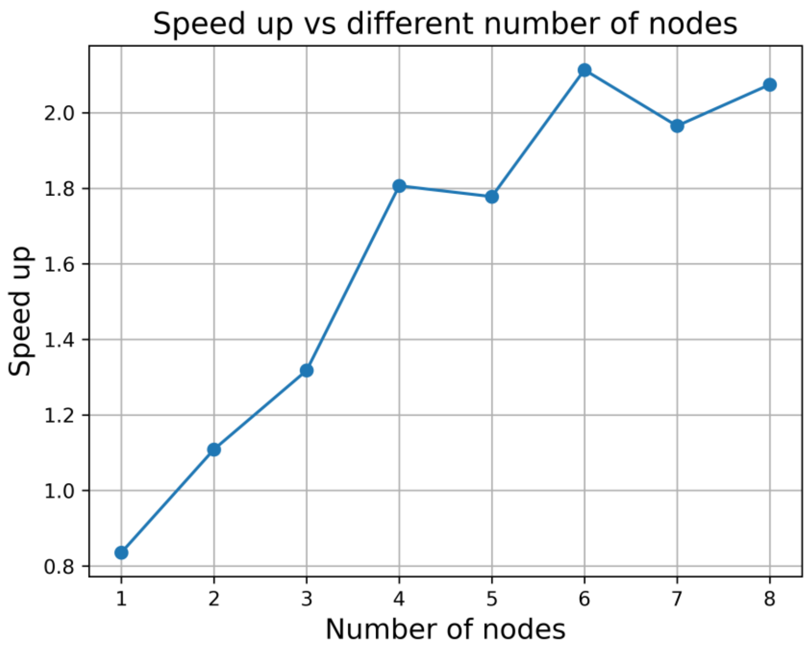
Week scaling:

p	Scaler	Runtime	Speedup
1	0.109	0.129	0.844961
2	0.228	0.195	1.1875
3	0.312	0.085	1.567839
4	0.455	0.062	2.198068
5	0.581	0.063	1.782209
6	0.652	0.053	1.993884
7	0.796	0.057	2.551282
8	0.903	0.054	3



Strong scaling:

p	Scaler	Runtime	Speedup
1	0.112	0.134	0.835821
2	0.112	0.101	1.108911
3	0.112	0.085	1.317647
4	0.112	0.062	1.806452
5	0.112	0.063	1.777778
6	0.112	0.053	2.113208
7	0.112	0.057	1.964912
8	0.112	0.054	2.074074



The algorithm for matrix-vector multiplication (C[N] = A[N][N] x B[N]) using MPI relies on a data parallelism approach, wherein each process is responsible for computing a subset of the elements of the resulting vector.

Algorithm Description:

1. **Initialization**: Each process initializes its portion of matrix A and vector B, which can effectively increase the data size with the number of processes.

2. **Data Partitioning**:

   The rows of matrix A are partitioned evenly among all processes. If there are P processes and N rows, then each process gets N/P rows. Also, each process deals with p * 1000 rows where p is the number of processes (or one in the strong scaling part). Hence, as p increases, the number of rows each process is responsible for also increases linearly.

3. **Data Movement / Communication**:

Broadcast: The vector B is broadcast to all processes. This is because every process needs the complete vector B for its calculations.

Scatter / Gather: The root process scatters the rows of A and gathers the results of C.

4. **Timing Metrics**: The root process records the time before and after the computational phase to measure the execution time.

Communication Strategy:

Broadcast: The MPI_Bcast function is used to send the vector B to all processes. This is a one-to-all communication.

Scatter: The MPI_Scatter function is used to distribute the rows of A among all the processes. This is a one-to-all communication.

Gather: The MPI_Gather function is used to collect the calculated portions of C from all processes back to the root process. This is an all-to-one communication.

By employing this partitioning and data movement strategy, the algorithm aims to achieve optimal utilization of all available processes, thereby improving the speedup and efficiency.

5. **PageRank: (10points, 5 points for write-up)**

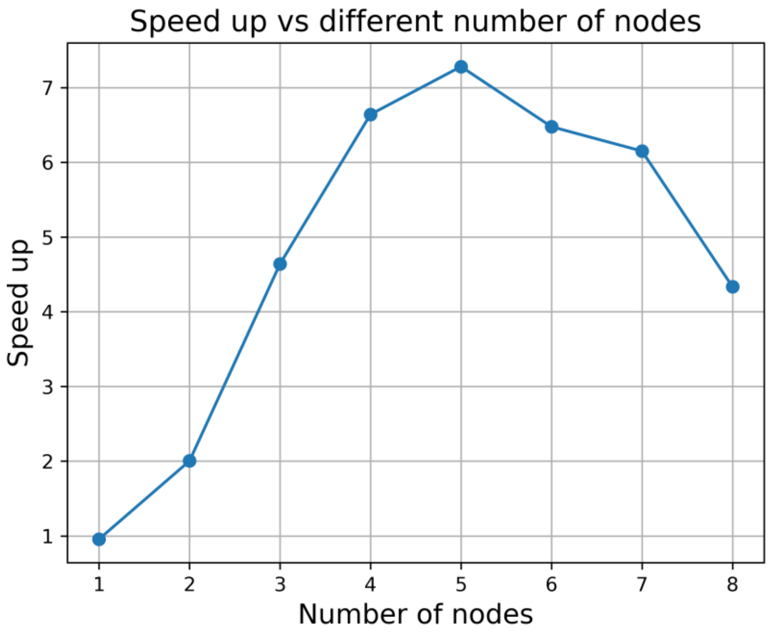
Matrix-vector multiply is a core operation for PageRank computation using the iterative method, and is invoked iteratively until convergence. https://en.wikipedia.org/wiki/PageRank (see the iterative method section under "implementation"). Here the matrix A (M on the Wikipedia page) is fixed, and the page range vector B (R on the Wikipedia page) is updated repeatedly, so efficient communication of B is important. Extend your implementation from 4, implement the core component of PageRank as:

B(t+1) = A*B(t)

Evaluate the strong scaling performance of your algorithm for A = 4000x4000, and B = 4000x1, for 100 iterations, for 1 to 8 nodes, and present a table as well as a plot for the strong scaling results.

The mpi PageRank code is available in mpi_pagerank.c and the sequential version in seq_pageramk.c. The following table and figure represent the strong scaling result for 100 iterations and 1 to 8 nodes.

p	Scaler	Runtime	Speedup
1	5.744	5.999	0.957493
2	5.744	2.866	2.004187
3	5.744	1.238	4.639742
4	5.744	0.865	6.640462
5	5.744	0.789	7.280101
6	5.744	0.887	6.475761
7	5.744	0.934	6.149893
8	5.744	1.325	4.335094


Speed up vs different number of nodes
Speed up
Number of nodes
1
2
3
4
5
6
7
8
1
2
3
4
5
6
7