algorithms

MDPI

Article

Finding Bottlenecks in Message Passing Interface Programs by Scalable Critical Path Analysis

Vladimir Korkhov [1,*], Ivan Gankevich [1], Anton Gavrikov [1], Maria Mingazova [1], Ivan Petriakov [1], Dmitrii Tereshchenko [1], Artem Shatalin [2] and Vitaly Slobodskoy [2]

[1] Faculty of Applied Mathematics and Control Processes, Saint Petersburg State University, Universitetskaya emb. 7-9, St. Petersburg 199034, Russia; i.gankevich@spbu.ru (I.G.); gavrikovantonkapi@gmail.com (A.G.); marie.mingazova@gmail.com (M.M.); st049350@student.spbu.ru (I.P.); st064145@student.spbu.ru (D.T.)

[2] Huawei, Nizhny Novgorod 603006, Russia; artem.shatalin@huawei.com (A.S.); vitaly.slobodskoy@huawei.com (V.S.)

* Correspondence: v.korkhov@spbu.ru

Abstract: Bottlenecks and imbalance in parallel programs can significantly affect performance of parallel execution. Finding these bottlenecks is a key issue in performance analysis of MPI programs especially on a large scale. One of the ways to discover bottlenecks is to analyze the critical path of the parallel program: the longest execution path in the program activity graph. There are a number of methods of finding the critical path; however, most of them suffer a performance drop when scaled. In this paper, we analyze several methods of critical path finding based on classical Dijkstra and Delta-stepping algorithms along with the proposed algorithm based on topological sorting. Corresponding algorithms for each approach are presented including additional enhancements for increasing performance. The implementation of the algorithms and resulting performance for several benchmark applications (NAS Parallel Benchmarks, CP2K, OpenFOAM, LAMMPS, and MiniFE) are analyzed and discussed.

Keywords: parallel program; MPI; performance; bottleneck; imbalance; critical path; program activity graph; topological sorting; Dijkstra; Delta-stepping

Citation: Korkhov, V.; Gankevich, I.; Gavrikov, A.; Mingazova, M.; Petriakov, I.; Tereshchenko, D.; Shatalin, A.; Slobodskoy, V. Finding Bottlenecks in Message Passing Interface Programs by Scalable Critical Path Analysis. *Algorithms* **2023**, *16*, 505. https://doi.org/10.3390/a16110505

Academic Editors: Charalampos Konstantopoulos and Grammati Pantziou

Received: 10 August 2023
Revised: 5 October 2023
Accepted: 19 October 2023
Published: 31 October 2023

1. Introduction

Despite the fact that parallel programming technologies have been studied and improved for many years finding a bottleneck in a parallel MPI program is still difficult. MPI (Message Passing Interface) [1] offers a parallel programming model that allows for the sending of messages between any two processes and also conducts collective communications between an arbitrary number of processes; however, the notion of a task is implicit in MPI—any code between subsequent MPI calls is considered to be a task. There are many possible implementations of point-to-point and collective communication. A simple `MPI_Send` call can copy the data in the internal buffer and return immediately or block until the user-supplied data are sent. In `MPI_Reduce` call, all parallel processes may not wait for each other but continue execution without blocking after their portion of the data has been processed. As a result, different implementations may produce different information dependencies between the tasks. It is not easy to use such dependencies to find a bottleneck, and we have to resort to a «trial-and-error» approach instead.

One approach that uses explicit dependencies between the tasks (and suffers from the aforementioned problems) is based on critical path finding [2]. We represent parallel program execution as a graph in which vertices represent time points and edges represent either the computation between subsequent MPI calls or communication inside MPI calls. Then, we find a critical path between the first and the last vertex in this graph. Reducing the time of any operation (computation or communication) that has the corresponding

edge on the critical path reduces the total execution time of the program, i.e., the critical path represents all possible bottlenecks in the program.

In order for this approach to be successful, we have to ensure that the graph is acyclic. This is especially difficult when a program tries to do collective operation using point-to-point operations: e.g., each process sends data to the right neighbor and receives data from the left neighbor simultaneously. Graph cycles make the graph unsuitable for parallel critical path-finding algorithms and slow down sequential ones. The most robust way of implementing this approach is to build it in the MPI library itself; then, we do not have to guess how each MPI call is actually implemented.

The aforementioned complexities led us to believe that there should be a more robust approach that uses implicit dependencies between the tasks. This approach is based on topological sorting: sorting the vertices by the lengths of their longest incoming paths produces a topological ordering. Usually topological sorting is used to determine the execution order of dependent tasks. Here, we have already executed all of the tasks, and now we want to find the critical path. To find it, we sort every computation edge by the start time, then find groups of overlapping edges, and, in each group, choose the largest edge. In this approach, communication edges are redundant: we can replace these with an edge that connects the largest computation edges from the previous and the next group. If we remove communication edges, then our graph will not have cycles. In fact, the graph will have the form of n lists of computation edges: one list for each process. This means that the communication is implicit, we infer task dependencies only based on the starting and ending times of the task.

Why is topological sorting able to find bottlenecks when we do not consider communication between the processes? This is possible for programs that adhere to the Bulk Synchronous Parallelism (BSP) model [3]. This model assumes that the program consists of sequential supersteps that are internally parallel. Each superstep consists of a parallel computation stage and global synchronization stage during which the processes exchange and combine the results of the computations (Figure 1). In particular, in BSP, a parallel program executes as a series of parallel supersteps separated by barrier synchronizations on a set of processors. Every superstep consists of three sequential steps: (1) a local computation phase, where each process can carry out computation using local data values and send communication requests; (2) a global communication phase, where data are exchanged between processes in accordance with the requests made during the local computation phase; and (3) a barrier synchronization, which waits for all data transfers to be finished and makes the received data available for use in the next superstep. Most iterative programs adhere to this model: each iteration of such a program is a superstep. For topological sorting, this means that computations within each superstep overlap, and we can easily group these overlapping computation edges to find the largest one.

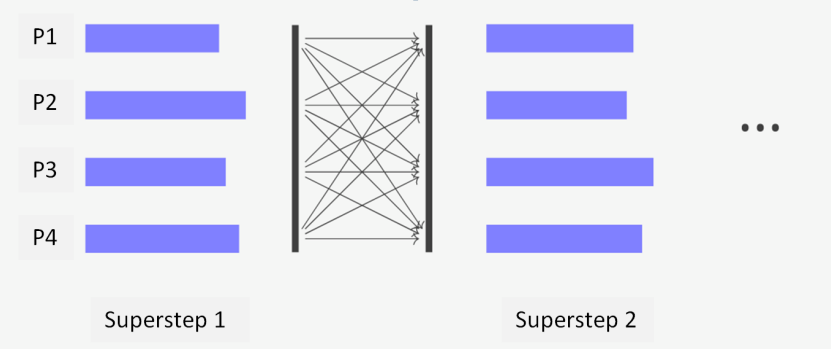

Figure 1. BSP model.

Even if the program does not adhere to the BSP model (i.e., computations overlap communication) topological sorting still works but shows the actual execution order of the tasks, not the order in which they are written in the source code and not the order in which

they are supposed to be executed considering MPI implementation. This is enough to at least obtain a hint about where the the bottlenecks are.

To summarize, there are two approaches to finding bottlenecks in the MPI program. One of them is based on finding the critical path in the graph, which represents task dependencies inferred from the program and the MPI calls it makes. The graph must not have cycles, and having the graph without cycles is only guaranteed if we implement this approach directly in the MPI library. Another approach is based on topological sorting, i.e., finding groups of overlapping computations and choosing the longest one inside the group. This approach does not use task dependencies inferred from the program but infers these dependencies from the actual execution order of the tasks. This approach works for any program but shows only actual bottlenecks related to computation without analyzing the communication patterns of the program.

In this paper, we analyze these two approaches. We present the corresponding algorithms for each approach; these are the algorithms for time synchronization that are essential for obtaining the correct MPI program profile in a real-world scenario. Then, we present the implementation of both approaches for the OpenMPI library and benchmark results for several real-world applications. We discuss the advantages and disadvantages of each approach and implementation and conclude with a few examples of bottlenecks that we found in real-world programs using one of these approaches.

The following parallel MPI-based benchmarks and software packages were selected to evaluate the proposed approach: NAS Parallel Benchmarks [4], CP2K [5], LAMMPS [6], OpenFOAM [7], and MiniFE [8]. These packages are well known in the community and represent examples of real-world applications often used in practical cases. In particular, the NAS Parallel Benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications in the original specification, later extended with unstructured adaptive meshes, parallel I/O, multi-zone applications, and computational grids. CP2K represents the domain of quantum chemistry and solid-state physics with atomistic simulations of solid-state, liquid, molecular, periodic, material, crystal, and biological systems. LAMMPS is a classical molecular dynamics (MD) code that models ensembles of particles in a liquid, solid, or gaseous state. It can model atomic, polymeric, biological, solid-state (metals, ceramics, oxides), granular, coarse-grained, or macroscopic systems using a variety of interatomic potentials (force fields) and boundary conditions. It can model 2D or 3D systems with sizes ranging from only a few particles up to billions. OpenFOAM is an open-source CFD software that has an extensive range of features to solve anything from complex fluid flows involving chemical reactions, turbulence, and heat transfer to acoustics, solid mechanics, and electromagnetics. MiniFE is an application for unstructured implicit finite element (FE) codes. It executes all FE phases: generation, assembly, and analysis. The physical domain is a 3D box simulated by a hexahedral element. More details on each of the packages and details of benchmark configuration will follow in Section 6.

The paper is structured as follows: Section 2 gives an overview of methods of critical path finding and approaches to this in the scope of MPI programs; Section 3 presents an overview of the proposed approach to collect information about MPI program activity and build the critical path; Section 4 considers several methods that can be used for critical path finding based on the Dijkstra algorithm, Delta-stepping, and topological sorting along with some enhancements to the methods' implementations; Section 5 describes methods and implementation for collecting information about the MPI program's execution; and Section 6 presents the experimental results of using the proposed critical path-finding method on a set of popular benchmarks (NAS Parallel Benchmarks, CP2K, OpenFOAM, LAMMPS, and MiniFE). Section 7 concludes the paper.

2. Related Work

Imbalance within the processes of a parallel MPI program is one of the key performance issues for the MPI workloads. One of the ways to find the root cause of the imbalance is

the analysis of the critical path of the MPI program, which is the longest execution path with an important feature: in general, there should be no wait states on the critical path. Optimizing program activities on the critical path can result in a reduction in the elapsed time of the overall workload, whereas optimizing activities that are not on the critical path might lead to an increase in the overall wait time.

In a general case, the critical path is the longest path between the first and the last milestones (a milestone is a node in the graph). For directed acyclic graphs, the longest path in the graph is the shortest path with negative weights. So, to find the critical path, we can use any algorithm for finding the shortest path in the graph.

Since the 1980s, researchers have developed methods for identifying the critical path on the execution graph of parallel programs in order to identify the underlying causes of imbalance problems. A Program Activity Graph (PAG) is a graph that represents the Program Activity while a program is being executed [9]. Program Activity (PA) is a discrete, time-bound job that does not overlap with other tasks. The PAs are arranged in a hierarchy; certain PAs must be accomplished before others may begin. A PAG is a directed, weighted, acyclic graph whose edges reflect the length of the PAs and whose vertices represent the start and finish of the PAs connected with certain communication events (such as send/receive) in a program.

Figure 2 shows an example of a typical PAG segment. The vertical axis shows the MPI rankings, while the horizontal axis is the time (from left to right). The PAG edges (Program Activities) are shown by green lines. The yellow lines are the communication edges that provide a precedence connection between the endings and beginnings of PAs (graph vertices denoted as v#). If there are many outbound communication edges, only the shortest one is indicated by a solid yellow line; the others are indicated by dotted yellow lines and are not regarded as graph edges. The PAG runs from right to left (from the point at which an application's execution ends to the point at which it starts).

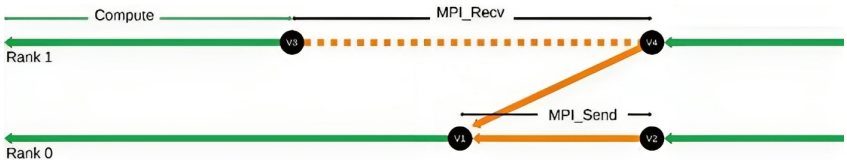

Figure 2. Part of program activity graph.

The critical path is the graph's longest path. Using shortest path methods with negative weights is the traditional method for locating the longest path in the directed acyclic graph [10]. The Dijkstra method, whose complexity increases linearly with the number of edges, is a well-known sequential algorithm for solving this issue for directed acyclic networks. It has several adaptations, including the parallel Delta-stepping method [11] and its variations [12,13] with acceptable performances on huge clusters and supercomputers, or the A* algorithm [14], which operates effectively under specific assumptions. Another method, which compares well to Delta-stepping in parallel implementation and demonstrates up to a 46% greater performance in single-thread implementation when compared to the Dijkstra algorithm, is based on the knowledge of incoming edges for every vertex [15]. With a parallel version of the longest route algorithm, based on [16], a first attempt to apply critical path analysis approaches to parallel programs was provided in [9]. Later, a suggested online algorithm for locating the critical path was proposed [17]. Their strategy is based on the notion that instrumentation messages may be combined with application messages to reduce analysis overhead by about 5% for synthetic benchmarks.

Critical path analysis began to focus on MPI workloads in the 2000s. An approach for locating the critical path based on MPI call tracking was presented in [2]. The scalability of this approach is in doubt because it is reliant on combining local subgraphs into a single graph with additional critical path analysis carried out on a single node, even if the authors report just 8% of overhead on 64 nodes (ranks) on a specific scenario. In order to speed up processing, graph reduction steps are introduced. The authors also

recommend recording the call stacks for the MPI functions connected to each graph node to make it easier to identify the underlying reasons for the imbalance. The paper lists several key constraints, one of which is the inability to manage lengthy applications due to the potentially enormous amount of data captured on every rank and the need to store local subgraphs. A method to determine the critical path for MPI workloads similar to [17], with just 1% overhead per 10,000 MPI calls per rank on synthetic benchmark, has been studied in [18]. While critical path analysis can show good results in certain situations (such as synthetic benchmarks, small scale, etc.), the outcomes on real-world workloads were less encouraging. In [19], researchers described a critical path-based analysis used with the Scalasca toolkit [20]. In order to exchange the data necessary for the performance analysis at each recorded synchronization point using a communication operation similar to the one used by the program originally, their method is based on recording all MPI calls and then replaying pertinent calls in both the backward and forward directions. The report only mentions one benchmark result, which results in a 148% increase in the overall critical route analysis overhead. A backward playback could potentially result in stability problems and hangs. A method for an automatic search of the optimization suggestions developed on top of GASPI [21] applications with the help of a critical path analysis of the task graph was presented in [22]. The extended task graph is backward traversed to determine the critical path, and the parallel version of the critical path analysis algorithm is not taken into account. It is assumed that the entire graph is present on a single node, but this is probably the first method in which memory access analysis and critical path data have been integrated. The concept of critical-path candidates (a collection of paths that could potentially make up the critical path) was introduced in [23] by using critical path analysis for performance modeling. However, to perform precise microarchitecture-independent modeling, instruction, and communication counts are employed instead of execution time as the key parameter for critical path building. The significance of critical path for the effective performance analysis of real-world HPC workloads is highlighted in one of the most recent papers [24]. To describe dispersed workloads, a new metric referred to as "Workflow Critical Path" is created. The technique, suggested in [13], has been used to create a cloud-based infrastructure for critical path analysis; however, its implementation's performance has not been fully analyzed.

In a recent paper [25], the authors propose a highly scalable MPI call replay-based solution for constructing the critical path with less than 5% of the collection overhead and less than 5% of the application's elapsed time spent on post-processing independently on the number of ranks. The approach has been tested on various real-world workloads and stays within performance targets even on a relatively high scale.

Despite the fact that the longest path algorithm is similar to that for the shortest path, there are specialized algorithms for the longest path:

- The parallel algorithm for undirected weighted graphs [26];
- The heuristic algorithm that uses pseudo-topological order [27];
- The genetic algorithm [28].

Before computing the critical path of the parallel MPI program, we have to record all MPI calls. The standard MPI distinguishes between point-to-point and collective calls and blocking and non-blocking calls. Each of the four combinations needs unique handling [29–32] when the records are read from the log and processed after the program ends. For example, for collective calls, we do not know the exact implementation of these calls, and we have to use some mathematical model to analyze these calls and compute the imbalance ratio.

When converting the log of MPI calls into the graph, we can remove some edges based on the logic of the calls. In [2], the following rules are used:

- The critical path cannot contain program edges leading to receive nodes that incur a blocking wait time.
- The critical path cannot contain communication edges, which do not lead to blocking wait times.

Applying these rules helps reduce the graph size and speeds up the critical path finding.

3. Approach Overview

Considering the analysis that was conducted in the previous section, we propose a high-level overview of the implementation of the MPI critical path-finding method which consists of the following stages:

1. Synchronize timestamps of all MPI processes.
   - We use a monotonic clock to save timestamps and synchronize clocks using `MPI_Allreduce` at the start of the program (in `MPI_init`).
2. Log relevant MPI operations with relevant arguments and start–end timestamps. Link local operations into a linear graph.
   - We log every MPI call that involves process communication (e.g., `MPI_Send`, `MPI_Recv`, `MPI_Reduce`, `MPI_Broadcast`, etc.) using the MPI profiling interface.
   - For each MPI call, we log the start timestamp, end timestamp, function name, and function arguments (except user data).
   - Each MPI process maintains its own log-in memory.
   - The log represents a linear graph: an edge connects the log record N to the log record $N + 1$. The weight of the edge equals the difference between the start time $N + 1$ and the end time N.
3. After `MPI_Finalize`, link corresponding local and remote operations (send/recv) to produce the final graph.
   - After the previous step, we have the log which is distributed across all MPI processes, and we now convert it to the graph. The local log contains only the edges for the vertices of the corresponding MPI process, and we need to create edges between the different MPI processes. (For each MPI call, we have a starting and ending vertex that is connected with the edge.)
   - To achieve this, each MPI process goes over all log records, and, for each MPI call, it requests information from the processes that were involved in this call. (It does not matter whether the receiver or sender obtains the information; we need to create the edges only in one MPI process not to produce duplicates.) The information is requested using the appropriate MPI calls (`MPI_Send`, `MPI_Recv`, `MPI_Reduce`, etc.), and the calls are different for each collective and point-to-point operation.
   - We can optimize the graph during conversion using techniques from [2]. For example, for `MPI_Barrier`, we need only one edge that connects the starting vertex for the last process to reach the barrier with the ending vertex for the last process to leave the barrier. For other blocking collective operations, the optimizations are similar.
4. Find the critical path using different algorithms and existing data distribution between nodes.
   - Now, we have the final graph that is composed of local graphs, which are stored in the corresponding MPI processes. The global graph is partitioned already, and all we need is to find the critical path.
   - For parallel processing, we need to map each vertex and edge of the graph to the rank of the MPI process that stores this edge or vertex. This is trivial to implement in the previous step by saving the rank of the target MPI process for each inter-process edge; all other vertices and edges are local to the node.
   - There are several algorithms for parallel graph search. The most common one is Delta-stepping (see Section 4). We make all weights negative and start the search from the vertex with the largest timestamp (which can also be determined in the previous step).
   - The algorithm can be optimized using the techniques in [12,33].

The implementation of the methods and algorithms for each stage is described in the following sections.

4. Methods and Algorithms for Finding Critical Path

4.1. Sequential Dijkstra

We implemented a sequential Dijkstra algorithm as the baseline for finding the critical path for the purpose of comparison with a parallel counterpart. Our implementation uses the classic algorithm described in [34]. In order to run this algorithm, we copy all the graph nodes and edges to the first MPI process and run the search there.

4.2. Sequential Delta-Stepping

The sequential Delta-stepping algorithm resembles the Dijkstra algorithm but uses buckets to store nodes. Each bucket stores the nodes for a particular range of tentative distances. The size of the range equals *delta* (hence, the name of the algorithm). This *delta* is called the "bucket width" or "step width". In each iteration of the algorithm, we remove the "light" nodes from the first non-empty bucket, update their distances, and redistribute them across buckets using the updated distance value. A "light" node is a node's tentative distance that is less than the minimum distance of the bucket, and a "heavy" node is a node's tentative distance that is greater than the maximum distance of the bucket. This process repeats until the bucket becomes empty. After that, we update the distances of heavy nodes and redistribute them across the buckets.

When $delta = 1$, we obtain the Dijkstra algorithm. When $delta = \infty$, we obtain the Bellman–Ford algorithm. The parallel version of Delta-stepping processes nodes in the bucket in parallel.

4.3. Parallel Dijkstra

We implemented parallel Delta-stepping for $delta = 1$. This algorithm is equivalent to the Dijkstra algorithm but processes all nodes and edges in parallel in a distributed memory and exchanges only a small number of nodes that need to be processed in the current iteration.

The algorithm works as follows.

1. We start with the nodes and edges distributed across the MPI processes: each process stores only the nodes that correspond to the MPI calls that were made by this process and all incoming edges of these nodes. The process that contains a particular node is determined using its global identifier and simple division.
2. Each process pushes the source node to its own per-process queue and the main loop begins.
3. For each iteration, each process extracts the next node from the queue and finds all incoming edges of this node. Then, we group the edges by the rank of the process that stores the source node of each edge. After that, each process sends each non-empty group to the corresponding rank.
4. After the communication, each process concatenates all of the groups of edges that were received from other processes. The resulting edges are scanned, and, if the new distance is smaller, then the distance is updated, and the source node of the edge is pushed into the queue with the new distance (distances are stored in the local hash table).
5. The iterations continue until all per-process queues are empty. If the queue becomes empty, then the corresponding process takes part in the collective operations but does not process nodes from the queue.
6. After the last iteration, the hash table that stores the path that was followed by the algorithm is gathered in the first MPI process. Then, the algorithm terminates.

The parallel Dijkstra algorithm does not give any improvements in the execution time, but it consumes a smaller amount of memory because we no longer need to gather all graph nodes and edges in a single MPI process.

4.4. Parallel Splits

Many MPI programs follow the Bulk Synchronous Model (BSP) of parallel execution [3], i.e., the program consists of sequential parts that are internally parallel, and after each such part comes global synchronization. These global synchronization points allow us to decompose the graph into multiple independent parts (Figure 3). We detect these points when the graph is generated from the log: we track each asynchronous MPI operation, and, when we encounter a global collective operation, we complete the following to determine whether it is a synchronization point:

1. If some asynchronous MPI operations have not been completed yet, then this is not a synchronization point.
2. If any edges of the collective MPI operation have a negative time, then this is not a synchronization point.
3. Otherwise, it is a global synchronization point, and we can split the graph at this point.

When the graph is fully generated, we use the source and the target node of each split to run the critical path-finding algorithm on each split individually. This can be performed in multiple ways:

1. Run a sequential algorithm on each split sequentially.
2. Run a parallel algorithm on each split sequentially.
3. Run a parallel algorithm on each split in parallel.
4. Run a sequential algorithm on each split in parallel.

We explored the first three options, and the first and the second options have not given us substantial performance advantages compared to running sequential or parallel algorithms without splits. The third option was implemented using parallel threads, which led to an oversubscription of system resources, but this option actually improved the performance. The proper implementation of the third option is challenging but possible: on each iteration of the main loop of the parallel Dijkstra algorithm, we can multiplex processing of multiple splits.

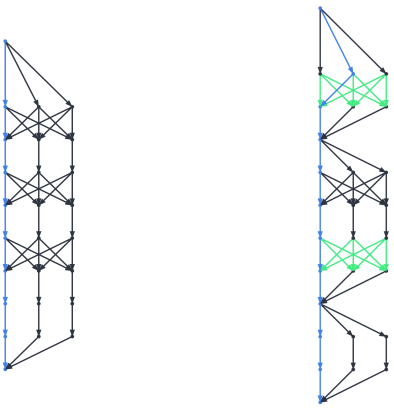

Figure 3. Parallel splits: graph with no splits (**left**); graph with splits (**right**).

4.5. Topological Sorting

The main idea of this algorithm is taken from topological sorting: "Sorting the vertices by the lengths of their longest incoming paths produces a topological ordering" [35]. Topological sorting is used to determine the order of execution of dependent tasks (Figure 4). Here, we have the inverse problem: we already executed each task and now want to find the critical path. To find it, we sort each program edge by the start time, then find groups of overlapping edges, and, in each group, choose the largest edge.

The main idea of inverse topological sorting is to decompose the edges into groups that come together (overlap). A straightforward implementation of this idea for any realistic MPI program will produce only one group that consists of all edges of the program run. The more practical approach is to group computation and communication edges separately

or even to skip communication edges as they are redundant for deriving a critical path. We used the latter approach.

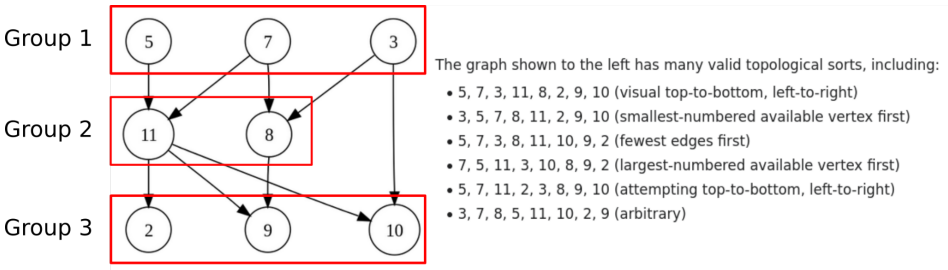

Figure 4. Example of topological sorting.

The sequential topological sorting-based algorithm is as follows:

1. Record intervals that denote MPI edges and program edges.
2. Gather all intervals in rank 0.
3. Sort them by start time (actually merge sorted arrays).
4. Find overlapping program edges. In each group, the longest edge belongs to the critical path (Figure 5).
5. Connect the longest program edges to each other.

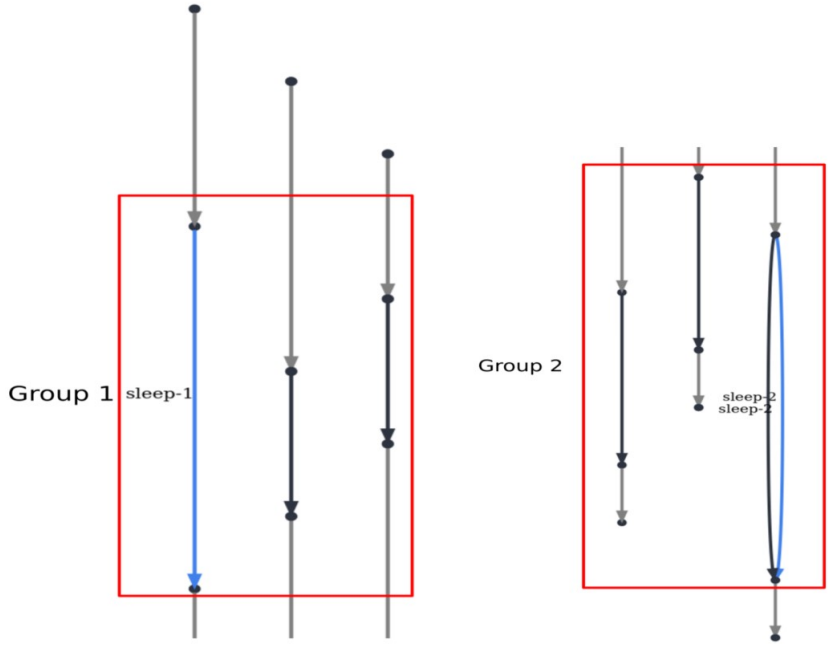

Figure 5. Groups of overlapping program edges.

The parallel algorithm is mostly the same, but we have to distribute (shuffle) the data between the MPI processes (Figure 6), execute the algorithm on each individual part, and then collect the results from all processes to the root process. The algorithm follows:

1. Record intervals that denote MPI edges and program edges.
2. Shuffle intervals between ranks. Each rank receives its own period of time.
3. Merge shuffled arrays in each rank.
4. Find overlapping program edges. In each group, the longest edge belongs to the critical path.
5. Connect the longest program edges to each other.
6. Gather all critical path segments in rank 0.

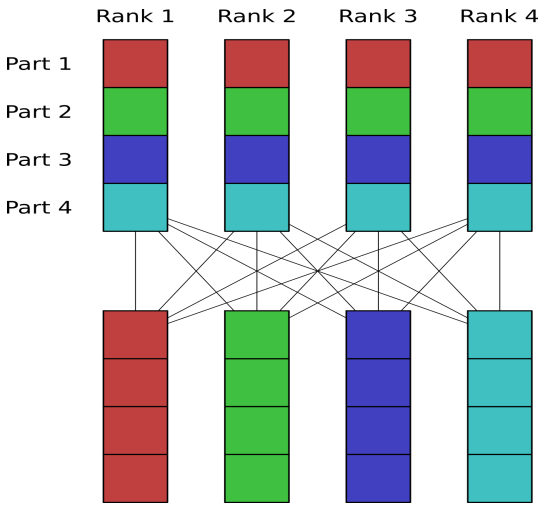

Figure 6. Shuffle.

The topological sorting algorithm has the following advantages.

- Linear time O(n).
- Reliable: works even without cross-process edges.
- No graph: no graph cycles are possible, nor are infinite program loops.

There are some disadvantages as well.

- Shuffle can be slow: need to transfer $(n-1)/n$ of the total size of the graph.
- Overlapping is good enough but not perfect: e.g., need to detect non-`MPI_COMM_WORLD` communicators.

The main disadvantage of this algorithm is that we have to somehow handle groups of interleaving intervals that spread two or more parts, i.e., the interval starts in part N and ends in part $>= N+1$. There are two possible algorithms here. First, we can search for the rank that contains the end of this interval. This would result in exchanging all of the edges that overlap this interval from the $currentrank+1$ up to the rank where the interval ends. This approach is simple but increases the amount of information sent over the network enormously: in the worst case, all MPI processes have to send all of their edges to all of the other processes, and the algorithm is no longer linear in terms of how much data are sent over the network—it is now quadratic. We call this algorithm *'topological-sorting-overlap'*, and this is the most accurate algorithm, although it requires sending an enormous amount of data over the network.

Second, we can search for the last interval that overlaps the interval that spreads several parts. This algorithm is more efficient than sending all intervals from the next part but is more sophisticated to implement. It is not clear whether it is faster considering the difficulty of the implementation. We call this algorithm *'topological-sorting-master-worker'*. In our benchmarks, we resorted to using a simple algorithm that does not handle overlap of the intervals across parts. This is due to the inefficiency of the *'overlap'* algorithm and the difficulty of the implementation of the *'master-worker'* algorithm. The technical implementation is also quite challenging. First of all, we have to sort edges in chronological order in order to find the groups of overlapping edges in linear time. However, each MPI process records only its own edges. This means that we have to somehow sort edges that are spread across the memory of different MPI processes.

This problem is common in big data; hence, we can borrow successful algorithms from this field. The most common approach here is to "shuffle" the data, i.e., assign each process an approximately equal chunk of the data and gather these data from whatever processes hold them. Then, all received data are sorted and groups of overlapping edges are found.

We implemented this algorithm in our program and then applied several performance optimizations. First of all, we have to balance the number of edges across all processes (Figure 7). The simplest approach here is to count them and spread them evenly. Initially,

we counted the time and spread by the time, but this led to a highly unbalanced number of edges. Second, we noticed that the shuffle performance deteriorates as we increase the number of MPI processes. We attributed this to the fact that, in our implementation, all processes send to process 0, then all processes send to process 1, and so on and so forth. We changed this logic to the following: all processes send to the right neighbor process (offset), then all processes send to the right neighbor of this neighbor process, and so on and so forth (Figure 8). This allowed us to decrease the variation in shuffle time when the number of processes is large (we expect that this optimization on some clusters may not increase performance at all). Finally, we noticed that waiting even for empty requests (`MPI_REQUEST_NULL`) takes a considerable amount of time (three seconds, in our case; see Figure 9). We believe that this problem can only be solved by substituting the MPI library with a more performant one (we use OpenMPI 4.1.1).

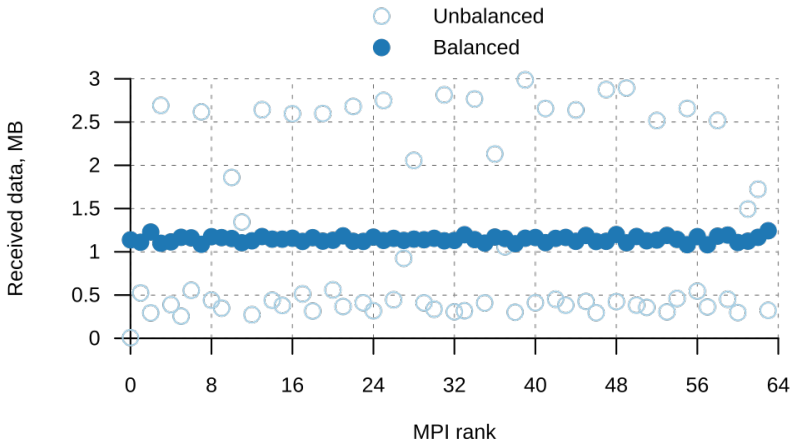

Figure 7. Unbalanced (initial) and balanced shuffle data distributions.

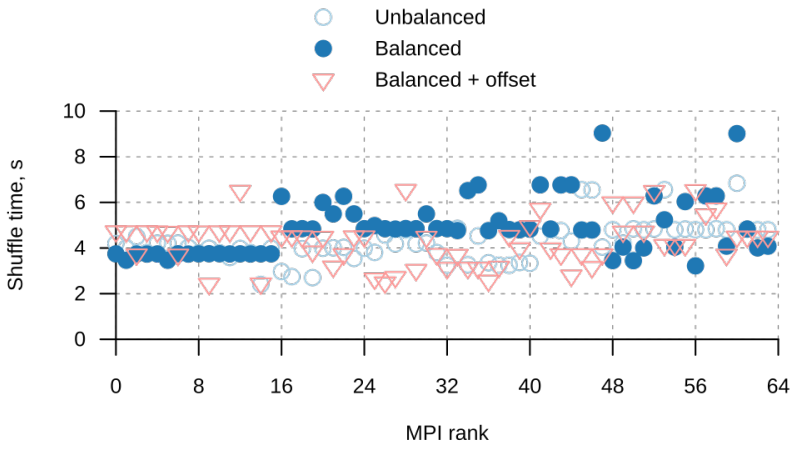

Figure 8. Shuffle time for unbalanced and balanced data distributions.

This approach is simple and well-known in the big data field and looks efficient; however, we still have to send all recorded edges over the network in the worst case. In order to optimize this, we compress the data, and, again, we use successful approaches from the big data field. The state-of-the-art compression approach is to convert the data representation into a columnar format (i.e., transpose the data) and then use suitable encoders and compressors. Then, we encode each integer column using run-length encoding (RLE), Delta-encoding, and variable-length integer (variant) encoding. After that, we use the Zstandard compressor to further compress the binary data. This approach allowed us to compress the data that we exchange between MPI processes during shuffling by 95% on average and to speed up shuffling by approximately 50% (Figures 10 and 11).

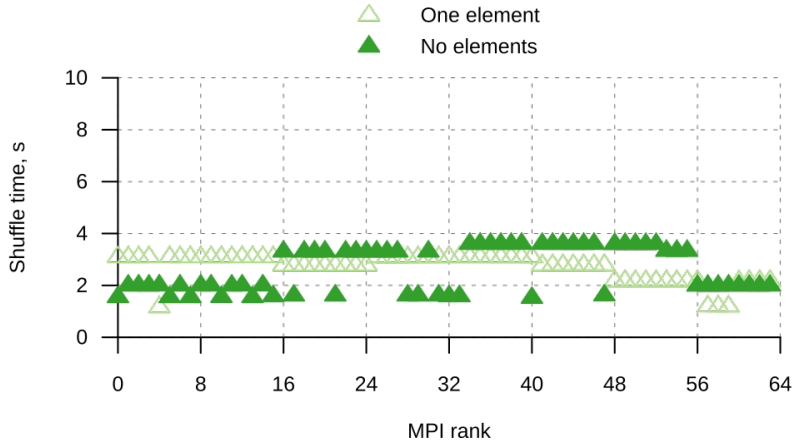

Figure 9. Shuffle time for empty and non-empty data requests.

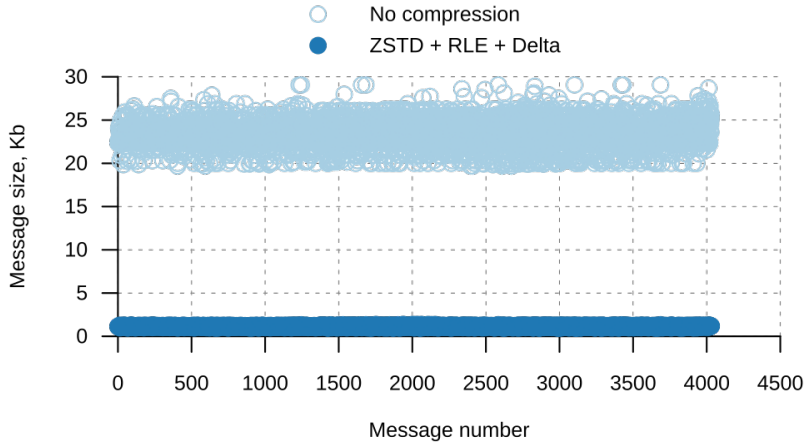

Figure 10. Results of applying compression: shuffle message size.

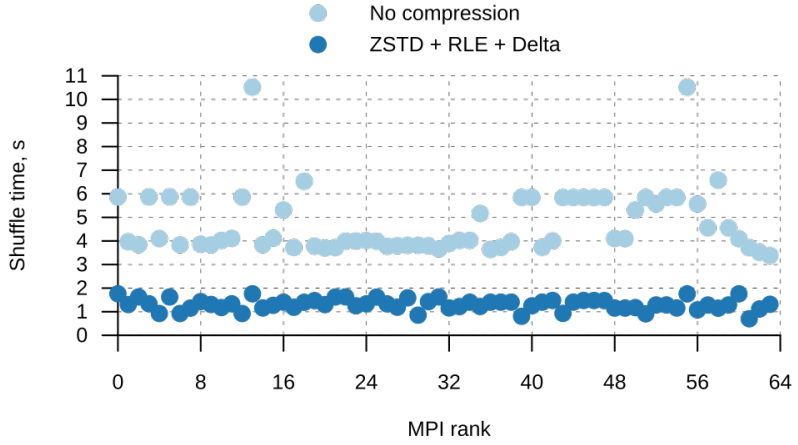

Figure 11. Results of applying compression: shuffle time.

Compared to graph-based algorithms, topological sorting offers several advantages. First of all, this algorithm cannot loop in an infinite graph cycle because there is no graph here. This advantage alone makes it the most robust algorithm that we have tried within the framework of this research work. Second, this algorithm is easy to parallelize because it is a well-known and well-studied algorithm in the big data field. Finally, since the data are sent in batches, we can use a columnar storage format and compression to speed up the algorithm.

One disadvantage of topological sorting is the linear dependence of running time on the number of MPI calls. This can become a problem because the more MPI processes your program has, the more MPI calls it will produce (Figure 12). This is clearly a scalability problem.

We have considered possible scalability issues and overheads related to both the number of MPI calls and the number of communicating processes in the tunable parameters of the proposed algorithm. These parameters help to adapt the overheads imposed by the algorithm and adjust the balance between fidelity and performance. In particular, the *threshold* parameter is introduced, which defines the minimal time between subsequent MPI calls, or the granularity unit size of the inter-communication computational operations. All of the communications and computations within the granularity unit are treated as a single composition block; computations of smaller sizes are not considered individually. This minimizes the overheads on counting data transmissions at the highest precision; however, it allows us to focus on meaningful parts of the program, neglecting other parts until they become meaningful, and, at the same time, allows us to establish a balance between the performance of the algorithms and the scale of the parallel execution. This solution is approximate, but it is good enough for practical applications and is in line with how real-world applications are profiled, diagnosed, and modified afterwards.

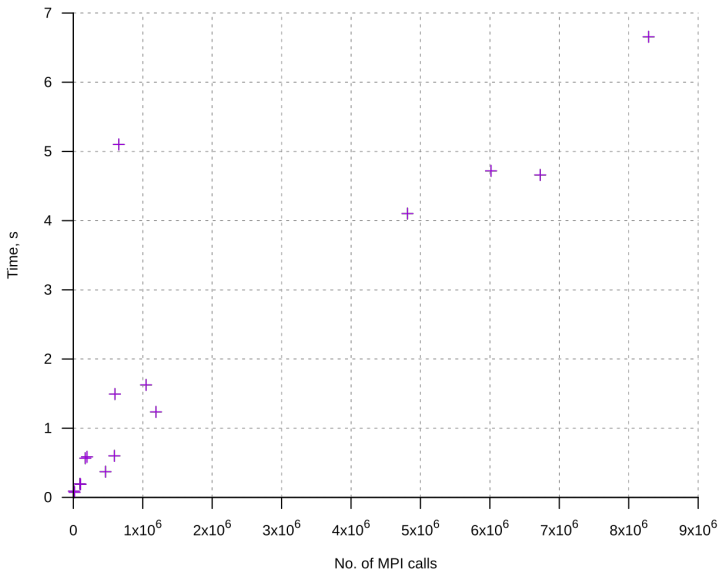

Figure 12. Topological sorting time dependence on the no. of MPI calls.

From a mathematical point of view, this algorithm finds the longest edge in each group of overlapping edges. If the program satisfies the BSP model, then groups of computation edges are interleaved with groups of communication edges (the communication step always follows the computation step). Therefore, the longest edge in each group of computation edges will indeed be part of the critical path, and groups of communication edges can be skipped because we can simply connect the end of the previous computation edge to the start of the next one. The imbalance can only occur if the parallel processes start/finish their computations for the current BSP superstep at different points in time. The only way to affect the topological sorting's correctness is to have unsynchronized clocks across the MPI processes. We addressed this shortcoming by implementing our own synchronization inside the program.

5. Profiling MPI Applications

5.1. Logging MPI Calls

In standard-compliant MPI libraries, every exported `MPI_XXX` function is in fact a weak symbol, and each such symbol has a strong counterpart named `PMPI_XXX`. When the symbol

is weak, we can replace it with a strong symbol possibly from another library during link time or during run time by dynamically pre-loading the library. The latter approach is used to implement MPI profilers—programs that wrap MPI calls and collect statistics from their arguments. Inside the wrapped MPI function, they usually call its PMPI counterpart but may perform any other calls when necessary.

We followed this approach to implement the *mpi-graph* library that is preloaded by the application using the *LD_PRELOAD* environment variable. In order to minimize the impact of our profiler on the execution time of the program, we perform a minimal number of PMPI calls inside each wrapped MPI function. Usually, it is only one call, but more than one MPI call is performed when we have a non-`MPI_COMM_WORLD` communicator and in some other corner cases.

For each MPI call, we record start time, end time, call number, and other relevant information, which are unique for each call (tag, source, destination, communicator, etc.). The log of all MPI calls is stored in the memory of each MPI process. We allocate memory directly from the operating system kernel using the *mmap*(2) system call and do not use pointers inside log record structures. This allows us to gradually move old log records to a disk if the log consumes too much memory.

When the program calls `MPI_Finalize`, we collect all of the logs inside the first MPI process, convert them to the graph, and find a critical path. When we switch to the parallel path-finding algorithm, we will no longer collect the logs but process them inside each process in parallel.

5.2. Converting the Log Records to the Graph

In order to transform the log to the graph, we repeat all of the MPI calls from the log, but, instead of the real data, we send the data that are needed by the receiving process to construct nodes and edges of the graph. The resulting graph is per process but may contain outgoing or incoming communication edges from other processes.

Blocking point-to-point call (send and receive) handling is straightforward. For non-blocking send and receive, we maintain a list of requests, and, when we encounter `MPI_Wait`, we find the corresponding request in this list and use it to complete the wait call. `MPI_Waitall` is simulated by calling `MPI_Wait` for each request in the corresponding array.

Blocking collective calls are handled by gathering the start and end times of the call for each rank in the first MPI process. Then, we compute the maximum start time and minimum end time for the statistics, broadcast all of the statistics and the start/end time to all processes in the communicator, and compute the per-rank statistics. Next, we generate all possible edges between the start and end nodes of each process (from the start of each process to the end of each process).

The per-rank statistics include the execution time (the time spent actually executing the collective MPI call), wait-before time (the time the process waited for other processes before executing the call), and wait-after time (the time the process waited after executing the call). These data are computed using the following formulas.

```
t_wait_before = t_start_max - t_start[i]
t_wait_after = t_end[i] - t_end_min
t_execution = t_end[i] - t_start[i] - t_wait_before - t_wait_after =
            = t_end_min - t_start_max
imbalance = (t_wait_before + t_wait_after) / t_execution
```

These formulas work very well if the MPI call has an implicit barrier at the end, but, if there is no such behavior, then the formulas may produce a negative wait time. Edges with negative times can be removed from the graph to speed up the critical path finding. The calls with implicit barriers can be found experimentally: in OpenMPI v4.1.1, we found that `MPI_Bcast`, `MPI_Barrier`, and `MPI_Reduceall` all have implicit barriers, but `MPI_Reduce` does not, for example.

Using these formulas, we computed the imbalance for each MPI call as the ratio of the total wait time to the total execution time. For MPI calls, this metric shows how much time (in percentages of execution time) we would save if the call was made well balanced (i.e., no wait time). The same metric is computed for each MPI process (but we sum all the numerators and denominators and add the execution time of the program edges to the denominator) and for the whole program (but we sum all the numerators and denominators across processes and divide them by the number of ranks). The corresponding formulas are presented below.

```
imbalance_call = (t_wait_before + t_wait_after) / t_execution
imbalance_process = sum(t_wait_before + t_wait_after for each call)
                  / (sum(t_execution for each call) + t_program_edges)
imbalance_program =
  sum(t_wait_before + t_wait_after for each call and process)
  / (sum(t_execution for each call and process)
     + sum(t_program_edges for each process))
```

5.3. Synthetic Benchmarks

We used mini-benchmarks to verify that the different MPI calls were properly converted to the graphs. In Figure 13, the left picture shows the graph for the program with a collective operation. In the right picture, we show the graph for the program—each process of which sends the data to the right process and receives from the left process using non-blocking point-to-point operations. The discovered critical path is depicted with the blue color.

The program output below shows the statistics and the imbalance metric computed by *mpi-graph* for the synthetic test program.

```
Per-process statistics:
rank  program_time  graph_time   total_time   total_execution_time
0  1.908003056   0.000988314  1.908991370  0.935452868
1  1.907246658   0.000208261  1.907454919  0.935452868
2  1.905164954   0.000214803  1.905379757  0.935452868
3  1.904170098   0.000349145  1.904519243  0.935452868

rank  total_wait_time   imbalance  num_calls  num_nodes   num_edges
0  0.018105627   0.019354932  3       8       13
1  1.344099347   1.436843472  3       6       13
2  1.335822343   1.427995347  3       6       13
3  1.332357843   1.424291793  3       6       13

Total statistics:
program_time=1.908642225
graph_time=0.000349145
total_time=1.908991370
total_execution_time=3.741811472
total_wait_time=4.030385160
imbalance=1.077121386
num_calls=12
num_nodes=26
num_edges=52
```

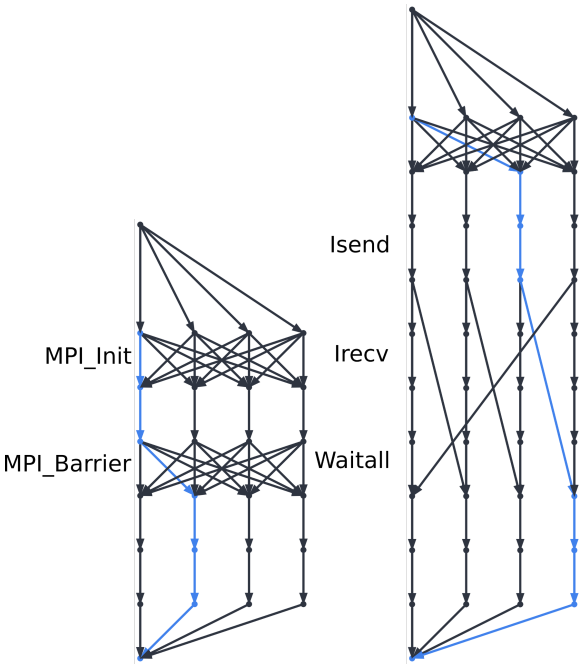

Figure 13. Program activity graphs: program with a collective operation (**left**); program with sending/receiving to/from neighbour processes using non-blocking point-to-point operations (**right**).

5.4. Time Synchronization

Accurate timestamps are needed for the correct computation of critical paths. Usually, this is achieved using external NTP servers, but we may not have control over them. We implemented similar functionality inside the *mpi-graph* tool. In order to synchronize the time, we perform a series of barriers and record timestamps before and after the barrier for each process. Then, we broadcast the timestamps of the first process and subtract each process's timestamp from the timestamp of the first process. If the magnitude of the resulting number is less than the time of the corresponding barrier, we nullify the number. Finally, we compute the median of all of the timestamp differences to determine the time offset of the current process.

We tested this algorithm using the *libfaketime* library that allowed us to change the time of one of the MPI processes. In our tests, the barrier time decreased with each subsequent call until it reached some stable value. The resulting graph and statistics were the same for the runs with and without the *libfaketime* library.

5.5. Visualization

In order to debug *mpi-graph* on small graphs we developed an interactive visualization tool. This tool reads the CSV logs produced by *mpi-graph* and shows the resulting graph in a web browser. The tool color codes the *wait-before*, *wait-after*, and execution time of each MPI call and computes the imbalance ratio for each collective MPI operation.

Examples of graphs produced by this tool are presented below in Section 6.6. The tool visualizes each call as a horizontal interval and each MPI process occupies its own row.

5.6. Load Imbalance Estimation

We have found several approaches to estimating the load imbalance of the MPI application: heuristic, analytic, and pattern-based.

5.6.1. Heuristic

In [36], the authors use a stacked graph approach. They sum the total time each process spends in a particular MPI call and then present the graph of these sums. The uneven "landscape" of the graph corresponds to the load imbalance (Figure 7.1 in [36]).

If we order the processes by the amount of time spent, we can see that the maximum time is spent by about 10 processes (out of 144), and the rest are waiting for them. Most of the processes are idle 40% of the time, and some are idle up to 80%.

5.6.2. Analytic

Different authors use various analytic formulas to compute the imbalance based on the max and average CPU times, number of processors used, distribution of CPU time across processors, communication time in point-to-point and collective operations, relation of communication and computing times, etc. [37–41].

5.6.3. Pattern-Based

In [38,39,42–45], the authors analyzed the performance of MPI applications using the following patterns.

- Late Sender. This property refers to the amount of time lost when the `MPI_Recv` call is sent before the corresponding `MPI_Send` is executed.
- Late Receiver. This property applies to the opposite case. `MPI_Send` is blocked until the corresponding receive operation is called. This can happen for several reasons. Either the default implementation works in synchronous mode, or the size of the message being sent exceeds the available buffer space, and the operation is blocked until the data are transmitted to the recipient. The behavior is similar to `MPI_SSend` waiting for the message to be delivered. The downtime is measured, and the sum of all downtime periods is returned as a severity value.
- Messages in Wrong Order. This property concerns the problem of sending messages out of order. For example, the sender can send messages in a certain order, and the recipient can expect them to arrive in the reverse order.
- Wait at Barrier. This property corresponds to the downtime caused by the load imbalance when the barrier is called. The idle time is calculated by comparing the execution time of the process for each `MPI_Barrier` call. To work correctly, the implementation of this property requires the participation of all processes in each call of the collective barrier operation. The final value is simply the sum of all measured downtime periods.

6. Benchmarking

6.1. NAS Parallel Benchmarks

The NAS Parallel Benchmarks [4] is a set of performance tests aimed at testing the capabilities of highly parallel supercomputers. They were developed in the early 1990s as part of the NASA Numerical Aerodynamic Simulation Program and are supported by the NASA Advanced Supercomputing (NAS) Division located at NASA Ames Research Center.

There are several classes for each benchmark that are related to the problem size.

- Class S: small for quick test purposes;
- Class W: workstation size (a 1990s workstation; now likely too small)
- Classes A, B, and C: standard test problems; $4\times$ size increase going from one class to the next;
- Classes D, E, and F: large test problems; $16\times$ size increase from each of the previous classes.

Different benchmarks have different requirements for the number of processes, as listed below:

- BT, SP—a square number of processes (1, 4, 9, ...);
- LU—2D (n1 $\times$ n2) process grid where n1/2 <= n2 <= n1;
- CG, FT, IS, MG—a power-of-two number of processes (1, 2, 4, ...);
- EP, DT—no special requirement.

The required process count is checked at runtime. By default, a run will abort if the process count requirement is not met. For IS and DT, there is a minimal process count for a given class of problem size. Table 1 presents the description of each benchmark including the problem sizes and parameters of Class C, which were used in the experiments.

We tested the performance of the *mpi-graph* tool (applying different versions of the critical path algorithms) using the NAS Parallel Benchmarks of Class C run on 4, 16, 32, and 64 processors. We measured the following:

- total run time with MPI call recording;
- total run time with MPI call recording + critical path finding;
- total run time without *mpi-graph*.

The results for the sequential Dijkstra algorithm are presented in Figure 14.

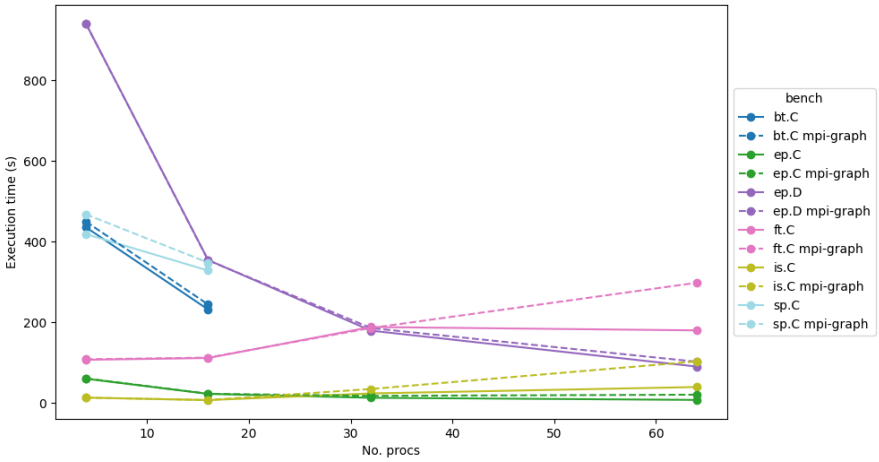

Figure 14. NAS Parallel Benchmarks test results (sequential Dijkstra).

Figure 14 shows that the *mpi-graph* tool incurs some overhead, the magnitude of which depends on the particular benchmark. Further analysis revealed that most of the overhead is caused by critical path finding rather than log collection.

Figures 15 and 16 present the bar charts of the NAS Benchmarks' execution time on various numbers of processors without/with *mpi-graph* usage (with the parallel Dijkstra algorithm). Figure 17 shows the relative overhead of using the *mpi-graph* tool (MPI call recording + critical path finding with the parallel Dijkstra). In this plot, the overhead is measured in the number of times the execution of the application with *mpi-graph* exceeds the execution of the original program. In particular, we can see that the *ep.D* benchmark runtime is more than 2.5 times longer when *mpi-graph* is used with the parallel Dijkstra algorithm, which demonstrates that this approach is not scalable enough.

Our experiments with classical algorithms (sequential and parallel Dijkstra, Delta-stepping) revealed noticeable overheads for some benchmark cases. To deal with this further, we concentrated on the topological sorting algorithm, which showed much better results in the same cases. In addition, the compression technique described in the same section allowed for further improvement to the results (see Table 2). In this table, the relative overhead corresponds to the quotient of difference between *mpi-graph* and non-*mpi-graph* runtimes divided by the *mpi-graph* runtime. In some cases, however, compression can lead to slowdown (e.g., in the case in which large data volumes are sent over fast communications and slow computing nodes), so the effect of using compression must be evaluated in each particular case; compression can be switched on or off in the configuration file.

Table 1. NAS Parallel Benchmarks description with problem sizes and parameters of Class C.

Benchmark	Name	Description	Problem Sizes and Parameters (Class C)
MG	Multi-grid on a sequence of meshes, long- and short-distance communication, memory intensive	Approximation of the solution of a three-dimensional discrete Poisson equation using the V-cycle multigrid method.	grid size: 512 × 512 × 512 no. of iterations: 20
CG	Conjugate gradient, irregular memory access and communication	Approximation to the smallest eigenvalue of a large sparse, symmetric positive-definite matrix using inverse iteration together with the conjugate gradient method as a subroutine for solving linear systems of algebraic equations.	no. of rows: 150,000 no. of nonzeros: 15 no. of iterations: 75 eigenvalue shift: 110
FT	Discrete 3D fast Fourier Transform all-to-all communication	Solving a three-dimensional partial differential equation using the Fast Fourier Transform (FFT).	grid size: 512 × 512 × 512 no. of iterations: 20
IS	Integer sort, random memory access	Sorting small integers using pocket sorting.	no. of keys: 2^{27} key max. value: 2^{23}
EP	Embarrassingly Parallel	Generation of independent normally distributed random variables using Marsaglia polar method.	no. of random-number pairs: 2^{32}
BT	Block tri-diagonal solver	Solves a synthetic system of nonlinear diffs. partial differential equations (a 3-dimensional system of Navier–Stokes equations for a compressible liquid or gas) using a three-block tridiagonal scheme with the method of variable directions (BT), a scalar five-diagonal scheme (SP), and a method of symmetric sequential upper relaxation (SSOR algorithm, LU problem).	grid size: 162 × 162 × 162 no. of iterations: 200 time step: 0.0001
SP	Scalar penta-diagonal solver	Solution of the heat equation taking into account diffusion and convection in a cube. The heat source is mobile, the grid is irregular, and changes every 5 steps.	grid size: 162 × 162 × 162 no. of iterations: 400 time step: 0.00067
LU	Lower-upper Gauss–Seidel solver	Same problem as SP, but the method is different.	grid size: 162 × 162 × 162 no. of iterations: 250 time step: 2.0

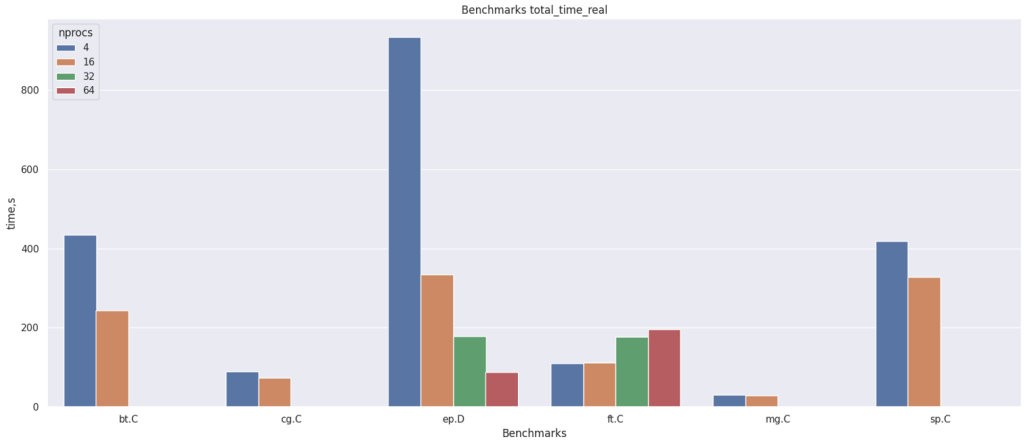

Figure 15. NAS Benchmarks execution time: real time (no critical path data collection).

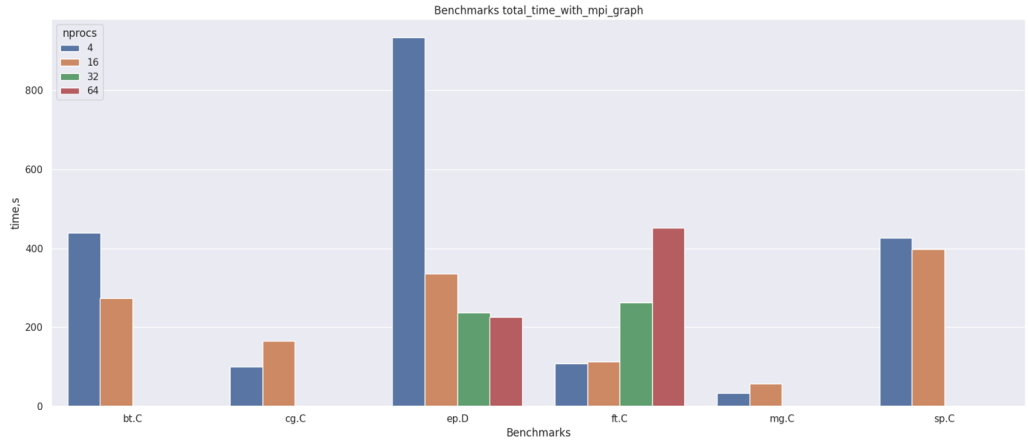

Figure 16. NAS Benchmarks execution time: with mpi-graph (parallel Dijkstra algorithm).

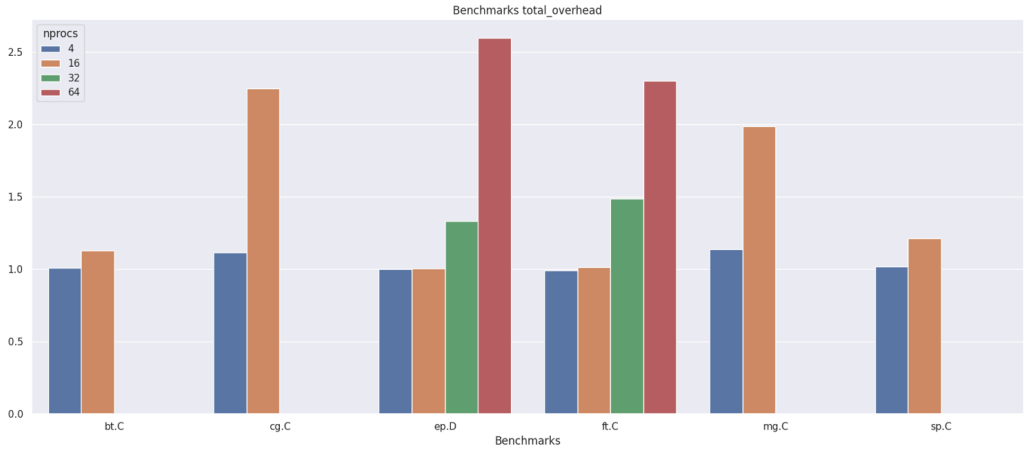

Figure 17. Relative overhead of *mpi-graph* (parallel Dijkstra).

Table 2. Comparison of versions without/with compression of shuffle data using topological sorting algorithm.

Benchmark	Nprocesses	Nnodes	Tasks-per-Node	Rel. Overhead (No Compression)	Rel. Overhead (Compression)
ep.C	4	1	4	0.02	0.01
ep.C	16	2	8	0	0.01
ep.C	32	4	8	0.09	0.02
ep.C	64	8	8	0.44	0.05
is.C	4	1	4	0	0
is.C	16	2	8	0.02	0.001
is.C	32	4	8	0.19	0.06
is.C	64	8	8	0.06	0
lu.C	4	1	4	0	0.01
lu.C	16	2	8	0.06	0.02
lu.C	32	4	8	0.10	0.04
lu.C	64	8	8	0.16	0.09
mg.C	4	1	4	0	0.01
mg.C	16	2	8	0.04	0
mg.C	32	4	8	0.14	0.02
mg.C	64	8	8	0.25	0.06

The detailed benchmark results obtained for different NPB cases with the topological sorting algorithm follow in Tables 3–8 and the benchmark parameters in Table 9:

Table 3. bt.C.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
2	2	432.61	0.024	0.01%
4	4	150.25	0.21	0.14%
8	8	100.78	1.52	1.49%

Table 4. ep.C.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
1	8	31.51	0.001	0.00%
2	8	15.89	0.010	0.06%
4	8	8.07	0.069	0.85%
8	8	5.19	0.059	1.14%

Table 5. is.C.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
1	8	7.18	0.16	2.23%
2	8	16.96	0.17	1.03%
4	8	18.91	0.11	0.57%
8	8	26.60	0.13	0.48%

Table 6. mg.C.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
1	8	21.56	0.03	0.14%
2	8	14.82	0.13	0.89%
4	8	16.81	0.37	2.14%
8	8	8.43	0.83	8.94%

Table 7. ft.C.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
2	2	198.51	0.09	0.05%
4	4	113.77	0.38	0.33%
8	8	105.89	0.24	0.23%

Table 8. lu.C.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
2	2	260.86	0.51	0.19%
4	4	92.11	1.66	1.77%
8	8	58.76	5.34	8.33%

Table 9. NPB benchmark parameters.

Test Case	Compression	Threshold, s
bt.C	no	0
ep.C	no	0
is.C	no	0
mg.C	no	0
ft.C	no	0
lu.C	yes (min. level)	0

6.2. CP2K

CP2K is a quantum chemistry and solid-state physics software package that can perform atomistic modeling of solid-state, liquid, molecular, periodic, material, crystalline, and biological systems [5]. CP2K provides a common framework for various modeling methods, such as DFT, using mixed Gaussian and plane waves and the GPW and GAPW approaches. Supported theory levels include DFT, LDA, GS, MP2, RPA, semi-empirical methods (AM1, PM3, PM6, RM1, MNDO, etc.), and classical force fields (AMBER, CHARMM, etc.). CP2K can perform molecular dynamics, metadynamics, Monte Carlo, Ehrenfest dynamics, vibrational analysis, spectroscopy at the core level, energy minimization, and transition state optimization using the NEB or dimer method.

Fayalite-FIST is a short molecular dynamics run for 1000 time steps in an NPT ensemble at 300 K (in the isothermal–isobaric ensemble, the amount of substance (N), pressure (P), and temperature (T) are conserved). It consists of 28,000 atoms—a 103 supercell with 28 atoms of iron silicate (Fe_2SiO_4, also known as Fayalite) per unit cell. The simulation employs a classical potential with long-range electrostatics using Smoothed Particle Mesh Ewald (SPME) summation.

CP2K comes with different versions of the executable listed in Table 10.

Table 10. lu.C.

Acronym	Meaning
ssmp	Single process + symmetric multiprocessor (OpenMP)
sdbg	ssmp + debug settings
psmp	Parallel (MPI) + symmetric multiprocessor (OpenMP)
popt	psmp + optimized
pdbg	psmp + debug settings

We used cp2k.popt in our benchmarks. The benchmark results for topological sorting are shown in Table 11 with benchmark parameters in Table 12.

Table 11. CP2K benchmark results.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
2	2	12.24	0.011	0.09%
4	4	24.74	0.027	0.11%
8	8	121.92	0.371	0.30%
10	8	142.55	4.162	2.84%

Table 12. CP2K benchmark parameters.

Test Case	No. of Iterations	Compression	Threshold, s
fayalite	10	yes (min level)	0

6.3. OpenFOAM

OpenFOAM (Open-Source Field Operation And Manipulation CFD ToolBox) is an open integrated platform for the numerical simulation of continuum mechanics problems [7]. OpenFOAM is both a C++ library and a set of applications that were built using the library. The applications are divided into two categories:

- There are solvers, each of which is designed to solve a specific problem of continuum mechanics. Each solver has at least one tutorial that shows its use.
- There are utilities designed to perform tasks related to data manipulation. OpenFOAM comes with pre-/post-processing environments, each of which has its own utilities.

We ran the performance benchmarks for the *pitzDailyExptInlet* case from the OpenFOAM tutorial based on the *simpleFoam* solver, which is a steady, incompressible flow solver based on Spalding and Patankar's SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm. We changed only the number of iterations and the number of

processors for each benchmark; other parameters were left unchanged. The configuration of the benchmark can be found in the original repository [46]. The benchmark results are presented in Table 13.

Table 13. OpenFOAM benchmark results.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
2	2	70.56	1.47	2.05%
4	4	104.35	2.98	2.79%
8	8	219.03	9.62	4.21%
10	8	230.47	10.02	4.18%

Common benchmark parameters are shown in Table 14.

Table 14. OpenFOAM benchmark parameters.

Test Case	No. of Iterations	Compression	Threshold, s
pitzDailyExptInlet	1000	yes (min level)	0.001

6.4. LAMMPS

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) is a free package for classical molecular dynamics [6]. The package can be used for large calculations (up to tens of millions of atoms). To work on multiprocessor systems, the MPI interface is used.

The LAMMPS package has built-in benchmarks that can be used, in particular, for independent testing. In our benchmarks, we used the LJ case: atomic fluid, the Lennard–Jones potential with a 2.5 sigma cutoff (55 neighbors per atom), NVE integration, 32,000 atoms, and 100 timesteps. More details on the LAMMPS benchmarks can be found in [47]. The results of the benchmarks are shown in Table 15 and the benchmark parameters in Table 16.

Table 15. LAMMPS benchmark results.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
2	2	125.92	0.28	0.22%
4	4	63.52	0.49	0.76%
8	8	35.31	1.53	4.16%
10	8	49.33	2.64	5.08%

Table 16. LAMMPS benchmark parameters.

Test Case	No. of Iterations	Compression	Threshold, s
in.lj	10,000	yes (min level)	0.001

6.5. MiniFE

MiniFE is a scientific mini-application developed at Sandia National Laboratory (USA), which includes algorithms for unstructured implicit finite elements or solvers [8]. The program simulates the stationary thermal conductivity of the beam. The numerical approach follows a simple unconditional conjugate gradient (CG) algorithm, which implies sparse matrix-vector products during the CG iteration. These data are stored in the local memory of the participating nodes. The implicitness of the numerical method and the locality of the data lead to a scenario with intensive data exchange as the size of the problem increases. The size of the physical brick (discretized by a given number of grid cells per spatial dimension in the problem statement) serves to control the properties of weak scaling for a given number of participating cores in the simulation.

The results of the benchmarks are shown in Table 17 and the benchmark parameters in Table 18.

Table 17. MiniFE benchmark results.

No. of Nodes	No. of Processes per Node	Program Time, s	MPI-Graph Time, s	Overhead, %
2	2	137.73	0.05	0.03%
4	4	52.43	0.22	0.42%
8	8	39.78	1.00	2.44%
10	8	36.38	2.35	6.06%

Table 18. MiniFE benchmark parameters.

Test Case	Compression	Threshold, s
nx = 300 ny = 300 nz = 300	yes (max level)	0

6.6. Case Studies

We used our software to find bottlenecks and imbalances in a number of software packages. We visualized every bottleneck using our software. Figures 18–20 show examples of found bottlenecks.

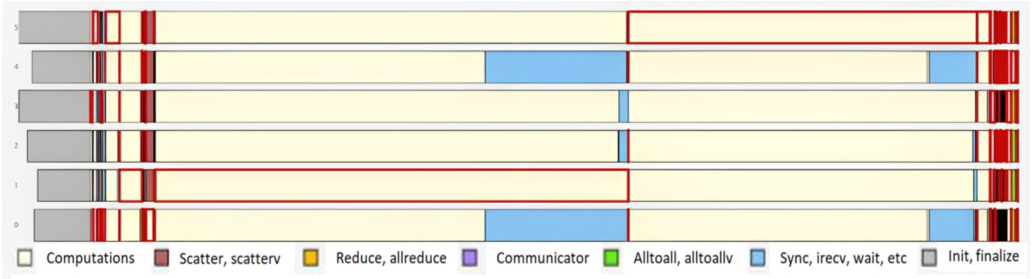

Figure 18. OpenFOAM, snappyHexMesh, motorbike test case.

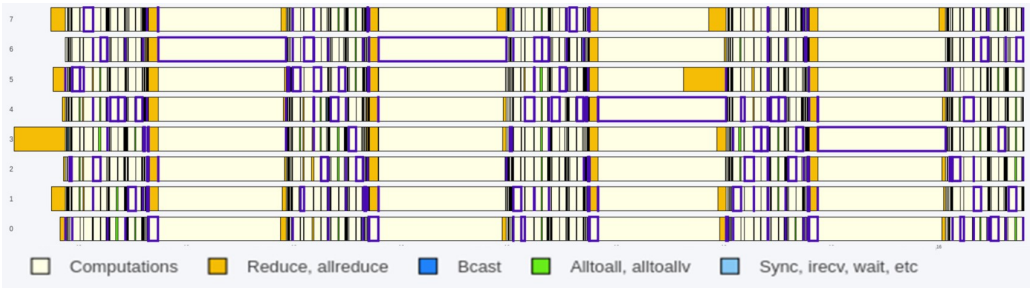

Figure 19. CP2K, Fayalite test case.

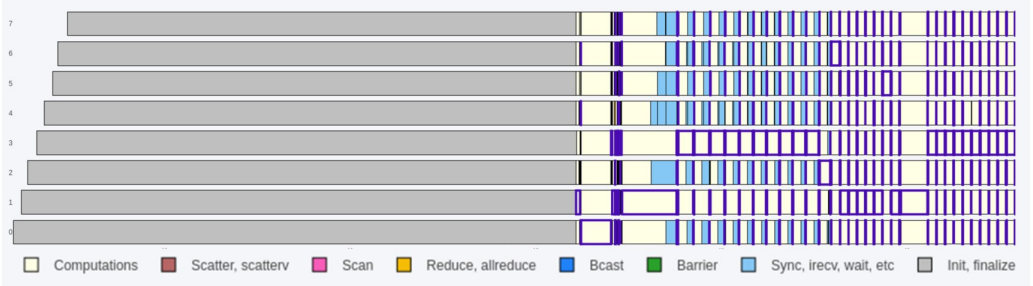

Figure 20. LAMMPS, lj test case.

7. Conclusions

The problem of finding bottlenecks in parallel workloads is important to consider to provide high performance and efficiency of the execution and becomes more challenging with the increasing scale of parallelization. One of the ways to discover bottlenecks is to analyze the critical path of the parallel program: the longest execution path in the program activity graph. There are a number of methods for finding the critical path; however, most of them suffer a performance drop on a large scale.

In this paper, we analyzed several methods of critical path finding based on the classical Dijkstra and Delta-stepping algorithms along with an algorithm based on topological sorting. While being slightly less generic compared to the classical algorithms, topological sorting can provide significantly higher performance at the expense of adhering to the Bulk Synchronous Parallelism (BSP) model and neglecting micro-communications that do not affect the overall view at the bottlenecks. At the same time, this approach removes some of the limitations of the classical methods, such as the acyclicity of the directed program activity graph. Even being designed with the BSP model in mind, this approach works for any program, showing actual bottlenecks related to computation without analyzing details of the communication patterns of the program.

We presented the corresponding algorithms for each approach including additional enhancements for increasing performance (parallel splits, shuffle, load balancing, compression, and thresholding). We presented the implementation of the approaches for the OpenMPI library and results for several benchmark applications: the NAS Parallel Benchmarks, CP2K, OpenFOAM, LAMMPS, and MiniFE. We presented the advantages and disadvantages of each approach and implementation and concluded with a few examples of bottlenecks that we found in real-world programs using the presented approach.

Author Contributions: Conceptualization, V.K., I.G., A.S. and V.S.; methodology, V.K. and I.G.; software, I.G., A.G., M.M., I.P. and D.T.; validation, V.K., A.S. and V.S.; formal analysis, I.G.; investigation, V.K., I.G., A.G., M.M., I.P. and D.T.; resources, V.K., A.S. and V.S.; writing—original draft preparation, I.G. and V.K.; writing—review and editing, V.K., A.S. and V.S.; visualization, M.M., D.T., V.K. and I.G.; supervision, V.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially supported by Saint Petersburg State University, project ID 86066736, 94062114.

Data Availability Statement: Data sharing not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. MPI Forum. Available online: https://www.mpi-forum.org/ (accessed on 18 October 2023).
2. Schulz, M. Extracting Critical Path Graphs from MPI Applications. In Proceedings of the 2005 IEEE International Conference on Cluster Computing, Burlington, MA, USA, 27–30 September 2005. [CrossRef]
3. Valiant, L.G. A bridging model for parallel computation. *Commun. ACM* **1990**, *33*, 103–111. [CrossRef]
4. Bailey, D.H.; Schreiber, R.S.; Simon, H.D.; Venkatakrishnan, V.; Weeratunga, S.K.; Barszcz, E.; Barton, J.T.; Browning, D.S.; Carter, R.L.; Dagum, L.; et al. The NAS parallel benchmarks—Summary and preliminary results. In Proceedings of the 1991 ACM/IEEE Conference on Supercomputing—Supercomputing'91, Albuquerque, NM, USA, 18–22 November 1991; ACM Press: New York, NY, USA, 1991. [CrossRef]
5. Kühne, T.D.; Iannuzzi, M.; Del Ben, M.; Rybkin, V.V.; Seewald, P.; Stein, F.; Laino, T.; Khaliullin, R.Z.; Schütt, O.; Schiffmann, F.; et al. CP2K: An electronic structure and molecular dynamics software package—Quickstep: Efficient and accurate electronic structure calculations. *J. Chem. Phys.* **2020**, *152*, 194103. [CrossRef] [PubMed]
6. Thompson, A.P.; Aktulga, H.M.; Berger, R.; Bolintineanu, D.S.; Brown, W.M.; Crozier, P.S.; Veld, P.T.; Kohlmeyer, A.; Moore, S.G.; Nguyen, T.D.; et al. LAMMPS—A flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Comp. Phys. Comm.* **2022**, *271*, 10817. [CrossRef]
7. Chen, G.; Xiong, Q.; Morris, P.; Paterson, E.; Sergeev, A.; Wang, Y. OpenFOAM for computational fluid dynamics. *N. Am. Math. Soc.* **2014**, *61*, 354–363. [CrossRef]
8. Lin, P.T.; Heroux, M.A.; Barrett, R.F.; Williams, A. Assessing a mini-application as a performance proxy for a finite element method engineering application. *Concurr. Comput. Pract. Exp.* **2015**, *27*, 5374–5389. [CrossRef]

9. Yang, C.Q.; Miller, B.P. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, CA, 13–17 June 1988; pp. 366–373. Available online: https://ftp.cs.wisc.edu/paradyn/papers/CritPath-ICDCS1988.pdf (accessed on 18 October 2023).
10. Uehara, R.; Uno, Y. Efficient Algorithms for the Longest Path Problem. In *Algorithms and Computation, Proceedings of the 15th International Symposium, ISAAC 2004, Hong Kong, China, 20–22 December 2004*; Fleischer, R., Trippen, G., Eds.; Springer: Berlin/Heidelberg, Germay, 2005; Volume 15, pp. 871–883.
11. Meyer, U.; Sanders, P. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms* **2003**, *49*, 114–152. [CrossRef]
12. Chakaravarthy, V.T.; Checconi, F.; Murali, P.; Petrini, F.; Sabharwal, Y. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 2031–2045. [CrossRef]
13. Kranjvcevi'c, M.; Palossi, D.; Pintarelli, S. Parallel Delta-Stepping Algorithm for Shared Memory Architectures. *arXiv* **2016**, arXiv:1604.02113.
14. Zeng, W.; Church, R.L. Finding shortest paths on real road networks: The case for A*. *Int. J. Geogr. Inf. Sci.* **2009**, *23*, 531–543. [CrossRef]
15. Alves, D.R.; Krishnakumar, M.S.; Garg, V.K. Efficient Parallel Shortest Path Algorithms. In Proceedings of the 2020 19th International Symposium on Parallel and Distributed Computing (ISPDC), Warsaw, Poland, 5–8 July 2020. [CrossRef]
16. Chandy, K.M.; Misra, J. Distributed computation on graphs: Shortest path algorithms. *Commun. ACM* **1982**, *25*, 833–837. [CrossRef]
17. Hollingsworth, J.K. Critical Path Profiling of Message Passing and Shared-Memory Programs. *IEEE Trans. Parallel Distrib. Syst.* **1998**, *9*, 1029–1040. [CrossRef]
18. Dooley, I.; Arya, A.; Kalé, L.V. Detecting and using critical paths at runtime in message driven parallel programs. In Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW), Atlanta, GA, USA, 19–23 April 2010; pp. 1–8.
19. Böhme, D.; Wolf, F.A.; Geimer, M. Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, China, 21–25 May 2012; pp. 2538–2541.
20. Geimer, M.; Wolf, F.; Wylie, B.; Ábrahám, E.; Becker, D.; Mohr, B. The SCALASCA performance toolset architecture. *Concurr. Comput. Pract. Exp.* **2010**, *22*, 702–719. [CrossRef]
21. Grünewald, D.; Simmendinger, C. The GASPI API specification and its implementation GPI 2.0. In Proceedings of the 7th International Conference on PGAS Programming Models, Edinburgh, UK, 3–4 October 2013.
22. Herold, C.; Krzikalla, O.; Knüpfer, A. Optimizing One-Sided Communication of Parallel Applications Using Critical Path Methods. In Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Orlando, FL, USA, 29 May–2 June 2017; pp. 567–576. [CrossRef]
23. Chen, J.; Clapp, R.M. Critical-path candidates: Scalable performance modeling for MPI workloads. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; pp. 1–10.
24. Nguyen, D.D.; Karavanic, K.L. Workflow Critical Path: A data-oriented critical path metric for Holistic HPC Workflows. *BenchCounc. Trans. Benchmarks Stand. Eval.* **2021**, *1*, 100001. [CrossRef]
25. Shatalin, A.; Slobodskoy, V.; Fatin, M. Root Causing MPI Workloads Imbalance Issues via Scalable MPI Critical Path Analysis. In *Supercomputing: 8th Russian Supercomputing Days, RuSCDays 2022, Moscow, Russia, September 26–27, 2022, Revised Selected Papers;* Voevodin, V., Sobolev, S., Yakobovskiy, M., Shagaliev, R., Eds.; Springer International Publishing: Cham, Switzerland, 2022; pp. 501–521.
26. Fieger, K.; Balyo, T.; Schulz, C.; Schreiber, D. Finding optimal longest paths by dynamic programming in parallel. In Proceedings of the Twelfth Annual Symposium on Combinatorial Search, Napa, CA, USA, 16–17 July 2019.
27. Raggi, M. Finding long simple paths in a weighted digraph using pseudo-topological orderings. *arXiv* **2016**, arXiv:1609.07450.
28. Portugal, D.; Antunes, C.H.; Rocha, R. A study of genetic algorithms for approximating the longest path in generic graphs. In Proceedings of the 2010 IEEE International Conference on Systems, Man and Cybernetics, Istanbul, Turkey, 10–13 October 2010; pp. 2539–2544. [CrossRef]
29. Pjesivac-Grbovic, J.; Angskun, T.; Bosilca, G.; Fagg, G.; Gabriel, E.; Dongarra, J. Performance Analysis of MPI Collective Operations. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, Denver, CO, USA, 4–8 April 2005. [CrossRef]
30. Pješivac-Grbović, J.; Angskun, T.; Bosilca, G.; Fagg, G.E.; Gabriel, E.; Dongarra, J.J. Performance analysis of MPI collective operations. *Clust. Comput.* **2007**, *10*, 127–143. [CrossRef]
31. Saif, T.; Parashar, M. Understanding the Behavior and Performance of Non-blocking Communications in MPI. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 173–182. [CrossRef]
32. Hoefler, T.; Lumsdaine, A.; Rehm, W. Implementation and performance analysis of non-blocking collective operations for MPI. In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing—SC'07, Reno, NV, USA, 10–16 November 2007; ACM Press: New York, NY, USA, 2007. [CrossRef]

33. Ueno, K.; Suzumura, T. Highly scalable graph search for the Graph500 benchmark. In Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing—HPDC'12, Online, 21–25 June 2012; ACM Press: New York, NY, USA, 2012. [CrossRef]
34. Dijkstra, E.W. A note on two problems in connexion with graphs. *Numer. Math.* **1959**, *1*, 269–271. [CrossRef]
35. Kahn, A.B. Topological sorting of large networks. *Commun. ACM* **1962**, *5*, 558–562. [CrossRef]
36. Aguilar, X. Performance Monitoring, Analysis, and Real-Time Introspection on Large-Scale Parallel Systems. Ph.D. Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2020.
37. Garcia, M.; Corbalan, J.; Labarta, J. LeWI: A runtime balancing algorithm for nested parallelism. In Proceedings of the 2009 International Conference on Parallel Processing, Vienna, Austria, 22–25 September 2009; pp. 526–533.
38. Arzt, P.; Fischler, Y.; Lehr, J.P.; Bischof, C. Automatic low-overhead load-imbalance detection in MPI applications. In *European Conference on Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 19–34.
39. Pearce, O.; Gamblin, T.; De Supinski, B.R.; Schulz, M.; Amato, N.M. Quantifying the effectiveness of load balance algorithms. In Proceedings of the 26th ACM International Conference on Supercomputing, Venice, Italy, 25–29 June 2012; pp. 185–194.
40. Tallent, N.R.; Adhianto, L.; Mellor-Crummey, J.M. Scalable identification of load imbalance in parallel executions using call path profiles. In Proceedings of the SC'10: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 13–19 November 2010; pp. 1–11.
41. Schmitt, F.; Stolle, J.; Dietrich, R. CASITA: A tool for identifying critical optimization targets in distributed heterogeneous applications. In Proceedings of the 2014 43rd International Conference on Parallel Processing Workshops, Minneapolis, MN, USA, 9–12 September 2014; pp. 186–195.
42. Wolf, F.; Mohr, B. Automatic performance analysis of MPI applications based on event traces. In *European Conference on Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 123–132.
43. Wolf, F.; Mohr, B. Automatic performance analysis of hybrid MPI/OpenMP applications. *J. Syst. Archit.* **2003**, *49*, 421–439. [CrossRef]
44. Schmitt, F.; Dietrich, R.; Juckeland, G. Scalable critical path analysis for hybrid MPI-CUDA applications. In Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, 19–23 May 2014; pp. 908–915.
45. Hermanns, M.A.; Miklosch, M.; Böhme, D.; Wolf, F. Understanding the formation of wait states in applications with one-sided communication. In Proceedings of the 20th European MPI Users' Group Meeting, Madrid, Spain, 15–18 September 2013; pp. 73–78.
46. OpenFOAM Tutorial, pitzDailyExptInlet Case Code Repository. Available online: https://github.com/OpenFOAM/OpenFOAM-9/tree/master/tutorials//incompressible/simpleFoam/pitzDailyExptInlet (accessed on 18 October 2023).
47. LAMMPS Benchmarks. Available online: https://docs.lammps.org/Speed_bench.html (accessed on 18 October 2023).