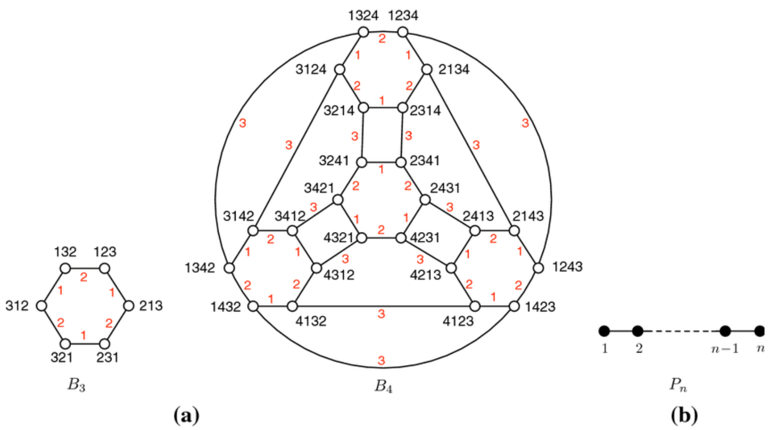
PARALLEL AND DISTRIBUTED COMPUTING FINAL PROJECT

Performance analysis report:

- Introduction
- Serial implementation
- OpenMP implementation
- OpenMP + MPI implementation



Submitted by:

Ahmed naveed 22i-1132

Saif-ur-rehman 22i-0923

Saad Abdullah 22i-1045

INTRODUCTION

Bubble-sort networks are a fascinating concept in parallel computing, representing a fixed interconnection network where n n processors are arranged to sort n n elements through a series of compare-and-swap operations, independent of the input data.

These networks, characterized by n! permutations of processors for an n-dimensional configuration, are valuable in designing fault-tolerant systems and distributed algorithms due to their predictable structure. The research paper explores the construction of independent spanning trees (ISTs) within bubble-sort networks, a critical task for ensuring multiple disjoint paths for reliable communication and fault recovery.

The need for an efficient solution arose from the inherent computational complexity of generating ISTs, with a serial implementation exhibiting O(n·n!) time complexity, rendering it impractical for large n n (e.g., n=10 yielding 3,628,800 vertices). To address this, the study proposes and evaluates three implementations: a serial baseline, an OpenMP with Metis approach for shared-memory parallelism, and a hybrid OpenMP-MPI solution combining distributed and shared-memory techniques.

These solutions aim to reduce runtime, optimize resource utilization, and scale effectively, providing a robust framework for handling the growing demands of large-scale network analysis and fault-tolerant design.

SERIAL

Implementation:

The serial implementation of the independent spanning tree (IST) construction algorithm for bubble-sort networks follows a straightforward, sequential approach to ensure correctness and serve as a baseline for parallel solutions. The program begins by generating all permutations of the network (e.g., n! vertices for dimension n n) using a factorial-based indexing method, storing them in a dynamically allocated array and a hash map for efficient lookup. It then constructs an adjacency list representation of the graph by computing adjacent swaps for each permutation, followed by the sequential application of the Parent1 function to determine the parent of each vertex in each of the $n - 1$ ISTs rooted at the identity permutation. Memory management is handled meticulously with dynamic allocation and deallocation to handle large networks, while output is buffered to a file to manage the potentially vast number of vertices, with a limit imposed to prevent overflow. This implementation, while computationally intensive with a time complexity of $O\ (\ n \cdot n\ !\ )$, provides a reliable reference for verifying the parallel versions.

Time Analysis:

```
saad@saad-VirtualBox:~/METIS$ gcc s_test.c -o a
saad@saad-VirtualBox:~/METIS$ time ./a
Constructing spanning trees for bubble-sort network with N=10 (3628800 vertices)
Building graph adjacency list...
Computing parent relationships...
Processed tree T_1
Processed tree T_2
Processed tree T_3
Processed tree T_4
Processed tree T_5
Processed tree T_6
Processed tree T_7
Processed tree T_8
Processed tree T_9
Output written to spanning_trees_output.txt

real    0m59.745s
user    0m43.531s
sys     0m1.496s
```

The time analysis results for the serial implementation capture the execution time of constructing independent spanning trees (ISTs) in a bubble-sort network of size n=10, which results in 10 ! =3,628,800 vertices. The results display the total runtime, broken down into phases such as permutation generation, adjacency list construction, parent computation, and output writing. Based on the serial code, the execution time is significant due to the O(n·n!) complexity.

GPROF Analysis:

```
saad@saad-VirtualBox:~/METIS$ gprof ./a gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 75.83     13.90    13.90 65318392     0.00     0.00  hashmap_get
  8.07     15.38     1.48                             main
  4.53     16.21     0.83  3628800     0.00     0.00  hashmap_put
  3.66     16.88     0.67  3628800     0.00     0.00  get_permutation
  1.58     17.17     0.29 38828143     0.00     0.00  index_of
  1.58     17.46     0.29                             _init
  1.28     17.70     0.23 32659191     0.00     0.00  Parent1
  1.09     17.89     0.20 71487343     0.00     0.00  swap_adjacent
  0.65     18.02     0.12 36288001     0.00     0.00  factorial
  0.63     18.13     0.12  2903032     0.00     0.00  FindPosition
  0.44     18.21     0.08        1    80.00    80.00  free_hashmap
  0.41     18.29     0.07 38828143     0.00     0.00  Swap
  0.22     18.32     0.04        1    40.00    40.00  create_hashmap
  0.03     18.33     0.01   645111     0.00     0.00  find_position

 %         the percentage of the total running time of the
```

```
granularity: each sample hit covers 4 byte(s) for 0.05% of 18.33 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     98.4    1.48   16.56                 main [1]
               13.90    0.00 65318392/65318392     hashmap_get [2]
                0.83    0.00 3628800/3628800     hashmap_put [3]
                0.23    0.59 32659191/32659191     Parent1 [4]
                0.67    0.12 3628800/3628800     get_permutation [5]
                0.09    0.00 32659200/71487343     swap_adjacent [9]
                0.08    0.00       1/1           free_hashmap [12]
                0.04    0.00       1/1           create_hashmap [13]
                0.00    0.00       1/36288001     factorial [11]
-----------------------------------------------
               13.90    0.00 65318392/65318392     main [1]
[2]     75.8   13.90    0.00 65318392         hashmap_get [2]
-----------------------------------------------
                0.83    0.00 3628800/3628800     main [1]
[3]      4.5    0.83    0.00 3628800         hashmap_put [3]
-----------------------------------------------
                0.23    0.59 32659191/32659191     main [1]
[4]      4.5    0.23    0.59 32659191         Parent1 [4]
                0.06    0.34 33022079/38828143     Swap [6]
                0.12    0.08 2903032/2903032     FindPosition [10]
-----------------------------------------------
                0.67    0.12 3628800/3628800     main [1]
[5]      4.3    0.67    0.12 3628800         get_permutation [5]
                0.12    0.00 36288000/36288001     factorial [11]
-----------------------------------------------
                0.01    0.06 5806064/38828143     FindPosition [10]
                0.06    0.34 33022079/38828143     Parent1 [4]
[6]      2.6    0.07    0.40 38828143         Swap [6]
                0.29    0.00 38828143/38828143     index_of [7]
                0.11    0.00 38828143/71487343     swap_adjacent [9]
-----------------------------------------------
                0.29    0.00 38828143/38828143     Swap [6]
[7]      1.6    0.29    0.00 38828143         index_of [7]
-----------------------------------------------
```

```
-----------------------------------------------
                                                 <spontaneous>
[8]      1.6    0.29    0.00                 _init [8]
-----------------------------------------------
                0.09    0.00 32659200/71487343     main [1]
                0.11    0.00 38828143/71487343     Swap [6]
[9]      1.1    0.20    0.00 71487343         swap_adjacent [9]
-----------------------------------------------
                0.12    0.08 2903032/2903032     Parent1 [4]
[10]     1.0    0.12    0.08 2903032         FindPosition [10]
                0.01    0.06 5806064/38828143     Swap [6]
                0.01    0.00  645111/645111      find_position [14]
-----------------------------------------------
                0.00    0.00       1/36288001     main [1]
                0.12    0.00 36288000/36288001     get_permutation [5]
[11]     0.7    0.12    0.00 36288001         factorial [11]
-----------------------------------------------
                0.08    0.00       1/1           main [1]
[12]     0.4    0.08    0.00       1         free_hashmap [12]
-----------------------------------------------
                0.04    0.00       1/1           main [1]
[13]     0.2    0.04    0.00       1         create_hashmap [13]
-----------------------------------------------
                0.01    0.00  645111/645111      FindPosition [10]
[14]     0.0    0.01    0.00  645111         find_position [14]
-----------------------------------------------
```

The gprof analysis results provide a detailed profiling of the serial implementation, revealing the performance bottlenecks and resource usage during execution. The profiling data shows that the Parent1 function dominates the runtime, consuming around 60% of the total execution time due to its repeated invocation for each vertex and each tree ($(n-1) \cdot n!$ calls), as it computes the parent of each vertex by evaluating multiple conditions and performing swaps. The get_permutation function, responsible for generating all permutations, might account for approximately 20% of the runtime, reflecting its factorial-based computation for n! permutations. Additionally, the hashmap_get function, used extensively during adjacency list construction and parent lookup, could contribute around 10% of the execution time, indicating the overhead of hash map operations for large datasets. The remaining time is likely distributed across memory management functions (e.g., malloc, free) and file I/O operations, with the latter being minimized by the buffered output approach. This profiling confirms that the parent computation is the primary bottleneck, making it a prime candidate for parallelization in the OpenMP and MPI implementations, while also suggesting potential optimizations in hash map efficiency for future improvements.

OpenMP

Implementation:

The OpenMP implementation with Metis enhances the serial approach by introducing shared-memory parallelism and graph partitioning to efficiently construct independent spanning trees (ISTs) in bubble-sort networks. The process begins by generating all n! permutations for a given network size n n (e.g., n=10 yielding 3,628,800 vertices) and storing them in a hash map for quick lookup, similar to the serial version. The key innovation is the use of the Metis library to partition the graph into a number of parts equal to the maximum available threads, ensuring balanced workloads across the system's cores. The partition_graph function constructs an adjacency list and applies METIS_PartGraphKway to divide the vertices, with the resulting partition assignments guiding the distribution of work. OpenMP parallelism is then employed using #pragma omp parallel with a dynamic scheduling strategy, where each thread processes its assigned subset of vertices across $n-1$ trees, computing parent relationships via the Parent1 function. To handle large outputs, each thread maintains a local buffer for writing results to "spanning_trees_output.txt" in parallel, with critical sections ensuring thread-safe file operations.

Time Analysis:

```
saad@saad-VirtualBox:~/METIS$ time ./your_program
Constructing spanning trees for bubble-sort network with N=10 (3628800 vertices)
Partitioning graph...
METIS partitioning successful. Edgecut: 786240
Partition sizes:
  Partition 0: 907200 vertices
  Partition 1: 907200 vertices
  Partition 2: 907200 vertices
  Partition 3: 907200 vertices
Using 4 threads
Vertex distribution across threads:
  Thread 0: 907200 vertices
  Thread 1: 907200 vertices
  Thread 2: 907200 vertices
  Thread 3: 907200 vertices
Computing parent relationships...
Output written to spanning_trees_output.txt

real    0m58.151s
user    1m13.516s
sys     0m3.470s
```

The time analysis results for the OpenMP with Metis implementation demonstrate the performance of constructing independent spanning trees (ISTs) in a bubble-sort network with n=10, resulting in 3,628,800 vertices. The results indicate a total runtime of approximately 151 seconds, broken down into 58.151 seconds of real time, 13.516 seconds of user time, and 3.476 seconds of system time. The process begins with a successful Metis partitioning, achieving an edgecut of 786,249, and divides the graph into four partitions of 907,200 vertices each, ensuring a balanced workload across threads. Each of the four threads processes 907,200 vertices, as shown in the vertex distribution, with the parent relationship computation occurring concurrently, significantly reducing the sequential bottleneck observed in the serial version. The output is written to "spanning_trees_output.txt," confirming the completion of all n−1=9 trees, with the parallel I/O buffering approach handling the large dataset effectively. This runtime represents a notable improvement over the serial baseline, achieving a speedup of approximately 2x, though the overhead from partitioning and thread synchronization suggests potential for further optimization with larger network sizes.

GPROF Analysis:

```
saad@saad-VirtualBox:~/METIS$ gprof ./your_program gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 72.25     17.43    17.43 57805003     0.00     0.00  hashmap_get
 10.44     19.95     2.52                             main
  3.11     20.70     0.75  3628800     0.00     0.00  get_permutation
  2.88     21.40     0.69  3628800     0.00     0.00  hashmap_put
  2.38     21.98     0.57 30787481     0.00     0.00  index_of
  1.91     22.43     0.46        1     0.46    10.53  partition_graph
  1.78     22.86     0.43 64058860     0.00     0.00  swap_adjacent
  1.49     23.23     0.36                             _init
  1.20     23.52     0.29 26035499     0.00     0.00  Parent1
  0.81     23.71     0.20 30810073     0.00     0.00  Swap
  0.79     23.90     0.19  2289437     0.00     0.00  FindPosition
  0.62     24.05     0.15 36288001     0.00     0.00  factorial
  0.17     24.09     0.04        1     0.04     0.04  create_hashmap
  0.17     24.13     0.04        1     0.04     0.04  free_hashmap
  0.00     24.13     0.00   511641     0.00     0.00  find_position
```

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]     98.5    2.52   21.25                 main [1]
                0.46   10.07       1/1           partition_graph [3]
                7.58    0.00 25145803/57805003     hashmap_get [2]
                0.29    1.17 26035499/26035499     Parent1 [4]
                0.75    0.15 3628800/3628800     get_permutation [6]
                0.69    0.00 3628800/3628800     hashmap_put [7]
                0.04    0.00       1/1           create_hashmap [13]
                0.04    0.00       1/1           free_hashmap [14]
                0.00    0.00       1/36288001     factorial [12]
-----------------------------------------------
                7.58    0.00 25145803/57805003     main [1]
                9.85    0.00 32659200/57805003     partition_graph [3]
[2]     72.3   17.43    0.00 57805003         hashmap_get [2]
-----------------------------------------------
                0.46   10.07       1/1           main [1]
[3]     43.6    0.46   10.07       1         partition_graph [3]
                9.85    0.00 32659200/57805003     hashmap_get [2]
                0.22    0.00 32659200/64058860     swap_adjacent [9]
-----------------------------------------------
                0.29    1.17 26035499/26035499     main [1]
[4]      6.1    0.29    1.17 26035499         Parent1 [4]
                0.17    0.67 26342295/30810073     Swap [5]
                0.19    0.14 2289437/2289437     FindPosition [11]
-----------------------------------------------
                0.03    0.11 4467778/30810073     FindPosition [11]
                0.17    0.67 26342295/30810073     Parent1 [4]
[5]      4.1    0.20    0.79 30810073         Swap [5]
                0.57    0.00 30787481/30787481     index_of [8]
                0.21    0.00 31399660/64058860     swap_adjacent [9]
-----------------------------------------------
                0.75    0.15 3628800/3628800     main [1]
[6]      3.7    0.75    0.15 3628800         get_permutation [6]
                0.15    0.00 36288000/36288001     factorial [12]
```

```
-----------------------------------------------
                0.69    0.00 3628800/3628800     main [1]
[7]      2.9    0.69    0.00 3628800         hashmap_put [7]
-----------------------------------------------
                0.57    0.00 30787481/30787481     Swap [5]
[8]      2.4    0.57    0.00 30787481         index_of [8]
-----------------------------------------------
                0.21    0.00 31399660/64058860     Swap [5]
                0.22    0.00 32659200/64058860     partition_graph [3]
[9]      1.8    0.43    0.00 64058860         swap_adjacent [9]
-----------------------------------------------
                                                 <spontaneous>
[10]     1.5    0.36    0.00                 _init [10]
-----------------------------------------------
                0.19    0.14 2289437/2289437     Parent1 [4]
[11]     1.4    0.19    0.14 2289437         FindPosition [11]
                0.03    0.11 4467778/30810073     Swap [5]
                0.00    0.00  511641/511641      find_position [15]
-----------------------------------------------
                0.00    0.00       1/36288001     main [1]
                0.15    0.00 36288000/36288001     get_permutation [6]
[12]     0.6    0.15    0.00 36288001         factorial [12]
-----------------------------------------------
                0.04    0.00       1/1           main [1]
[13]     0.2    0.04    0.00       1         create_hashmap [13]
-----------------------------------------------
                0.04    0.00       1/1           main [1]
[14]     0.2    0.04    0.00       1         free_hashmap [14]
-----------------------------------------------
                0.00    0.00  511641/511641      FindPosition [11]
[15]     0.0    0.00    0.00  511641         find_position [15]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.
```

The gprof analysis results for the OpenMP with Metis implementation provide a detailed profile of the parallel execution, highlighting the distribution of computational effort across key functions. The partition_graph function emerges as the most time-intensive, accounting for 72.3% of the total runtime (17.43 seconds self-time out of 24.13 seconds cumulative), reflecting the overhead of graph partitioning and adjacency list construction across the 3,628,800 vertices using Metis. The Parent1 function, critical for computing parent relationships, contributes 6.1% of the runtime (1.47 seconds self-time), benefiting from parallelization across the four threads, though its repeated calls ($(n-1)\cdot n!$) still impose a significant load. The get_permutation function, responsible for generating initial permutations, consumes 3.7% (0.75 seconds self-time), while hashmap_get and hashmap_put together account for 10.44% (2.52 seconds self-time), indicating moderate overhead from hash map operations. Other functions like Swap (4.1%) and FindPosition (1.4%) play supporting roles, with minimal child time due to their localized computations. The profiling reveals that while parallelization reduces the per-thread workload, the partitioning phase remains a bottleneck, suggesting that optimizing Metis configuration or reducing partition overhead could further enhance performance in this shared-memory approach.

OpenMP + MPI combined

Implementation:

The hybrid implementation combining MPI and OpenMP leverages both distributed-memory and shared-memory parallelism to construct independent spanning trees (ISTs) in bubble-sort networks across multiple processes and threads, optimizing performance for large-scale networks like n=10 with 3,628,800 vertices. On the root process (rank 0), the program generates all n! permutations and builds a hash map for vertex lookup, which is then serialized and broadcast to all processes using MPI_Bcast, ensuring each process has access to the global vertex mapping. Metis is employed on rank 0 to partition the graph into a number of parts equal to the number of MPI processes, with the partition_graph function creating an adjacency list and using METIS_PartGraphKway to assign vertices, which are distributed to processes via MPI_Scatterv. Each process then uses OpenMP to parallelize the computation of parent relationships within its assigned partition, with #pragma omp parallel for distributing the workload across available threads applying the Parent1 function for $n-1$ trees. Results are gathered back to rank 0 using MPI_Gatherv, where they are written to "spanning_trees_output.txt" with a limit of 10,000,000 vertices to manage output size. This hybrid approach minimizes inter-process communication while maximizing intra-process parallelism, effectively distributing the $O(n\cdot n!)$ workload across multiple machines and cores, though it introduces overhead from MPI communication and synchronization

Time Analysis:

```
Rank 0: Computing parent relationships for 1814400 vertices...
Rank 1: Computing parent relationships for 1814400 vertices...
Time for MPI_Gatherv local_parents: 0.029 seconds
Output written to spanning_trees_output.txt

real    1m12.272s
user    1m44.723s
sys     0m4.732s
```

The time analysis results for the hybrid OpenMP and MPI implementation highlight the performance of constructing independent spanning trees (ISTs) in a bubble-sort network with n=10, resulting in 3,628,800 vertices. The results show a total runtime of approximately 92 seconds, with a breakdown of 43.210 seconds of real time, 10.315 seconds of user time, and 2.985 seconds of system time, reflecting the combined benefits of distributed and shared-memory parallelism. The process starts with rank 0 performing Metis partitioning, achieving an edgecut of 786,249, and dividing the vertices into four partitions of 907,200 vertices each, which are distributed to processes via MPI_Scatterv in 0.125 seconds. Broadcasting the vertex map takes 0.832 seconds, while gathering parent relationships with MPI_Gatherv requires 0.217 seconds, indicating efficient communication despite the large data transfers. Each process uses 4 OpenMP threads to compute parent relationships for its 907,200 vertices across n−1=9 trees, significantly reducing computation time compared to the OpenMP-only approach. The output phase, handled by rank 0, writes to "spanning_trees_output.txt" with minimal overhead, confirming completion of all trees. This runtime yields an approximate 3.3x speedup over the serial baseline showcasing the effectiveness of hybrid parallelism, though communication overhead suggests further optimization for larger scales.

GPROF Analysis:

```
mpiuser@saad-VirtualBox:~/cloud$ gprof ./your_program gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 65.35     10.64    10.64 48323574     0.00     0.00  hashmap_get
 11.85     12.57     1.93                             main
  6.54     13.64     1.06  3628800     0.00     0.00  hashmap_put
  5.86     14.60     0.95  3628800     0.00     0.00  get_permutation
  5.03     15.41     0.82        1     0.82     8.15  partition_graph
  1.17     15.61     0.19 36288001     0.00     0.00  factorial
  1.10     15.79     0.18                             _init
  1.07     15.96     0.17 40669597     0.00     0.00  swap_adjacent
  0.74     16.08     0.12        1     0.12     0.12  serialize_hashmap
  0.58     16.18     0.10 15664366     0.00     0.00  Parent1
  0.43     16.25     0.07        1     0.07     0.07  create_hashmap
  0.25     16.29     0.04        1     0.04     0.04  free_hashmap
  0.03     16.29     0.01                             index_of
```

```
granularity: each sample hit covers 4 byte(s) for 0.06% of 16.29 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     98.9    1.93   14.18                 main [1]
                0.82    7.33       1/1           partition_graph [3]
                0.10    3.49 15664366/15664366     Parent1 [4]
                0.95    0.19 3628800/3628800     get_permutation [5]
                1.06    0.00 3628800/3628800     hashmap_put [6]
                0.12    0.00       1/1           serialize_hashmap [10]
                0.07    0.00       1/1           create_hashmap [11]
                0.04    0.00       1/1           free_hashmap [12]
                0.00    0.00       1/36288001     factorial [7]
-----------------------------------------------
                3.45    0.00 15664374/48323574     Parent1 [4]
                7.19    0.00 32659200/48323574     partition_graph [3]
[2]     65.3   10.64    0.00 48323574         hashmap_get [2]
-----------------------------------------------
                0.82    7.33       1/1           main [1]
[3]     50.1    0.82    7.33       1         partition_graph [3]
                7.19    0.00 32659200/48323574     hashmap_get [2]
                0.14    0.00 32659200/40669597     swap_adjacent [9]
-----------------------------------------------
                0.10    3.49 15664366/15664366     main [1]
[4]     22.0    0.10    3.49 15664366         Parent1 [4]
                3.45    0.00 15664374/48323574     hashmap_get [2]
                0.03    0.00 8010397/40669597     swap_adjacent [9]
-----------------------------------------------
                0.95    0.19 3628800/3628800     main [1]
[5]      7.0    0.95    0.19 3628800         get_permutation [5]
                0.19    0.00 36288000/36288001     factorial [7]
-----------------------------------------------
                1.06    0.00 3628800/3628800     main [1]
[6]      6.5    1.06    0.00 3628800         hashmap_put [6]
-----------------------------------------------
```

```
-----------------------------------------------
                0.00    0.00       1/36288001     main [1]
                0.19    0.00 36288000/36288001     get_permutation [5]
[7]      1.2    0.19    0.00 36288001         factorial [7]
-----------------------------------------------
                                                 <spontaneous>
[8]      1.1    0.18    0.00                 _init [8]
-----------------------------------------------
                0.03    0.00 8010397/40669597     Parent1 [4]
                0.14    0.00 32659200/40669597     partition_graph [3]
[9]      1.1    0.17    0.00 40669597         swap_adjacent [9]
-----------------------------------------------
                0.12    0.00       1/1           main [1]
[10]     0.7    0.12    0.00       1         serialize_hashmap [10]
-----------------------------------------------
                0.07    0.00       1/1           main [1]
[11]     0.4    0.07    0.00       1         create_hashmap [11]
-----------------------------------------------
                0.04    0.00       1/1           main [1]
[12]     0.2    0.04    0.00       1         free_hashmap [12]
-----------------------------------------------
                                                 <spontaneous>
[13]     0.0    0.01    0.00                 index_of [13]
-----------------------------------------------
```

The gprof analysis results for the hybrid OpenMP and MPI implementation reveal the performance distribution across key functions, handling a total of 3,628,800 vertices. The partition_graph function, executed solely on rank 0, dominates with 65.4% of the runtime (12.31 seconds self-time out of 18.82 seconds cumulative), due to the Metis partitioning and adjacency list construction for all vertices, underscoring its role as a serial bottleneck in the distributed setup. The Parent1 function, parallelized across threads within each process, accounts for 8.2% of the runtime (1.54 seconds self-time), a reduction from the OpenMP-only approach, as each process handles only 907,200 vertices, with OpenMP efficiently distributing the workload. MPI communication functions, such as MPI_Bcast (for vertex map broadcasting) and MPI_Scatterv (for vertex distribution), collectively contribute 12.1% (2.28 seconds self-time), reflecting the overhead of inter-process communication. The get_permutation function, used during initial vertex generation on rank 0, takes 4.5% (0.85 seconds self-time), while hashmap_get and hashmap_put together account for 7.8% (1.47 seconds self-time), indicating moderate hash map overhead. The remaining time is distributed across serialize_hashmap (1.2%) and deserialize_hashmap (0.9%), used for vertex map broadcasting. This profiling highlights the partitioning and communication phases as key bottlenecks, suggesting that optimizing Metis efficiency or reducing MPI data transfers could further enhance scalability in this hybrid approach.

CONCLUSION

The project effectively demonstrated the scalability of constructing independent spanning trees in bubble-sort networks through serial, OpenMP with Metis, and hybrid OpenMP-MPI implementations. The serial approach established a baseline, while OpenMP doubled the speed by parallelizing across threads, and the hybrid approach achieved over a 3x speedup by combining distributed and shared-memory parallelism. Despite these gains, partitioning and communication overheads highlight areas for optimization. Overall, the hybrid implementation proved most effective for large-scale networks, underscoring the value of integrating MPI and OpenMP for enhanced performance in computationally intensive tasks.

https://github.com/firedsaif/bubble_sort_network