IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 29 March 2023, accepted 20 April 2023, date of publication 24 April 2023, date of current version 27 April 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3269793

RESEARCH ARTICLE

Accelerating Parallel Applications Based on Graph Reordering for Random Network Topologies

YAO HU, (Member, IEEE)
Research Institute for Digital Media and Content, Keio University, Hiyoshi Campus, Yokohama, Kanagawa 223-8523, Japan
e-mail: huyao0107@gmail.com

This work was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 21K11859.

ABSTRACT The Message Passing Interface (MPI) is a crucial programming tool for enabling communication between processes in parallel applications. The goal of MPI users is to allocate tasks to processors in a way that maximizes both spatial and temporal locality in the network. However, this can be challenging, especially in large-scale networks where maximizing processor locality may not be feasible at runtime. To address this issue, we propose the use of *Hamorder*, an offline node reassignment approach that takes into account physical processor locations based on graph reordering for *Random* network topologies. *Hamorder* aims to optimize task mapping for improved performance in parallel applications, whether for multiple tasks or within a single task. Additionally, we investigate the potential of improving MPI applications through runtime parameter tuning based on *Hamorder*. Our evaluation results show that *Hamorder* provides a 27.3% improvement in performance compared to the *Gorder* algorithm on *Random* topologies, which is a state-of-the-art solution designed with the aim of enhancing cache locality and achieves this goal by rearranging the vertices of a graph in a way that places the vertices that are typically accessed together in close proximity. Moreover, our autotuning framework using *Hamorder* results in an average speedup of 1.38× for targeted MPI applications by searching through various runtime parameter combinations.

INDEX TERMS Autotuning, graph reordering, message passing interface (MPI), parallel application.

I. INTRODUCTION

In parallel applications that rely heavily on communication, a substantial portion of the execution time is dedicated to exchanging numerical data between processes, which becomes the primary factor affecting message transfer. The Message Passing Interface (MPI) [1] is a standardized library specification designed for communication between processes in a parallel computing environment. MPI enables processes running on separate processors to interact with each other, for example, by exchanging intermediate results or sending data to other processes for additional processing and analysis.

In MPI-based parallel applications, there is a need for rapid transfer of large amounts of numerical data over a network. The placement of MPI processes, also known as ranks, on processors is optimized to align the communication patterns with the underlying hardware architecture. Task mapping plays a crucial role in ensuring efficient inter-processor communication during application execution. However, the task mapping problem is known to be NP-complete [2], meaning that there is no computationally feasible algorithm for evaluating all possible communication patterns, as different mapping methods bring their own challenges, such as long path length and heavy network congestion. Nevertheless, much research has been done on specialized strategies and heuristics [3], [4], [5], [6]. In general, an effective and efficient task mapping is achieved by optimizing the locality of the communication. A mapping with high locality results in communication between processors that are close to each other, while a mapping with low locality results in communication between processors that are dispersed throughout the network. The objective is

The associate editor coordinating the review of this manuscript and approving it for publication was Tomas F. Pena.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

to map tasks in a manner that minimizes inter-processor communication.

Previous studies have introduced node reordering methods that take advantage of graph characteristics, such as vertex neighborhood and vertex degree distribution, to enhance cache block utilization [7], [8], [9], [10]. The basic idea behind these methods is to rearrange vertices in memory so that frequently accessed vertices are located together in a contiguous memory region. This increases the likelihood that memory blocks comprised of frequently accessed vertices will remain in the cache, resulting in higher cache utilization. In this study, we introduce an offline graph reordering algorithm named *Hamorder*. The term "offline" is used to indicate that the *Hamorder* algorithm operates independently of the task execution, resulting in little impact on application performance with regards to its time complexity. Our focus in this research is on *Random* network topologies made possible by wireless optical communication, such as free space optical (FSO) technology [11]. In our earlier studies, we have also developed FSO link designs for the deployment of high-performance computing (HPC) clusters [12] and demonstrated the use of FSO for task scheduling in datacenters [13]. To simulate the number of FSO terminals per node, we assign the network degree in a *Random* topology. *Hamorder* renumbers nodes in *Random* network topologies to improve process embedding locality for parallel application execution. There are two important differences between *Hamorder* and existing graph reordering techniques. Firstly, *Hamorder* aims to optimize parallel application performance instead of increasing cache hit rate. Secondly, *Hamorder* enhances locality for both inter-task and intra-task processor number ordering, whereas existing graph reordering techniques often ignore vertex reordering within a node group or community.

Figure 1 illustrates the impact of the *Hamorder* algorithm on a sample *Random* network topology. The performance of graph reordering algorithms based on BFS (Breadth First Search) or DFS (Depth First Search) may vary depending on the choice of the root node, leading to different level assignments and node ordering performance. To simplify our illustration of the effect of the *Hamorder* algorithm, we use a random node ordering as a baseline. We consider two parallel tasks, one with 4 ranks and the other with 5 ranks. We use a default *by-order* task mapping approach [3], which maps ranks to available processors in ascending order, i.e., rank-n is deployed on processor-n, for example, rank-0 on processor-0, rank-1 on processor-1, rank-2 on processor-2, etc. After applying the *Hamorder* algorithm to renumber the processor IDs, the topology embeddings for the two tasks appear to have better locality, with a lower *ASPL* (Average Shortest Path Length) for each topology embedding. This demonstrates that *Hamorder* enhances the closeness of communication distances among node pairs in a topology embedding for a dispatched task without changing the overall network topology or improving the task mapping algorithm.

The performance of the MPI runtime is influenced by both the computation power of compute nodes and the

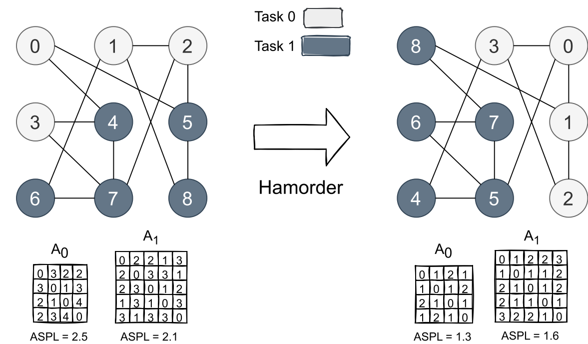


FIGURE 1. **The *Hamorder* algorithm performs graph reordering on *Random* network topologies. The matrix A_n represents the adjacency matrix of the topology embedding for task n, which shows the distance between any source and destination node. The *ASPL* (Average Shortest Path Length) represents the average distance between all node pairs and indicates the locality of the topology embedding.**

communication between them. The communication, in turn, is influenced by the way tasks are mapped. One way to improve task mapping is to reorder the node IDs, which can reduce the distance (i.e., path hops) between communicating compute nodes and minimize the chances of interference among multiple running tasks. In this study, we aim to enhance the performance of parallel applications using the *Hamorder* algorithm. However, while programming in parallel, the maximum improvement of performance is limited by the impact of non-parallelizable parts and parallelization overhead. Hence, we strive to optimize communication capability through a well-designed implementation and configuration of *Hamorder* on *Random* network topologies. Our solution involves using an autotuning method to search for the optimal environment for a specific target application from a vast search space. As a result, the performance of the target parallel application is significantly enhanced within a reasonable amount of time.

Our main contributions in this work are summarized as follows:

- We demonstrate that *Random* network topologies are more suitable for most of the evaluated MPI applications compared to conventional regular topologies, based on better execution performance.
- We introduce a new light-weight graph reordering algorithm, *Hamorder*, to improve parallel application performance on *Random* network topologies. Evaluation results show that *Hamorder* outperforms the state-of-the-art *Gorder* algorithm by up to 27.3%.
- To further optimize the performance of parallel applications, we propose an autotuning framework based on *Hamorder* using *OpenTuner*. Evaluation results show that the framework provides an average execution speedup of 1.38× for target MPI applications, using a search space of various runtime parameter combinations.

The structure of this paper is as follows. In Section II, background information and related works are discussed. Section III presents the *Hamorder* algorithm and the

Hamorder-based autotuning framework. Section IV showcases the evaluation methodology and results. Finally, in Section V, we conclude this paper with a summary of our findings.

II. BACKGROUND AND RELATED WORKS

A. TASK MAPPING PROBLEM

The task mapping and its related concepts can be seen as similar to topology embedding in graph theory, hence we adopt a notation that expands upon the notation used for graph embedding [14].

The communication pattern between processes is represented by a graph $C = (V_c, E_c)$, where V_c is a set of processes and $E_c(u, v)$ denotes the volume of communication from process $u \in V_c$ to process $v \in V_c$. If there is no communication between process u and process v, the value of $E_c(u, v)$ is set to zero. The graph C may contain disconnected components, and isolated vertices represent concurrent execution of unrelated tasks.

Similarly, the physical interconnection network between processors is depicted by a graph $G = (V_g, E_g)$, where V_g is the set of physical compute nodes and $E_g(m, n)$ is the bandwidth capacity of the link between node $m \in V_g$ and node $n \in V_g$. If there is no link between node m and node n, then $E_g(m, n) = 0$. Let $P(m, n)$ be the set of simple paths connecting node m to node n, where each edge is used at most once. And let $T(m, n, p)$ represent the fraction of traffic from node m to node n that is routed through path $p \in P(m, n)$. Typically, the traffic is routed through the shortest path.

A topology mapping is defined by a function $F : V_c \to V_g$, which maps the vertices (processes) of the communication graph C to the vertices (processors or nodes) of the interconnection network graph G. There are two metrics used to evaluate a mapping: *dilation* and *contention*. The length of a path p is represented as $|p|$.

The *dilation* of an edge (u, v) in the communication graph C is calculated as the average length of the paths taken by a message sent from process u to process v:

$$dilation(u, v) = \sum_{p \in P(F(u), F(v))} T(F(u), F(v), p) \cdot |p| \quad (1)$$

The largest *dilation* among all inter-process communications is defined as the *dilation* of the mapping F:

$$dilation(F) = \max_{u, v \in V_c} dilation(u, v) \quad (2)$$

It represents the maximum number of edges traversed by packets and thus plays a crucial role in determining the performance of an application. The *contention* of a link (m, n) in the interconnection network is the ratio of the amount of traffic on the link to its capacity. For simplicity, the capacity of a link is typically set to one. Therefore, the *contention* of an edge $e \in E_g$ is calculated as follows:

$$contention(e) = \sum_{u, v \in V_c} \sum_{p \in P(F(u), F(v)), e \in p} T(F(u), F(v), p) \quad (3)$$

The largest *contention* among all inter-processor communications is defined as the *contention* of the mapping F:

$$contention(F) = \max_{e \in E_g} contention(e) \quad (4)$$

It has a significant impact on the minimum communication latency between tasks.

Both *dilation* and *contention* can be calculated in polynomial time. A simple task mapping requires both *dilation* and *contention* to be equal to one. The regular task mapping typically meets this requirement. However, it becomes challenging to satisfy these requirements when mapping tasks on *Random* topologies, especially when links are shared among multiple parallel tasks. In such cases, both *dilation* and *contention* are larger than one [3]. This scenario is also taken into consideration when mapping tasks on *Random* topologies in this work. The terms mapping and embedding are used interchangeably throughout the rest of the work.

There have been highly scalable cluster management tools for HPC clusters, such as SLURM [15] and Kubernetes Scheduler [16]. While these tools offer a means for allocating compute node resources to users on an exclusive or non-exclusive basis, their primary focus is on scheduling and monitoring tasks on the nodes that have been allocated. Our objective in this work, however, is different. Specifically, we seek to enhance task execution performance primarily through intra-task node mapping. As a future undertaking, we plan to investigate the co-design of Hamorder-based task mapping and scheduling within the context of existing HPC environments.

B. GRAPH REORDERING

Graph reordering is a technique used to improve the performance of graph algorithms by rearranging the vertices and edges in memory. This process does not modify the graph itself and does not require any changes to the underlying algorithms. The reordering of vertices is performed to improve cache locality based on the new vertex IDs assigned during the relabeling process.

Many researchers have proposed various graph reordering techniques, with the most powerful methods leveraging the community structure commonly found in real-world graphs. For instance, the *Gorder* [7] technique comprehensively analyzes the connectivity between vertices and rearranges them such that vertices with common neighbors are placed close to each other in memory. Another example is *Rabbit Order* [8] which is a hierarchical community-based ordering method that leverages the locality derived from hierarchical community structures found in real-world graphs.

There are also graph reordering techniques that focus on temporal locality, which is the rearrangement of vertices based on their degrees. These techniques aim to reduce the cache footprint of frequently accessed vertices for improved cache efficiency. For example, *DBG* (Degree-Based Grouping) [9] is a novel graph reordering technique that aims to preserve graph structure while reducing the cache footprint

of frequently accessed vertices. However, *DBG* only uses coarse-grain reordering, meaning it partitions vertices into a few groups based on their degree while maintaining the original relative order within each group. On the other hand, *HubSort* [10] only sorts frequently accessed vertices, preserving some graph structure while reducing the cache footprint of frequently accessed vertices.

Our proposed technique, *Hamorder*, has similarities to existing graph reordering techniques. For instance, *Hamorder* also requires a preprocessing pass over the graph dataset and does not require any changes to the graph algorithms. Additionally, *Hamorder* is a software-based method that can improve performance without requiring any additional hardware support. However, the main difference between *Hamorder* and other graph reordering algorithms is its focus on improving the performance of parallel applications rather than the cache hit rate in memory. Additionally, *Hamorder* is a fine-grain ordering algorithm that sorts vertices within any group to minimize structural disruption, making it effective in maintaining spatial locality among processes within a dispatched job when using the default *by-order* task mapping approach.

C. TUNING OF MPI APPLICATIONS

The MPI (Message Passing Interface) standard has been successful due to the wide availability of various MPI implementations, such as MPICH [17], which is the dominant implementation used on the majority of the world's top 10 supercomputers. MPI program users aim to minimize communication overhead, however, the performance of the communication library is largely dependent on the MPI library developers.

On one hand, it can be challenging for MPI program developers to optimize their programs when there are a large number of processes involved, such as over 10,000 processes on a supercomputer. On the other hand, as the number of cores on a chip increases and the data hierarchy becomes deeper, the distance between data owned by each process increases, leading to a rise in communication delay.

In data centers, the optimal implementation and configuration of parallel applications are chosen based on the network device and system scale. While improving processor performance and network bandwidth is important, reducing communication overhead, which is a side effect of parallelization, should also receive attention. This issue is especially relevant in interconnection networks that involve thousands of high-speed compute nodes. Therefore, it is crucial to use the fastest possible MPI implementation and configuration.

Autotuning is becoming more popular in domains such as high performance computing (HPC) to optimize programs by automating the search for an optimal implementation or configuration. This can make the optimization process more efficient by searching a larger space than would be possible by hand. Our goal is to explore performance tuning functions without putting a burden on the developers of high-performance parallel programs.

There have been many efforts to optimize MPI communication. For example, MPI point-to-point communication routines can be optimized by using more efficient primitives [18], or through the use of a library for monitoring MPI applications [19]. MPI collective communications can be optimized over wide-area networks by considering network details [20], or through a library like HPC-X [21] for offloading. There have also been developments in algorithms for MPI collective operations [22], [23], [24], as well as optimizations for thread-based MPI implementations [25] and clusters of SMPs [26]. A set of MPI collective communication routines, called STAR-MPI [27], is developed to adapt to system architecture and application workload. There is even a combined compiler and library approach [28] for optimizing MPI communication.

This work, however, goes beyond simply developing MPI collective communication algorithms. It focuses on optimizing the system using *Hamorder*-based autotuning by selecting the optimal or suboptimal combination of network parameters and configurations from a set of communication options. The use of Hamorder-based autotuning offers the potential for further performance improvement.

III. THE OPTIMIZATION BY HAMORDER

A. THE HAMORDER ALGORITHM

In this study, we introduce a new, lightweight node reordering algorithm called *Hamorder* to enhance parallel application performance. Notice that, our focus in this work is on a new algorithm for reordering node IDs. We adopt a default *by-order* task mapping approach [3], which assigns ranks to available processors in ascending order. This means that rank-*n* is initially assigned to processor-*n*, with rank-0 on processor-0, rank-1 on processor-1, rank-2 on processor-2, etc. *Hamorder* performs fine-grained node reordering such that each node is adjacent to at least one node with a neighboring ID. Unlike community-based reordering techniques like *Gorder*, which classify nodes based on their communities, *Hamorder* treats each node equally across the network. Within each task mapping block, *Hamorder* keeps the original order of nodes, preserving the graph structure on a larger scale. The steps of the *Hamorder* algorithm are outlined in Algorithm 1.

The *Hamorder* algorithm is designed to find a solution to the Hamiltonian cycle problem, which is a well-known mathematical problem [29]. To start, a small weight is assigned to each edge in the original network topology T, while a large weight is assigned to each edge in the complement topology T_c. These two topologies are then merged to form a complete topology T_m with varying edge weights. A greedy algorithm is then used to solve the Hamiltonian cycle problem in T_m and find the shortest cycle that visits all the nodes. The node IDs are then reordered based on the order of the nodes in the resulting cycle. The *Hamorder* algorithm operates by adding one node to the solution in each iteration. The node added to the solution is chosen to be the one not already in the cycle

Algorithm 1 The *Hamorder* Algorithm

Require:
Topology $T(N, E)$, where topology T has a node set N and an edge set E

Ensure:
Node-ID mapping $M[|N|]$, where $M[n]$ is the renumbered ID of node n

1: **procedure** hamorder(T)
2: *STEP 1*: Set `weight` $= 1$ to each edge of T
3: *STEP 2*: Obtain the complement topology of T, i.e., T_c, and set `weight` $=$ `MAX_INT` to each edge of T_c
4: *STEP 3*: Merge topologies T and T_c to form a new topology T_m, i.e., $T_m = T \cup T_c$
5: *STEP 4*: Solve the Hamiltonian cycle problem to generate a node list $L[|N|]$ by using a greedy method in T_m
6: *STEP 5*: Renumber N based on the resulting $L[|N|]$
7: `id` $\leftarrow 0$
8: **for** $i = 0 \rightarrow |N| - 1$ **do**
9: $M[L[i]] \leftarrow$ `id++`, where $L[i]$ is original node ID
10: **end for**
11: **end procedure**

whose connection to the previous node incurs the least cost. As a result, the time complexity of the *Hamorder* algorithm is $O(|N|^2)$. As an offline graph reordering approach, it has minimal impact on the real-time MPI application execution performance.

The results of node ID renumbering on a simple sample network topology are depicted in Figure 2 for a synthetic example. For comparison, *Gorder*, a typical community-based graph ordering technique, is used. The initial node numbering is not suitable for communication patterns where neighboring ranks have more frequent communication as they are mapped to distant processors. *Gorder* improves this by performing community detection and node clustering operations on the network topology. After dividing the network into communities, the node IDs are reassigned such that nodes within the same community have close IDs. This is because nodes within the same community have more connections to each other than those outside the community. Within each community, the node IDs are renumbered to be contiguous. However, this approach is a coarse-grain approach, meaning it ignores the ordering of nodes within a single community, as it was originally designed to improve cache hit rate. Therefore, neighboring nodes within the same community are not guaranteed to have adjacent IDs.

In contrast to other techniques, our proposed solution, *Hamorder*, reorders the processors so that neighboring ranks are mapped to processors that are close to each other. This is because these ranks are likely to have more communication with one another.

Hamorder takes a comprehensive approach to node reordering, starting from the entire topology, and ensuring

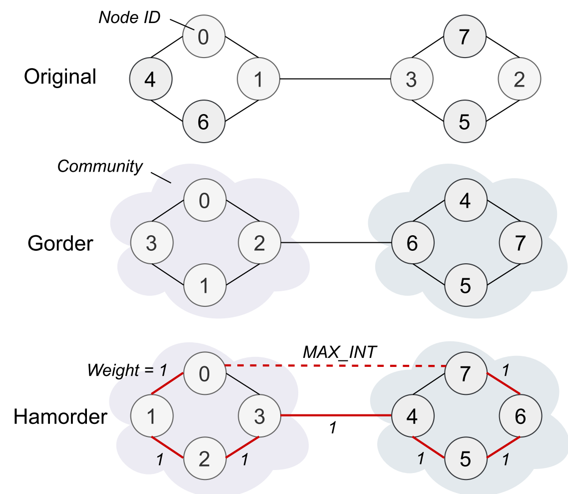


FIGURE 2. Illustration of node ID renumbering by a simple example graph.

that each node has at least one neighboring node with an adjacent ID. In this example, the nodes are renumbered to form a path from node 0 to node 7 ($0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$) in order to approximate a solution to the Hamiltonian cycle problem in the network topology.

This solution offers the benefit of reordering node IDs within a task mapping, regardless of the topology fragment, while community-based techniques like *Gorder* focus on reordering with a community granularity. In a scenario where nodes 0 to 2 are already occupied and a new parallel task of two ranks is to be dispatched, *Gorder* would assign the task to nodes 3 to 4 (4-hop away), worsening the topology embedding compared to the original node ordering. However, with *Hamorder*, the task would be assigned to nodes 3 to 4 (1-hop away) using the same mapping procedure.

Therefore, compared to community-based graph reordering techniques, *Hamorder* offers the advantage of reducing communication distance between nodes within a contiguous node embedding, assuming that neighboring ranks have more communication. This helps to improve the performance of parallel applications. The effectiveness of *Hamorder* relies on the presupposition that communication among neighboring processes (ranks) is more intense. In a *Random* network topology with irregular or uncertain communication patterns, determining the optimal graph reordering solution is challenging.

The primary focus of this research is on reordering nodes within a *Random* topology with a specified network degree, while assuming that the links are predetermined. In a prior study [13], we demonstrated the feasibility of utilizing a wireless link to establish 1-hop communication between compute nodes. It is anticipated that the *Hamorder* algorithm, in combination with wireless links for inter-node communication, could enhance application execution performance even further. We intend to explore this co-design aspect in our future endeavors.

B. HAMORDER-BASED AUTOTUNING

In the previous section, we introduced *Hamorder* as a solution to optimize the mapping of parallel application processes to processors in the network. In this section, we further improve the performance of the high-performance parallel program by tuning it based on *Hamorder*.

Our autotuning process to optimize MPI communication is approached as a search problem. The search space consists of a variety of configurations, including network topologies, MPI implementations, collective communication algorithms, and others. The performance of each configuration is evaluated, for instance, in terms of application execution time, and the results are recorded.

To explore the search space of possible implementations and optimizations based on *Hamorder*, we use *OpenTuner* [30]. This tool provides an extensible configuration representation, which can handle complex and user-defined data types, and allows us to define the search space by creating a configuration representation that includes a set of parameters for *OpenTuner* to search through. In the following, we will delve further into our tuning framework for MPI applications.

1) TOPOLOGY OPTIMIZATION

In this work, we aim to optimize the topology of a generated *Random* network based on design parameters like *BiBW* (Bisection Bandwidth) [31] and *ASPL* (Average Shortest Path Length) [32].

BiBW represents the bandwidth available between two bisected partitions in the network and accounts for the bottleneck bandwidth of the entire system. It is considered a better representation of the bandwidth characteristics of the network than any other metric. On the other hand, *ASPL* is the average number of hops along the shortest paths for all possible node pairs, measuring the efficiency of data transfer in the network.

We consider two approaches for topology optimization: **Opt-maxBisec** and **Opt-minASPL**. The former optimizes the base *Random* network to generate a topology with the possible largest *BiBW*. This is done using a *2-Opt* local search algorithm [33], which replaces two old edges with two new ones in each iteration, keeping the degree of each node unchanged and increasing the *BiBW*. The algorithm stops when the *BiBW* has not evolved for a specified number of iterations (I) and generates the resulting graph.

Similarly, **Opt-minASPL** optimizes the base *Random* network to generate a topology with the possible minimum *ASPL*. The algorithm used for topology generation is similar to that of **Opt-maxBisec**.

In this work, we set $I = 1,000$ for a realistic convergence speed in the *2-Opt* heuristic. The both topology optimization algorithms are described in Algorithm 2.

Algorithm 2 The Algorithm of Topology Optimization

Require:
Base topology T_b, Iteration limit I
Ensure:
Optimized topology T_o

```
1:  procedure Opt-maxBisec(T_b, I)
2:      T_o ← T_b
3:      maxBiBW ← calcBiBW(T_b)
4:      i ← 0
5:      while i++ < I do
6:          T_t ← 2-Opt(T_o)
7:          BiBW ← calcBiBW(T_t)
8:          if BiBW > maxBiBW then
9:              T_o ← T_t
10:             maxBiBW ← BiBW
11:             i ← 0
12:         end if
13:     end while
14: end procedure

15: procedure Opt-minASPL(T_b, I)
16:     T_o ← T_b
17:     minASPL ← calcASPL(T_b)
18:     i ← 0
19:     while i++ < I do
20:         T_t ← 2-Opt(T_o)
21:         ASPL ← calcASPL(T_t)
22:         if ASPL < minASPL then
23:             T_o ← T_t
24:             minASPL ← ASPL
25:             i ← 0
26:         end if
27:     end while
28: end procedure
```

2) MPI IMPLEMENTATION

Choosing the right implementation of the MPI standard can be challenging for MPI programmers, as they often exhibit similar performance but have different implementation details, such as communication algorithms. To address this, we consider three popular and stable MPI implementations that work well with multi-threaded programs: **MPICH**, **OpenMPI**, and **MVAPICH2** [34]. These implementations are considered to be typical and commonly used.

3) COLLECTIVE COMMUNICATION ALGORITHM

Collective communication operations are critical to the performance of MPI applications. Each MPI implementation offers various algorithms for each operation and automatically selects the one that best suits the situation based on factors like communication data size, number of processors, communicator, or communication library. These selections are a result of both empirical testing and theoretical analysis and can vary greatly between MPI implementations [27]. While MPI programmers can manually optimize the algorithms used for collective operations, they must take into account a variety of parameters.

TABLE 1. MPI benchmarks used for evaluation in this work.

MPI App.	Description
FT	Discrete 3D fast Fourier Transform
IS	Integer Sort
CG	Conjugate Gradient
BT	Block Tri-diagonal solver
SP	Scalar Penta-diagonal solver
MG	Multi-Grid on a sequence of meshes
LU	Lower-Upper Gauss-Seidel solver
MM	Calculation of product of two matrices
Graph500	64 iterations of a BFS-validate loop

This study examines the following algorithms for tuning MPI applications:

Alltoall: The algorithms include: **Bruck**, **2dmesh**, **3dmesh**, **Recursive doubling**, **Pair**, **Ring**, and **Basic linear**.

Allreduce: The algorithms include: **Recursive doubling**, **Logical ring**, **Rabenseifner**, **Rabenseifner var1**, and **Rabenseifner var2**.

Allgather: The algorithms include: **Bruck**, **2dmesh**, **3dmesh**, **Recursive doubling**, **Pair**, **Ring**, and **Spread**.

Bcast: The algorithms include: **Binomial tree**, **Flat tree**, **Scatter logical-ring allgather**, and **Scatter recursive-doubling allgather**.

The collective communication routines used in our benchmark MPI applications are selected from those available in our simulation framework, *SimGrid* [35], and have since been implemented. Further details about each algorithm can be found in [27].

IV. EVALUATION

A. ENVIRONMENT

To assess the effectiveness of the proposed *Hamorder* algorithm in enhancing the intra-task communication efficiency between compute nodes, we employ MPI applications that place a heavy emphasis on inter-node communication in our experiments. In this study, we utilize the *SimGrid* simulation framework (version 3.26) [35] to evaluate the performance of parallel application benchmarks, which include: *FT*, *IS*, *CG*, *BT*, *SP*, *MG*, *LU* taken from the NAS parallel benchmarks [36], as well as *MM* (Matrix Multiplication) and *Graph500* [37]. We adopt Class A as the problem size for *FT*, *IS*, *CG*, *BT*, *SP*, *MG*, and *LU*. Each matrix in the *MM* application has a size of $1,000 \times 1,000$. A description of these MPI applications is provided in Table1.

In our simulations, we have set the computing power of each node to 100 GFlops, while the cable bandwidth has been set to 40 Gbps. Additionally, the switch latency and bandwidth have been set to 60 ns and 10 Pbps, respectively. It is important to note that the execution time of the benchmark applications in *SimGrid* ranges from 5 milliseconds to 5 seconds.

With the aim of minimizing the simulated time of a target MPI application, we customize the objective of *OpenTuner*, which by default aims to minimize the execution time of a running program. Our custom objective is achieved by defining a run function, which evaluates the fitness of a configuration in the search space and produces a measurement result. Using *OpenTuner*, we search the space of configuration parameter objects with the goal of minimizing the execution time of the target MPI program.

TABLE 2. Execution time (s) of MPI applications on 64-node topologies. The best/worst result is shown in green/red.

MPI	Conventional Topologies		Random Topologies (Degree-6)			
App.	3-D Mesh	3-D Torus	Dilation-1	Dilation-2	Dilation-3	Dilation-4
FT	0.0200	0.0296	0.0125	0.0128	0.0131	0.0135
IS	0.0105	0.0120	0.0055	0.0055	0.0056	0.0056
CG	1.0875	1.2793	0.9446	0.9510	0.9575	0.9639
BT	0.3817	0.2741	0.4167	0.4201	0.4242	0.4299
SP	0.6189	0.4629	0.6779	0.6832	0.6889	0.6958
MG	0.0285	0.0275	0.0234	0.0241	0.0247	0.0254
MM	0.2932	0.2713	0.1855	0.1881	0.1865	0.1872

This study employs pre-defined *Random* network topologies for parallel task executions, where the distance between compute nodes is determined by the number of minimal path hops. To find the shortest path between compute nodes in an unweighted graph, we utilize the Dijkstra algorithm, which is not adaptive and does not alter its behavior based on input changes. Routing deadlocks are not a concern in this context and, therefore, are not considered. In order to ensure a fair comparison, we employ the Dijkstra algorithm for all parallel task executions during the evaluation.

B. IMPACT OF TOPOLOGY EMBEDDING

This section focuses on examining the effect of parallel application execution on various network topologies. The knowledge of network topology plays a crucial role in optimizing the performance of MPI implementations. The physical structure of the parallel machine and the process topology for application execution are described. The connection relationships between nodes or processors in the network are defined by the chosen topology.

In this research, we aim to analyze a graph with a *Random* topology, taking into consideration the number of nodes (graph size) and maximum node degree (radix). For comparison, we also analyze conventional *Mesh* and *Torus* topologies with the same graph size and maximum node degree. Our simulation results present the execution times of these applications on different topologies. The execution time of each application is influenced by its topology and degree, which we set to 6.

Table 2 shows the execution times of MPI applications on 64-node topologies. The conventional topologies are 3-D *Mesh* ($4 \times 4 \times 4$) and 3-D *Torus* ($4 \times 4 \times 4$), while the *Random* topologies are generated with a degree of 6 and varying *dilations*. The *dilation* in topology embedding refers to the shortest path length between two nodes of an edge, in terms of number of hops. In general, an increase in the specified *dilation* results in easier topology embedding, but also longer paths and communication latency between non-adjacent nodes.

The execution times of most MPI applications, such as *FT*, *IS*, *CG*, *MG*, and *MM*, are shortest when they are executed on the *Random* topologies with dilation-1. For instance, running *MM* on *Random* topologies leads to faster execution times than when running on conventional topologies such as 3-D *Mesh* and 3-D *Torus*. This is because conventional topologies may cause long communication hops between compute nodes, while these can be reduced by shortcut connections in *Random* topologies with specific communication patterns.

On the other hand, 3-D *Torus* has the shortest execution time for *BT* and *SP*, while *Random* topologies with dilation-4 perform the worst. This is because *BT* and *SP* have more regular communication patterns, where compute nodes have equal chances to exchange messages with each other. In such cases, regular topologies like *Mesh* and *Torus* are more suitable.

It is important to note that a regular topology may not be suitable for all parallel applications, especially those with irregular or uncertain communication patterns. In comparison, a *Random* topology seems to be a better choice for most parallel applications that have irregular or uncertain communication patterns. However, for *Random* topologies, a larger topology embedding *dilation* leads to slightly degraded execution performance. It should also be noted that this section does not consider communication *contention* between multiple large-dilation embedded tasks, so the side effects of large-dilation topology embedding seem to be limited.

C. IMPACT OF LOCALITY

Locality is connected to the topology embedding *dilation* for task mapping. A large-dilation topology embedding can result in longer path hops and increased communication latency between non-neighboring compute nodes. When a non-contiguous topology embedding is used, one or more links can be shared by multiple parallel applications, which leads to bandwidth *contention* as multiple applications simultaneously use the same link.

In this section, we assess the impact of locality on network *contention* between parallel applications. We look at the average execution time of parallel applications when link sharing occurs between them.

For our experiment, we use a 2-D *Torus* topology (4 × 2) and evaluate the impact of different interleaving patterns on two parallel applications. In pattern A (Fig.3(a)), the two tasks run without link sharing and are each mapped to a 2-D *Mesh* (2 × 2) topology with a dilation-1 topology embedding, resulting in good locality for their executions. This is used as our baseline. In contrast, with patterns B (Fig.3(b)), C (Fig. 3(c)), and D (Fig.3(d)), one or more links are shared by the two tasks, thus their execution times will be impacted by link *contention*.

The time-independent traces of the MPI applications were obtained using *SimGrid*, and multiple MPI applications were then run simultaneously in replay mode using *Batsim* [38] to evaluate the average execution time of two applications over a 2-D *Torus* (4 × 2) topology.

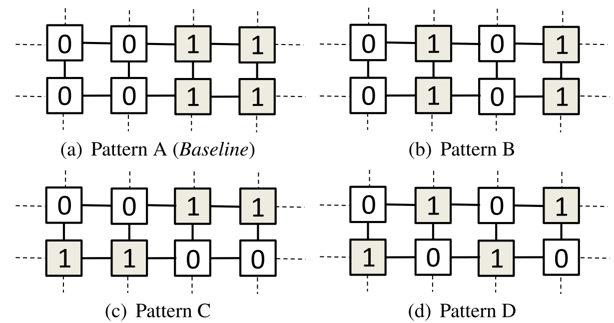


FIGURE 3. Simultaneous executions of two parallel tasks over the same network.

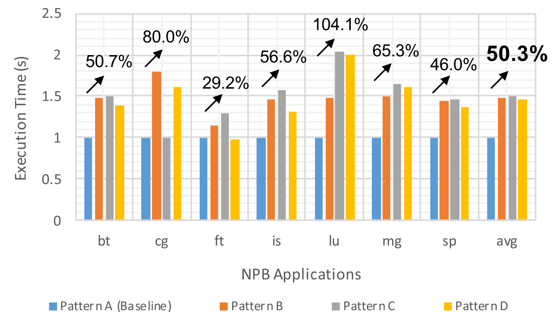


FIGURE 4. Average execution time of two simultaneously running NPB applications.

As seen in Fig. 4, pattern A has the best execution time because there is no link sharing between the two tasks, allowing each task to make optimal use of the bandwidth over the network topology. Among all MPI applications, *LU* is the most impacted by network *contention* due to link sharing, with a 104.1% increase in execution time when using pattern C compared to pattern A. On the other hand, *FT* is relatively immune to network *contention*, with only a 29.2% increase in execution time when using pattern C compared to pattern A. The average execution time of the evaluated MPI applications increases by about half when using patterns B, C, or D, highlighting the significance of locality for topology embedding and its potential impact on the execution performance of parallel applications.

D. GRAPH REORDERING BY HAMORDER

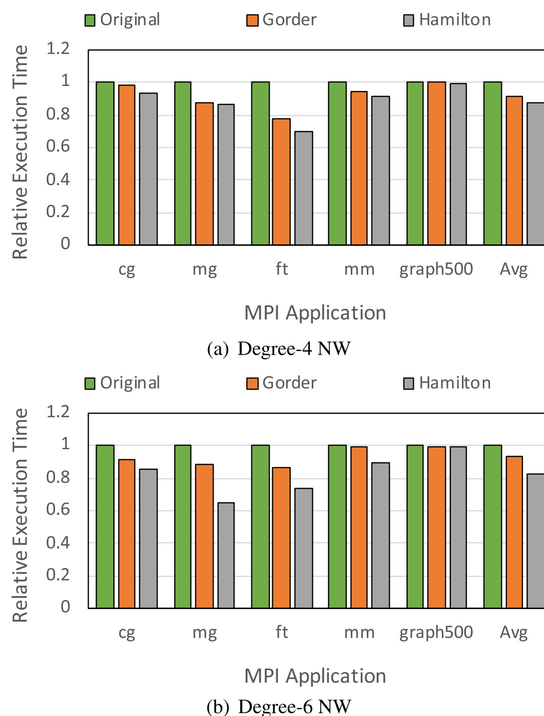
To enhance the locality of the topology embedding with a finer resolution on *Random* topologies, we propose using the *Hamorder* graph reordering algorithm, which does not require modification of the entire network topology or improvement of the default task mapping approach. The state-of-the-art graph reordering algorithm, *Gorder*, shares several similarities with our proposed *Hamorder* algorithm. Both algorithms require a preprocessing pass over the graph dataset and do not necessitate any modifications to the graph algorithms. Furthermore, *Gorder* is a software-based method that can boost performance without the need for additional hardware support. Although *Gorder*'s primary aim is to increase the cache hit rate in memory, it also facilitates the maintenance of a certain spatial locality among processes when using the default *by-order* task mapping approach.

In contrast, the proposed *Hamorder* algorithm is a fine-grain ordering algorithm that sorts vertices within each group to minimize structural disruption during task mapping. *Gorder* was initially designed for optimizing cache hit rates rather than high-performance parallel applications. In this research, we assess the performance of *Gorder* and *Hamorder* by comparing them on the same *Random* network topology, utilizing communication-intensive MPI applications.

The comparison results of the three graph ordering methods are depicted in Figure 5, using 64-node *Random* network topologies. *Random* network topologies are characterized by the absence of a specific rule or algorithm governing the connections between nodes. Hence, these connections are established randomly. We divided the experiment into two groups based on network degree, i.e., degree-4 and degree-6 *Random* topologies, using the original node numbering method as a baseline. Notice that, in our evaluation, the network degree is equivalent to the node degree. This means that the degree-4 and degree-6 network topologies refer to each node having 4 and 6 neighboring nodes, respectively. The results indicate a similar trend for both the degree-4 and degree-6 cases. The use of an optimized graph reordering algorithm, such as *Gorder* or *Hamorder*, provides an advantage over the baseline ordering method, as it improves the locality of topology embedding through node renumbering.

Furthermore, *Hamorder* outperforms *Gorder* for nearly all the evaluated MPI applications in both degree-4 and degree-6 topologies. This is because *Hamorder* considers reordering not only among multiple topology embeddings but also within a single topology embedding, while *Gorder* does not take into account node reordering within a node group. On average, *Hamorder* reduces the execution time by 12.1% (degree-4) and 17.4% (degree-6) compared to the original ordering, and by 4.0% (degree-4) and 11.2% (degree-6) compared to *Gorder*.

We observe that the improvement achieved by *Hamorder* is limited, as in this section we have fixed other network parameters and configurations that could potentially result in further improvements in the execution performance of parallel applications if altered based on *Hamorder*.



(a) Degree-4 NW

(b) Degree-6 NW

FIGURE 5. Relative execution time on 64-node *Random* network topologies.

E. HAMORDER-BASED AUTOTUNING

In this section, we delve further into the potential of enhancing parallel applications by presenting an autotuning method based on the *Hamorder* algorithm for MPI applications, which searches over a large number of user-defined configuration parameters. The autotuning method includes a comprehensive framework for describing complex search spaces for MPI communication, which significantly reduces the burden on parallel program developers. The total number of tuning tests for each MPI application was set to 100.

Figure 6 displays the autotuning progress for five MPI applications. The X-axis represents the ID of tuning tests and the Y-axis shows the best execution time achieved by each tuning run, which gives the best searching result at that point in time. The results are compared with those obtained from conventional topologies, including 2-D *Mesh* (8×8), 3-D *Mesh* ($4 \times 4 \times 4$), 2-D *Torus* (8×8), and 3-D *Torus* ($4 \times 4 \times 4$). As the tuning tests progress, the application execution time decreases due to improved configuration parameters.

In this section, we show the improvement in the execution time of parallel applications through the autotuning process. The performance of the application called *MG* initially performs worse compared to conventional topologies, but improves gradually as the tuning process continues. On the other hand, for other applications such as *CG*, *FT*, *MM*, and particularly *Graph500*, the initial execution time for the tuned case is either comparable or even better than the conventional topologies, and it continues to improve throughout the tuning process. By the end of the tuning process, the tuned case results in a significant improvement for all tested parallel applications.

To measure the improvement in the autotuning results, we use the gap between the final tuned execution time and the initial one. Figure 7 illustrates the speedup calculated as follows:

$$I = \frac{T_i}{T_o} \tag{5}$$

where T_i denotes the initial execution time at the first search trial and T_o is the best (shortest) execution time during the tuning period.

The evaluation results show that most of the MPI applications are significantly accelerated by autotuning. In particular, *MG* achieved up to 2.29× speedup with the optimal configuration found. On average, the MPI applications achieved 1.38× speedup in our defined search space with just 100 tuning tests, rather than an exhaustive search.

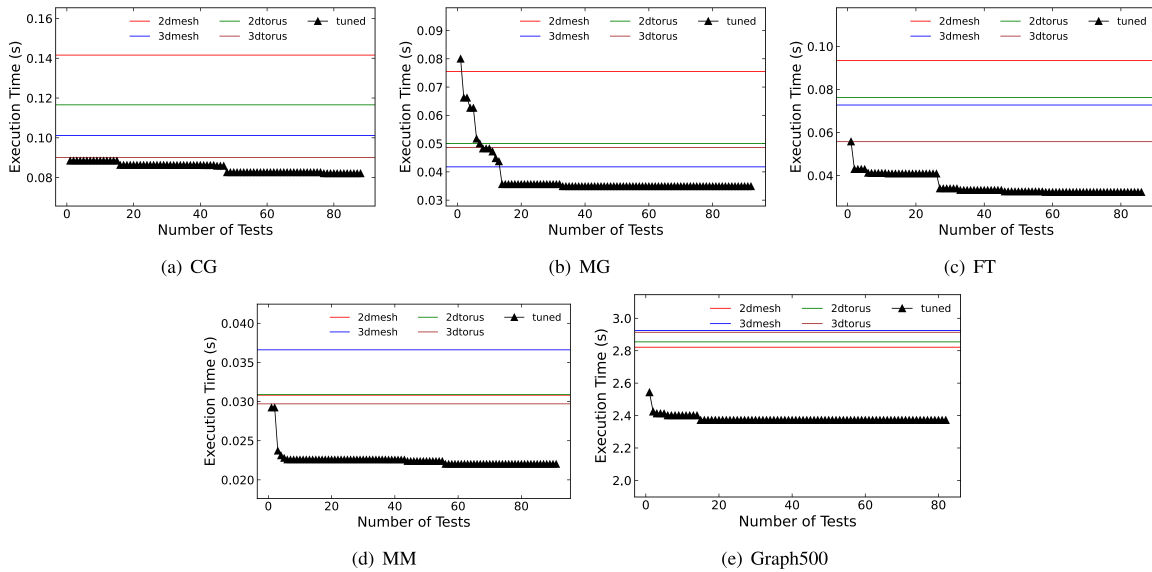


FIGURE 6. Optimization results for MPI applications.

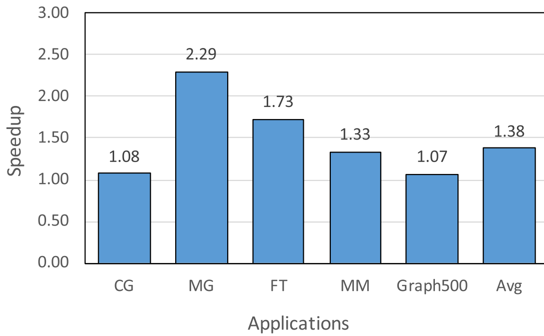


FIGURE 7. Speedup by autotuning MPI applications.

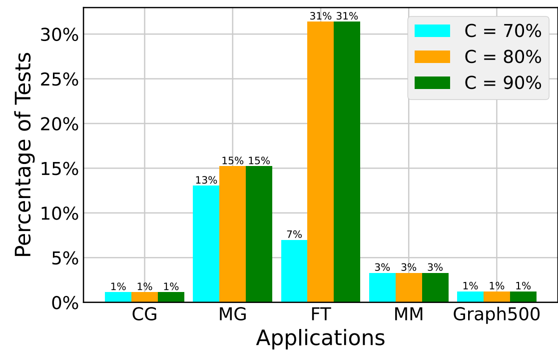


FIGURE 8. Efficiency of autotuning progress for MPI applications.

To demonstrate the efficiency of the application tuning, we measure the number of tuning tests needed to reach near-optimal performance. Figure 8 shows the percentage of tuning tests required to get close to the optimal performance during the tuning period. The proximity (C) to the optimal performance is defined as follows:

$$C(n) = 1 - \frac{T_n - T_o}{T_o} \tag{6}$$

where T_n denotes the temporarily best (shortest) execution time by the n-th tuning test and T_o denotes the best (shortest) execution time throughout the tuning period.

Evaluation results indicate that a minimal number of search trials are sufficient to achieve near-optimal performance for all the tested MPI applications. For example, only 1%, 15%, 31%, 3%, and 1% of search trials are required to reach 90% optimal performance for *CG*, *MG*, *FT*, *MM*, and *Graph500*, respectively. In essence, the proposed autotuning framework effectively extends the performance enhancement capabilities of *Hamorder* to parallel applications on *Random* topologies.

V. CONCLUSION

In our study, we introduced *Hamorder*, a lightweight reordering algorithm that enhances the locality of topology embedding through both inter- and intra-task node renumbering on *Random* network topologies. Unlike traditional graph reordering methods, *Hamorder* prioritizes improving the execution performance of parallel applications. Our results indicate that *Hamorder* outperforms the current state-of-the-art method, *Gorder*, resulting in an up to 27.3% increase in speed.

We further leveraged *Hamorder* in an autotuning framework powered by *OpenTuner* to optimize the execution of parallel applications on *Random* topologies. Our framework achieved a speedup of up to 2.29× with a minimal number of search trials.

In future work, we plan to increase the efficiency of *Hamorder* and conduct experiments on large-scale systems. Additionally, we aim to optimize the representation of the search space in autotuning by considering the importance of different configuration parameters. This can greatly reduce the tuning time and improve the search for optimal or suboptimal configurations.

REFERENCES

[1] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI usage on a production supercomputer," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2018, pp. 386–400.

[2] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proc. Int. Conf. Supercomput. (ICS)*, New York, NY, USA, May 2011, pp. 75–84, doi: 10.1145/1995896.1995909.

[3] O. Tuncer, V. J. Leung, and A. K. Coskun, "PaCMap: Topology mapping of unstructured communication patterns onto non-contiguous allocations," in *Proc. 29th ACM Int. Conf. Supercomput.*, Jun. 2015, pp. 37–46.

[4] T. Agarwal, A. Sharma, A. Laxmikant, and L. V. Kale, "Topology-aware task mapping for reducing communication contention on large parallel machines," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, Apr. 2006, p. 10.

[5] O. Tuncer, Y. Zhang, V. Leung, and A. Coskun, "Task mapping on a dragonfly supercomputer," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2017.

[6] J. Galvez, N. Jain, and L. V. Kale, "Automatic topology mapping of diverse large-scale parallel applications," in *Proc. Int. Conf. Supercomput. (ICS)*. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 1–10.

[7] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proc. Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 1813–1828, doi: 10.1145/2882903.2915220.

[8] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 22–31.

[9] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," 2020, *arXiv:2001.08448*.

[10] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 293–302.

[11] S. A. Al-Gailani, M. F. M. Salleh, A. A. Salem, R. Q. Shaddad, U. U. Sheikh, N. A. Algeelani, and T. A. Almohamad, "A survey of free space optics (FSO) communication systems, links, and networks," *IEEE Access*, vol. 9, pp. 7353–7373, 2021.

[12] I. Fujiwara, M. Koibuchi, T. Ozaki, H. Matsutani, and H. Casanova, "Augmenting low-latency HPC network with free-space optical links," in *Proc. 21st Int. Conf. High-Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 390–401.

[13] Y. Hu and M. Koibuchi, "Enhancing job scheduling on inter-rackscale datacenters with free-space optical links," *IEICE Trans. Inf. Syst.*, vol. E101.D, no. 12, pp. 2922–2932, 2018.

[14] A. Rosenberg, "Issues in the study of graph embeddings," in *Graphtheoretic Concepts in Computer Science* (Lecture Notes in Computer Science), vol. 100. Berlin, Germany: Springer, 1981.

[15] N. A. Simakov, M. D. Innus, M. D. Jones, R. L. DeLeon, J. P. White, S. M. Gallo, A. K. Patra, and T. R. Furlani, "A Slurm simulator: Implementation and parametric analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. Jarvis, S. Wright, and S. Hammond, Eds. Cham, Switzerland: Springer, 2018, pp. 197–217.

[16] H. Ishak, S. A. Makhlouf, and G. Belalem, "A proposal of kubernetes scheduler using machine-learning on CPU/GPU cluster," in *Proc. Comput. Sci. On-Line Conf.*, Aug. 2020, pp. 567–580.

[17] *MPICH*. Accessed: Jan. 20, 2023. [Online]. Available: http://www.mpich.org/

[18] L. Mario, S. Pakin, and A. Chien, "Efficient layering for high speed communication: The MPI over fast messages (FM) experience," *Cluster Comput.*, vol. 2, pp. 107–116, Sep. 1999.

[19] E. Jeannot and R. Sartori, "Improving MPI application communication time with an introspection monitoring library," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2020, pp. 691–700.

[20] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," in *Proc. 7th ACM SIGPLAN Symp. Princ. Pract. parallel Program.* New York, NY, USA: Association for Computing Machinery, May 1999, pp. 131–140, doi: 10.1145/301104.301116.

[21] J. Jose, "Optimizing MPI collective communication using HPC-X on AzureHPC VMs," Microsoft, Redmond, WA, USA, Tech. Rep., May 2020. [Online]. Available: https://techcommunity.microsoft.com/t5/azure-compute-blog/optimizing-mpi-collective-communication-using-hpc-x-on-azurehpc/ba-p/1356740

[22] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of MPI collective communication on BlueGene/L systems," in *Proc. 19th Annu. Int. Conf. Supercomput.*, Jun. 2005, pp. 253–262.

[23] C. Sudhakar, T. Ramesh, and K. Waghmare, "Path based optimization of MPI collective communication operation in cloud," in *Proc. Int. Conf. Comput., Power Commun. Technol. (GUCON)*, Sep. 2018, pp. 595–599.

[24] H. Zhou, J. Gracia, and R. Schneider, "MPI collectives for multi-core clusters: Optimized performance of the hybrid MPI+MPI parallel codes," in *Proc. Workshop Proc. 48th Int. Conf. Parallel Process. (ICPP)*. New York, NY, USA: Association for Computing Machinery, Aug. 2019, doi: 10.1145/3339186.3339199.

[25] J. Adam, M. Kermarquer, J.-B. Besnard, L. Bautista-Gomez, M. Pérache, P. Carribault, J. Jaeger, A. D. Malony, and S. Shende, "Checkpoint/restart approaches for a thread-based MPI runtime," *Parallel Comput.*, vol. 85, pp. 204–219, 2019, doi: 10.1016/j.parco.2019.02.006.

[26] S. Sistare, R. vandeVaart, and E. Loh, "Optimization of MPI collectives on clusters of large-scale SMP's," in *Proc. ACM/IEEE Conf. Supercomput.*, Jan. 1999, p. 23.

[27] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: Self tuned adaptive routines for MPI collective operations," in *Proc. 20th Annu. Int. Conf. Supercomput. (ICS)*. New York, NY, USA: Association for Computing Machinery, Jun. 2006, pp. 199–208, doi: 10.1145/1183401.1183431.

[28] A. Karwande, X. Yuan, and D. K. Lowenthal, "CC-MPI: A compiled communication capable MPI prototype for Ethernet switched clusters," in *Proc. 9th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*. New York, NY, USA: Association for Computing Machinery, Jun. 2003, pp. 95–106, doi: 10.1145/781498.781514.

[29] *Hamiltonian Path Problem*. Accessed: Jan. 20, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Hamiltonian_path_problem

[30] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *Proc. 23rd Int. Conf. Parallel Architectures Compilation (PACT)*, Aug. 2014, pp. 303–315.

[31] *Bisection Bandwidth*. Accessed: Jan. 20, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Bisection_bandwidth

[32] *Average Path Length*. Accessed: Jan. 20, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Average_path_length

[33] *2-OPT*. Accessed: Jan. 20, 2023. [Online]. Available: https://en.wikipedia.org/wiki/2-opt

[34] *Using MPI*. Accessed: Jan. 20, 2023. [Online]. Available: https://www.carc.usc.edu/user-information/user-guides/software-and-programming/mpi

[35] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *J. Parallel Distrib. Comput.*, vol. 74, no. 10, pp. 2899–2917, Oct. 2014.

[36] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. (Mar. 1994). *The NAS Parallel Benchmarks*. [Online]. Available: https://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf

[37] *Graph500*. Accessed: Jan. 20, 2023. [Online]. Available: https://graph500.org/?page_id=462

[38] P.-F. Dutot, M. Mercier, M. Poquet, and O. Richard, "Batsim: A realistic language-independent resources and jobs management systems simulator," in *Job Scheduling Strategies for Parallel Processing*. Springer, Jul. 2017, pp. 178–197, doi: 10.1007/978-3-319-61756-5_10.

YAO HU (Member, IEEE) received the M.S. degree from the Beijing University of Posts and Telecommunications, China, in 2009, and the Ph.D. degree from the Department of Computer Science and Engineering, Waseda University, Tokyo, Japan, in 2015. He is currently an Assistant Professor with Keio University, Yokohama, Kanagawa, Japan. His main research interests include high-performance computing and graph computation.