2022 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)

Implementing the Broadcast Operation in a Distributed Task-based Runtime

Rodrigo Ceccato*, Hervé Yviquel*, Marcio Pereira*, Alan Souza† and Guido Araujo*
*Institute of Computing, UNICAMP, Brazil
Email: r176848@dac.unicamp.br, {hyviquel,marciomp,guido}@unicamp.br
†Petrobras, Brazil
Email: alan.souza@petrobras.com.br

***Abstract*—Scientific applications that require high performance rely on multi-node and multi-core systems equipped with accelerators. Code for these heterogeneous architectures often mixes different programming paradigms and is hard to read and maintain. Task-based distributed runtimes can improve portability and readability by allowing programmers to write tasks that are automatically scheduled and offloaded for execution. Nevertheless, in large systems, communication can dominate the time spent during execution. To mitigate this, these systems usually implement collective operation algorithms that efficiently execute common data movement patterns across a group of processes. This work studies the usage of different broadcast strategies on the OpenMP Cluster (OMPC) task-based runtime. In addition to OMPC default behavior of on-demand data delivery, we introduce a routine to automatically detect data movement that is equivalent to a broadcast in the task graph and actively send it through a specialized algorithm. Our largest test setup using 64 worker nodes and broadcasting 64GB of data on the Santos Dumont cluster, using an extended version of Task Bench, showed a 2.02x speedup using the Dynamic Broadcast algorithm and 2.49x speedup when using the MPI broadcast routine, when compared to the default on-demand delivery.**

***Index Terms*—broadcast, collective operations, task-based runtime**

I. Introduction

High computational power is a necessity for a variety of applications. For instance, the increase in computing power allows scientists and engineers to run more fine-grained simulations and study more complex phenomena. Nevertheless, due to thermal constraints, single-chip performance cannot scale indefinitely and has hit the power wall, meaning that it is not viable to expect significant scaling in chip frequency. For this reason, High-Performance Computing (HPC) relies on large multi-core and multi-node systems that are often equipped with accelerators, like GPUs and FPGAs, to achieve high performance [1], [2].

Given their heterogeneity, the code that is written for these systems often mixes different paradigms to utilize all the available resources, leading to code that is hard to read and maintain [3]–[5]. In fact, Costa et al. [6] state that none of the existing programming models and runtimes are suitable for extreme-scale parallel systems.

The distributed task-based programming model can mitigate these issues. In this model, the program is written as a set of units of code, called tasks, with dependencies between them, represented by a Directed Acyclic Graph (DAG), in which each node represents a task and the edges represent their dependencies. A task is a unit of code that can be executed if all its dependencies are met, a dependency being, for example, some data that the task uses as input and is produced by another task [7].

In this model, the runtime is responsible for the efficient scheduling of the tasks, considering the system architecture and the data dependencies between them. This makes the code more readable and portable, as it does not have to be coded for one specific hardware, as the user only informs the dependencies while the runtime guarantees that the tasks are executed in a valid order. This differs greatly from a typical Message Passing Interface (MPI) application, in which the programmer must correctly move the data to a specific process rank, using low-level data movement routines, e.g. MPI pairs of send-receive functions. Shared-memory task-based runtimes usually use a work-stealing algorithm, in which threads execute tasks that are available in their local queue and steal tasks from other threads queues when they are idle. Diversely, on distributed task-based runtimes, wherein the cost of moving data is more significant, a more robust algorithm is typically used to schedule the task to the workers, e.g., the Heterogeneous Earliest Finish Time (HEFT) algorithm, that is an heuristic that tries to minimize the total execution time on heterogeneous resources.

Despite large distributed-memory systems being the solution to achieve high performance, the time spent moving data around can potentially represent a significant portion of the total execution time, even when the runtime offloads the computation using efficient scheduling algorithms. In this context, some systems implement specialized algorithms for some types of data movements, called *collective operations*, which designate data movement across a group of processes. One common movement is the broadcast operation, in which a given process sends the same data to every participant process. This kind of routine can represent a big portion of communication on scientific applications. In fact, from more than one hundred High Performance Computing (HPC) applications that Laguna et al. [8] studied, only one did not use collectives. Furthermore, other study found that some scientific applications spend more than eighty percent of communication time performing collectives operations [9].

978-1-6654-5157-4/22/$31.00 ©2022 IEEE
DOI 10.1109/SBAC-PADW56527.2022.00014

```
1  #pragma omp target enter data map(to: A[:SIZE])\
2                          nowait depend(inout: *A)
3  #pragma omp target nowait depend(in : *A)
4     foo(A);
5  #pragma omp target nowait depend(in : *A)
6     bar(A);
7  #pragma omp target nowait depend(in : *A)
8     baz(A);
9  #pragma omp target exit data map(from: A[:SIZE])\
10                         nowait depend(inout: *A)
```

Listing 1: OpenMP Cluster (OMPC) broadcast code example that can trigger a broadcast operation at runtime.

There are many types of collective operations that occur in scientific applications, e.g., reduction, gather, scatter. Each collective can be implemented with different algorithms, and the selection of an optimal algorithm to perform a given collective is itself a complex problem and varies according to the system architecture, and the size of the data, among other factors [9]. The MPI standard define routines to execute collectives, and most MPI implementations select between a set of algorithms at runtime, and many works propose additional approaches with improved performance in certain scenarios [10]–[12].

This work introduces the broadcast operation in the OMPC [13] programming model and compares the performance of three broadcast algorithms: the MPI broadcast routine, the implementation of the Dynamic Broadcast [14], and a naive peer-to-peer sequential send, used as a baseline. Finally, we extended the Task Bench [15] application, which is parameterized benchmark for task-based runtimes, to also test the broadcast operation and used it to evaluate our broadcast implementations. With this experimental setup we found that our contribution improves the execution time for a set of Task Bench configurations and idiomatically fits the OMPC directives.

The rest of the paper is organized as follows: Section II describes the OMPC runtime and programming model, Section III presents the addition of the broadcast operation in the OMPC model, Section IV exhibits and discusses the experimental results. Finally, Section V brifly comments how other task-based programming models handle collective operations and Section VI contains the conclusions and future works.

II. OpenMP Cluster

This section presents the OMPC [13][1] programming model and runtime, in which this work is built upon.

A. Programming model

OMPC is a distributed task-based programming model that aims to facilitate the writing of distributed applications. It extends the OpenMP programming model and uses code annotation to describe the portions of the code that can be offloaded to remote nodes. OMPC is fully compliant with the OpenMP standard and relies on its offloading constructs. The user annotates the code with said constructs to describe tasks and the runtime automatically handles their scheduling and offloading, using MPI to handle all inter-node communication. OMPC also offers fault-tolerance features, being able to detect if a node fails during execution [16].

OMPC is arguably easier to use when compared to other distributed runtimes because it uses only OpenMP directives, which are part of a widely used standard. We thus chose OMPC to study the introduction of the broadcast collective operation in its task-based programming model.

Listing 1 shows an example of a distributed program made with OMPC, using the OpenMP offloading semantics. In the code, the directive *target enter data map* on line 1 denotes that the array named *A* must be moved to the remote worker nodes. Lines 3-4 define a task that reads the array and executes the function *foo*, and this data dependency is with the directive *depend(in: *A)*. Similarly, other two tasks are defined by the lines 5-6 and 7-8. Finally, lines 9-10 indicate that the array *A* must be copied back to the node that started the execution.

In this code example it is also possible to see that the OMPC runtime must keep the data coherence across the worker nodes. Originally, the OpenMP target directives are designed for a local accelerator with only one memory region that all offloaded tasks can access. Since OMPC processes are spawned across many computers, each with its memory, the runtime must transmit the data to the corresponding node before executing a task that requires it.

B. Runtime

The OMPC plugin uses MPI as its underlying communication system, and one MPI process is spawned per node. One of these processes will be the head process, which will generate, schedule, and offload the tasks to the worker nodes. The runtime will carry out the data movements related to the *data map* directives from the user code, and also automatically move data between workers to keep data coherence, as a given buffer may be read or written in different worker nodes.

The LLVM implementation of OpenMP uses a *work-stealing* algorithm to schedule local tasks. Each worker thread has its queue with pending tasks, and when the queue is empty, the thread can *steal* tasks from other threads. This is an efficient approach for the shared memory model, in which the cost of stealing a task is very low and tasks are dynamically created and added to queues as the program executes. But since the OMPC runtime distributes tasks across remote worker nodes, the cost of communication must be taken into account during scheduling. Therefore, a static scheduler is used to map the tasks to the devices.

The tasks are first created from the user code and then passed to the OMPC scheduler. With the task graph, it then uses the HEFT [17] algorithm to allocate the tasks to the worker nodes. The HEFT algorithm tries to minimize the finish time by using estimates of the time to execute a given task and the time that it takes to move the data to a different node. Its heuristic uses the DAG with the *task graph* generated by OpenMP.

[1]OMPC is open-source and available at https://ompcluster.gitlab.io/

III. Broadcast operation in OMPC

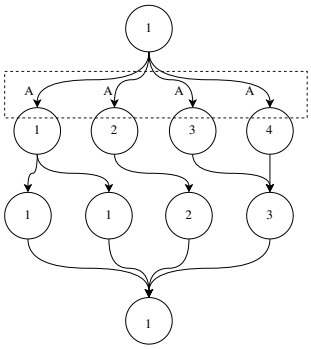


Fig. 1: Example of a scheduled task-graph: each vertex represents a task with the number of the node that they have been scheduled to execute written inside. The broadcast of some data *A* is being inferred from the scheduling.

Efficient implementation of collective operations algorithms is paramount to the performance of distributed runtimes, particularly when we scale the number of units that must communicate. The first challenge to introduce those operations in a task-based system is to find an idiomatic representation that the programmer can use to express them. As OpenMP was designed for shared-memory architectures, it does not provide an ordinary way to describe most collective operations. Moreover, the OMPC programming model differs a bit from the offloading process to a local accelerator, like a GPU, in which the data is copied to the accelerator memory before the kernel execution. By contrast, in the OMPC programming model, the user sends data through the *enter data map* directive and the runtime Data Management (DM) module handles the transfer between the remote worker nodes according to the dependencies.

A. Broadcast detection in OMPC

To introduce the broadcast operation in the OMPC programming model we check the scheduled task graph. As Figure 1 shows, we automatically detect patterns that correspond to a broadcast. Consequently, we can then use a specialized algorithm to actively perform the broadcast operation, instead of waiting for the *on-demand* delivery.

When not using any broadcast strategy, the OMPC runtime only sends the data to the node that will execute the first task that requires that buffer and then transmits the data on-demand between worker nodes. This transmission occurs when a task that requires that data is scheduled to a remote worker that does not contain a copy of the buffer. When the user enables a broadcast strategy, it delivers the data as soon as the broadcast is detected in the task graph, and the tasks that are created in that region only are dispatched after the broadcast algorithm completes, i.e., every participant worker has a copy of the data.

To construct a program that can trigger a broadcast strategy in the OMPC model, the user simply uses the *target enter data map* structure. If the buffers that the construct specify is used by multiple of the following tasks, then it represents a potential broadcast operation, as the example in Listing 1 shows. In this example, the three tasks defined on lines 3-4, 5-6 and 7-8 read the same buffer that has been sent previously thanks to the *target enter data map* pragma on lines 1. If the OMPC scheduler allocates them to distinct nodes, our implementation will use this information to automatically trigger a broadcast algorithm.

B. Broadcast algorithms

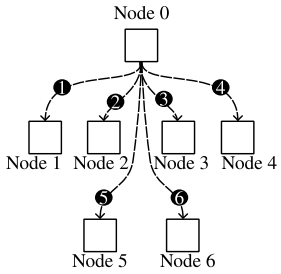
Some collective operation algorithms implementations may not be suited to the task-based model. For example, MPI follows the Single Program Multiple Data (SPMD) paradigm and requires that every process call the collective function to eventually progress and complete the operation. This kind of synchronization can kill a lot of benefits provided by task-based runtimes asynchronous execution. As Denis et al. [14] state, if a task-based runtime uses MPI as its underlying communication system and wishes to start a broadcast operation using the MPI broadcast routine it must notify every process with an order to call the routine. Figure 2b shows this limitation: since this command must be sent individually to each process, it adds a significant overhead that limits the benefits that the algorithms implemented for the MPI broadcast function try to achieve.

To mitigate this, Denis et al. [14] propose the Dynamic Broadcast algorithm. As Figure 2b shows, the routine uses self-contained messages that carry the propagation metadata along the data being broadcast. With this method, each process does not have to know in advance that they will participate in a broadcast operation. As a process receives the data, it also receives the list of processes to which it must propagate the data. This allows the broadcast operation to progress asynchronously and is suited for the distributed task-based model. In the example in the figure, node 0 broadcasts buffer *A* to all other nodes. In the first step, node 0 only sends a message to nodes 1 and 2, containing the data *A* with the propagation information, i.e., the order that node 1 must forward the data to nodes 3 and 4, and that node 2 must forward the data to nodes 5 and 6. Since this algorithm is designed for task-based systems, we implemented it as an strategy in OMPC.

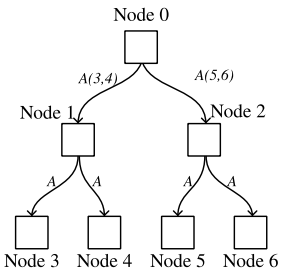
We added three broadcast options alongside the standard behavior of the OMPC runtime, that can be selected before execution. The available configurations are:

- **nobcast**: runtime DM delives data from detect broadcast on-demand.
- **P2P**: data from detected broadcast is sent to every device through a sequence of peer-to-peer communications.
- **mpibcast**: data from detected broadcast is sent to every device using the MPI broadcast routine
- **dynamicbcast**: data from detected broadcast is sent to every device using the Dynamic Broadcast algorithm.

All broadcast strategies are exemplified in Figure 3. Figure 3a shows the *p2p* strategy, which is a naive algorithm used here as a baseline. In this procedure, the root sends the data



(a) Dashed line represents an order to call the MPI Broadcast routine (steps 1 to 6), since MPI requires that every process calls the collective operation.



(b) Example of a Dynamic Broadcast algorithm: the root process (Node 0) sends the data with the list of nodes to which each receiver must forward the data to.

Fig. 2: Comparison between a task-based system using the MPI Broadcast routine and the Dynamic Broadcast algorithm.

to other participants using independent point-to-point transfers. The time complexity is thus linear with the number of nodes. Different from the *nobcast* strategy, the tasks execution only begins after every worker node has received a copy of the data.

Figure 2a shows the *mpibcast* strategy, that uses the MPI broadcast routine. As the OMPC runtime uses MPI as its underlying communication system, this allows it to use the MPI collective functionalities. To correctly execute a collective operation, all MPI processes must call the same operation, otherwise the program does not progress. This means that the root processes must notify every remote worker about the call to the MPI collective. Figure 2a shows the notification on steps 1 to 3 and the call to the collective function in step 4. The algorithm that MPI implements to perform the broadcast can vary with the MPI implementations.

Finally, since this synchronous notification step can be a limitation to the asynchronous potential of a task-based application, we also implemented the Dynamic Broadcast [14] algorithm. Figure 3c shows this approach, in which the propagation information is sent with the data. In the arrow indicated in the step 1, Node 0 sends the data with the instruction that Node 1 must forward it to Node 3 and Node 4. Different from *mpibcast*, in which the control information is sent before the broadcast starts, with the *dynbcast* strategy the control information is propagated with the data.

IV. Results

A. Experimental setup

We ran all benchmarks on the Santos Dumont (SDumont) supercomputer [18], located at National Laboratory of Scientific Computation (LNCC), located in Petrópolis, Rio de Janeiro. The SDumont cluster is the largest supercomputer dedicated to research in Latin America, with 36.472 CPU cores across 1.134 compute nodes. Specifically, the benchmarks were run in the SDumont partition composed of blades equipped with two Intel Xeon Cascade Lake Gold 6252 (24 cores @ 2.10 GHz) and 384GB of RAM. All nodes are interconnected with an InfiniBand network, including the distributed file system, with a theoretical throughput of 100Gb/s. For all experiments, we used MPICH 4.0.2 on nodes running Red Hat Enterprise Linux Server release 7.9. All executions were launched using the SLURM [19] job manager, to allocate the nodes with exclusivity, and we used Singularity [20] containers with all required dependencies to guarantee a replicable environment.

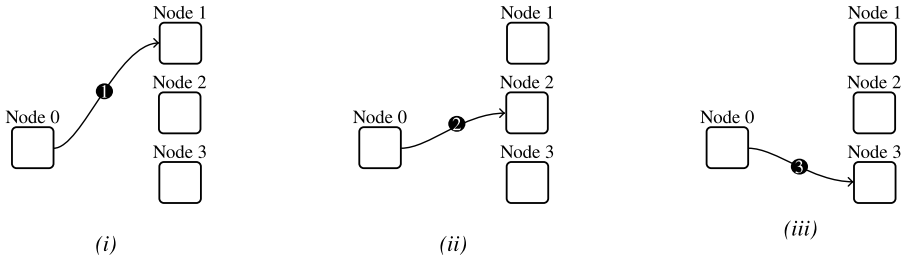
B. Task Bench

Task Bench [15] is a parameterized benchmark that can be implemented by distributed task-based runtimes and configured to execute benchmark scenarios. Once an implementation is done for a given runtime, it can run any Task Bench configuration. This eases the task of comparing different runtimes. OMPC has a Task Bench implementation, which allows its developers to compare its performance against other runtimes [13]. Currently, Task Bench is implemented in 17 distinct programming models, including some relevant task-based distributed runtimes that can be compared against OMPC, like StarPU and Charm++.

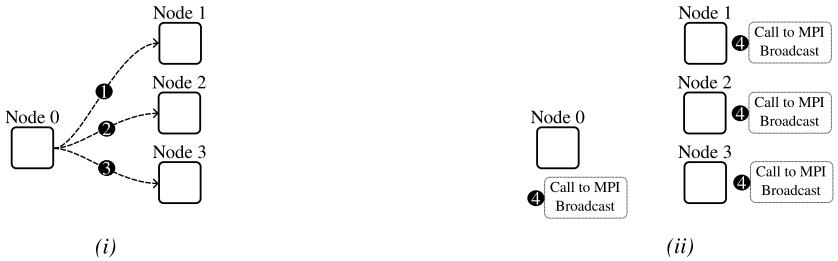
Task Bench accepts a sequence of flags to specify the parameters of the benchmark. Those flags set the size of the task graph, the kind of dependencies between the tasks, how much computation each task will perform, and how much data they will exchange. Figure 4 displays three examples of dependency types between tasks that occur in HPC applications.

In this work, we extended Task Bench to include a *broadcast parameter*. This introduces a flag that adds a buffer of parameterized size that is read by all tasks in the benchmark. This new parameter is set with the flag *-broadcast*, followed by the size in bytes of the buffer that will be broadcast. If left empty, no additional buffer is created. When the feature is used, it also checks for the correctness of the broadcast data. This configuration allows us to benchmark the broadcast feature that this work added to OMPC.

To evaluate the performance of the broadcast events, we compared a set of Task Bench executions using each broadcast strategy, varying the size of the buffer that is broadcast and the number of nodes. This configuration allows us to measure the differences between the event that relies on the MPI implementation, the naive P2P implementation, our implementation of the Dynamic Broadcast algorithm, and the original OMPC *on-demand* data delivery. We analyzed the total execution time



(a) *p2p* broadcast strategy: the root sequentially sends the data to each participant node, one after the other (steps 1 to 3).



(b) *mpibcast* strategy: the root sequentially orders (steps 1 to 3) every participant to call the MPI broadcast routine. Then, every participant calls the routine (step 4) and the MPI implementation executes the broadcst.

(c) *dynbcast* strategy: the root sends the data to its immediate children in the propagation tree. Along the data, the list of nodes to which each child must forward the buffer to is also sent.

Fig. 3: Comparison between each broadcast strategy implemented in OMPC.

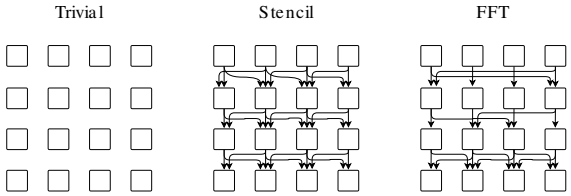


Fig. 4: Examples of dependency types between tasks that Task Bench can generate. Each square represents a task, and the arrows represent the dependencies. These are representative of common dependencies that occur in scientific applications.

using from 4 to 64 compute nodes, and broadcasting from 250MB to 64GB. This range is important since it can expose different optimizations that each strategy presents. In particular, the MPI implementation of collective operations may leverage hardware optimizations and use different algorithms based on the number of nodes or data size. It is important to note that this buffer is an additional data movement and is not related to the existing stencil dependencies that Figure 4 shows.

C. OMPC broadcast evaluation

All experiments use Task Bench with the *broadcast* parameter, using the stencil dependency type and task graph width that is equal to the number of nodes used. This is important to trigger the broadcast detection and evoke the specialized broadcast events that we added, since this guarantees that HEFT allocates a task to each node, thus forcing the broadcast buffer to be delivered to every participant.

The benchmark in Figure 6 shows how the runtime behaves for a small buffer. In Figure 5 is possible to note that even for the tests with the smallest buffer size, the naive P2P

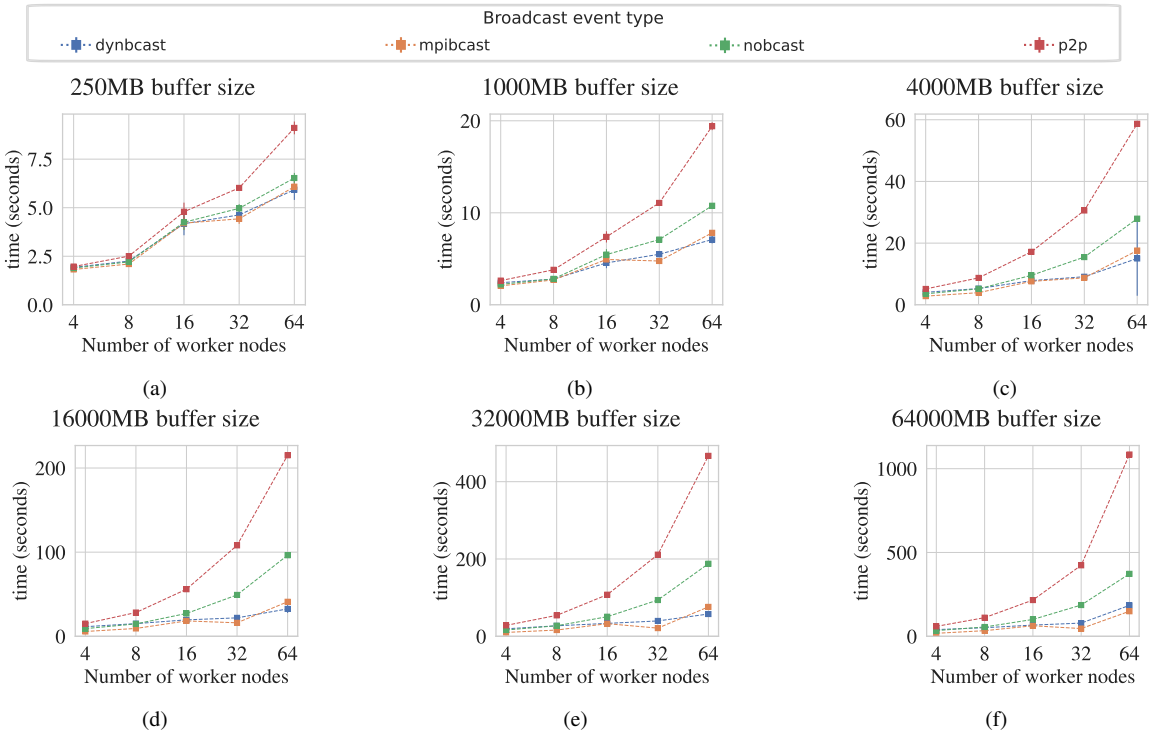


Fig. 5: Broadcast benchmarks for buffers from 250MB to 64000MB, from 4 to 64 worker nodes, using Task Bench with stencil dependencies. Average of 15 samples, error bar is standard deviation.

algorithm is easily outperformed for more than 32 worker nodes. For the other strategies, the performance is similar, as even for 64 nodes the *on-demand* OMPC behavior, indicated as *nobcast*, performs only slightly worse than a specialized broadcast algorithm. As we increase the buffer size, the gap between to the specialized broadcast algorithms grows, as can be seen on Figures 6a to 6c. This is expected for both *nobcast* and *P2P* strategies. When we increase the buffer size and the number of nodes, it takes longer for each iteration of the *P2P* algorithm, and it performs one iteration for each worker node. As for the *nobcast* strategy, it performs better than the *P2P* since it does not block the execution while the data is delivered to every node. Instead, the execution progress on the nodes that have the data, and it is delivered when a new worker requests the buffer, until every participant has a copy of it. Yet, it is greatly outperformed by the specialized routines, because since the data is delivered to one node at a time, as it is requested, it can at best scale linearly.

From Figure 5 we can observe that for 64 nodes and $64,000$MB of broadcast data, the *dynbcast* strategy shows a speedup of 2.06x compared to the *nobcast* strategy and of 5.88x when compared to the naive P2P strategy. When broadcasting a 500MB buffer, for 64 nodes, *dynbcast* outperforms the *nobcast* strategy by a factor of 1.31x, while the *mpibcast* outperforms the *nobcast* strategy by a factor of 1.24x.

Every algorithm scales with the number of nodes and buffer size, except for the *mpibcast* strategy, which sometimes becomes faster when we increase the number of nodes or the buffer size, as Figures 5d to 5f shows. The explanation for this behavior is that the MPICH implementation uses an internal routine to select which algorithm will be used to perform the broadcast operation according to the number of nodes and the buffer size. Thus, the improved performance when going from 16 to 32 workers can be explained by the selection of a different algorithm for more than 16 nodes.

Furthermore, the *mpibcast* strategy also performs better for large buffer sizes. That happens because MPICH can pipeline the sending procedure for buffers that are larger than a given threshold, and it is possible to visualize this conduct in Figures 5f and 6c. While in the other strategies that we added to OMPC each participant process must first receive the entire buffer before it starts sending it, MPICH (and thus the *mpibcast* strategy) can forward the buffer in chunks. This allows a receiving node to begin propagating the data without waiting for the entire buffer to arrive.

Besides, MPICH collectives potentially make use of hardware optimizations supported by the InfiniBand network. Since our implementation uses direct send-receive messages to fulfill the broadcast operation, it does not take advantage of network layer optimizations as the *mpibcast* strategy does. These factors explain why the *mpibcast* strategy has lightly better execution times. Still, as Figure 5 shows, as we increase the

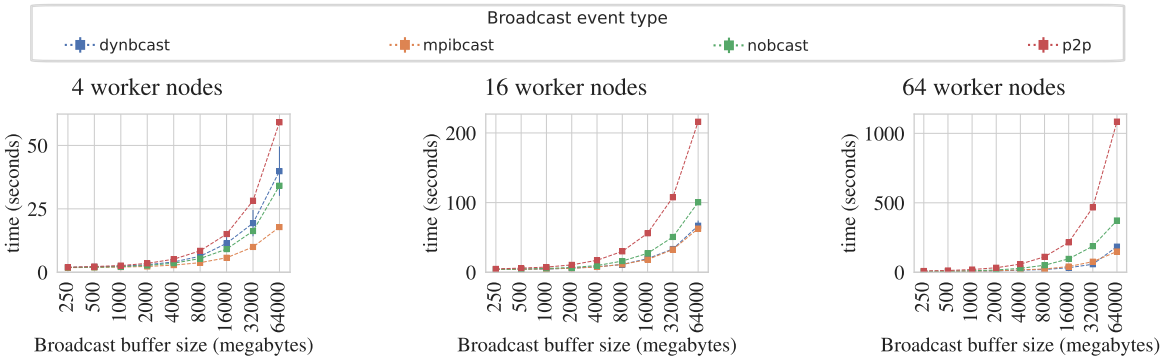


Fig. 6: Broadcast benchmarks for buffers from 250MB to 64000MB, from 4 to 64 worker nodes, using Task Bench with stencil dependencies. Average of 15 samples, error bar is standard deviation.

number of nodes, the gap between *mpibcast* and *dynbcast* shrinks.

Contrarily to the *nobcast* approach, which delivers the data as a worker node requires it, all other strategies pause the OMPC runtime execution until the data is delivered to every process. This synchronization does not let us take advantage of the asynchronous nature of the Dynamic Broadcast algorithm, which explicates why it does not outperform MPI as it does on the StarPU original implementation [14]. Even though the *dynbcast* strategy does not outperform the *mpibcast* in this experimental setup with its current implementation on the OMPC runtime, it still has advantages over the *mpibcast* method. Its interface exposes more granularity to the OMPC runtime. While the MPI broadcast routine only informs to the current processes if it completed its participation in the broadcast, the Dynamic Broadcast approach exposes to the OMPC runtime which processes have already received the buffer.

V. Related Work

There are many runtimes that aim to facilitate the development of distributed applications on heterogeneous clusters. Several employ the task-based model and integrate some collective operation algorithms.

StarPU [21] is a runtime system that focuses on heterogeneous architectures that provide an interface to unify execution on accelerators (and multi-core processors). It is a high-level library with task support with a simple interface to implement scheduling algorithms that can be chosen at runtime. The library transparently performs the data movements and is topology-aware. Nevertheless, collective operations must be called directly by the user though wrappers to MPI functions. Denis et al. [14] proposed the Dynamic Broadcast algorithm and implemented it in a new communication library for StarPU, which triggers the collective using auto-detection in the partial DAG.

Charm++ [22] is a C++ extension based on over-decomposition, in which the programmer subdivides the problem into units of parallelism called chares, that are balanced across the available resource by the runtime. The chares can be organized on arrays and such organization can be used to perform some collective operations. Invoking a method over an array of chars without a specific index results in a broadcast that is performed by the runtime considering the topology of the hardware and other architectural parameters. There is also support for reductions across chares in a chare array.

LibWater [23] also provides a uniform approach for heterogeneous distributed computing. It is a C/C++ library extension of the Open Computing Language (OpenCL) programming model, in which each OpenCL kernel can be scheduled to run on an OpenCL device (e.g. CPU or GPU). The library uses MPI to offload computation to remote nodes. The data transfer is handled by the runtime, but the user must specify which buffer a given kernel will read and/or write, and the transfer is made using OpenCL commands or MPI calls. The provided semantics do not allow the user to call collective operations, but LibWater implements a dynamic collective replacement algorithm that auto-detects some collective patterns by analyzing portions of the DAG at runtime and replacing point-to-point operations with the correspondent MPI collective operation.

Differently from these runtimes, our approach is idiomatically integrated with the OMPC architecture and programming model, and thus benefits from the widely used OpenMP directives and take advantage of the centralized DAG scheduling to identify the broadcast communication pattern when the tasks are scheduled, not having to rely on communication queue or partial DAG construction, which could miss some broadcast opportunities. These characteristics increment the relevance of our contributions.

VI. Conclusions and Future Work

HPC programs become more portable and readable when written for task-based systems. Yet, the way tasks and their dependencies are described can make it hard to idiomatically express collective operations, and these operations can be crucial to performance, especially at large scale. In this work,

we added the broadcast collective operation to the OMPC task-based programming model.

We added three distinct strategies to perform the broadcast operation and compared their performance on a modified version of Task Bench that we extended to perform a broadcast of configurable size. Our experiments showed that even on a distributed system with scheduling centralized on a head node, the usage of collective operations can have a meaningful impact in performance. Using 64 worker nodes and broadcasting a 64GB buffer, the use of a specialized broadcast strategy reduced the total execution time from 372s when using no broadcast strategy to 184s when using the Dynamic Broadcast algorithm (2.02x speedup) and to 149s (2.49x speedup) when using the MPI broadcast routine. In the current implementation, MPI collectives showed better performance as it leverages hardware optimizations and segments large buffers to improve the time it takes to forward the data in the broadcast. Still, the Dynamic Broadcast algorithm exposes more granular control to the runtime and can be further improved to better integrate with the dynamic execution of the tasks.

With LLVM 14, the *Dynamic Broadcast* implementation in OMPC can be improved to be more dynamic, in the sense that we could start executing tasks on nodes that already received the data, instead of waiting for the whole broadcast to complete. This is a direct next step from this work and can probably improve our implementation of this strategy.

In future works, it seems relevant to introduce other collective operations in the OMPC runtime, through auto-detection, and through new pragmas, since operations like reductions, gather, and scatter are common in many scientific applications.

Finally, it can be interesting to collect data from these collective operations with different applications, since we only used a synthetic benchmark. Scientific applications that execute broadcasts during execution and tests on larger number of nodes could provide better insights about the proposed strategies.

ACKNOWLEDGMENTS

This work is supported by Petrobras under grant 2018/00347-4, and by the National Council for Scientific and Technological Development (CNPq) under grant 402467/2021-3. We would also like to thank the LNCC for granting the computational resources at the Santos Dumont supercomputer that were crucial to perform the experiments for this project.

REFERENCES

[1] S. P. Crago and J. P. Walters, "Heterogeneous cloud computing: The way forward," *Computer*, vol. 48, pp. 59–61, 1 2015.

[2] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier, "High-performance computing: Clusters, constellations, mpps, and future directions," *Computing in Science and Engineering*, vol. 7, pp. 51–59, 3 2005. [Online]. Available: http://ieeexplore.ieee.org/document/1401802/

[3] D. D'Agostino, A. Clematis, E. Danovaro, E. Jeannot, and J. Zilinskas, "Heterogeneous parallel computing platforms and tools for compute-intensive algorithms: a case study," *High-Performance Computing on Complex Environments*, vol. 96, p. 193, 2014.

[4] C. M. Pancake, "What computational scientists and engineers should know about parallelism and performance," *Computer Applications in Engineering Education*, vol. 4, no. 2, pp. 145–160, 1996.

[5] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 392–403, 1995.

[6] G. D. Costa, T. Fahringer, J. A. Rico-Gallego, I. Grasso, A. Hristov, H. D. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. L. Stavrinides, D. Talia, P. Trunfio, and H. Astsatryan, "Exascale machines require new programming paradigms and runtimes," *Supercomputing Frontiers and Innovations*, vol. 2, no. 2, pp. 6–27, 2015.

[7] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9219, no. c, pp. 1–14, 2017.

[8] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of MPI usage in open-source HPC applications," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2019.

[9] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Tertiary Education and Management*, vol. 10, no. 2, pp. 127–143, 2004.

[10] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Tertiary Education and Management*, vol. 10, pp. 127–143, 2004.

[11] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Computing*, vol. 35, pp. 581–594, 12 2009.

[12] J. L. Träff and A. Ripke, "Optimal broadcast for fully connected processor-node networks," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 887–901, 7 2008.

[13] H. Yviquel, M. Pereira, E. Francesquini, G. Valarini, P. R. Gustavo Leite, R. Ceccato, C. Cusihualpa, V. Dias, S. Rigo, A. Souza, and G. Araujo, "The OpenMP Cluster Programming Model," *51st International Conference on Parallel Processing Workshop Proceedings (ICPP Workshops 22)*, 2022.

[14] A. Denis, E. Jeannot, P. Swartvagher, and S. Thibault, "Using dynamic broadcasts to improve task-based runtime performances," in *European Conference on Parallel Processing*. Springer, 2020, pp. 443–457.

[15] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca, S. Mirchandaney, W. Leek, S. Treichler, P. McCormick, and A. Aiken, "Task bench: A parameterized benchmark for evaluating parallel runtime performance," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2020-November, 2020.

[16] P. H. D. F. Rosso, "OCFTL: An MPI Implementation-Independent Fault Tolerance Library for Task-Based Applications," in *High Performance Computing: 8th Latin American Conference, CARLA 2021, Guadalajara, Mexico, October 6-8, 2021, Revised Selected Papers*. Springer Nature, 2022, p. 131.

[17] H. Topcuoglu, S. Hariri, and M. Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, 3 2002.

[18] I. Gitler, A. T. A. Gomes, and S. Nesmachnow, "The latin american supercomputing ecosystem for science," *Communications of the ACM*, vol. 63, no. 11, pp. 66–71, 2020.

[19] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[20] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[21] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, feb 2011. [Online]. Available: http://doi.wiley.com/10.1002/cpe.1631

[22] L. Kale, N. Jain, and J. Lifflander, "Charm++," in *Programming Models for Parallel Computing (Scientific and Engineering Computation)*, P. Balaji, Ed. The MIT Press, 2015.

[23] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "LibWater: Heterogeneous distributed computing made easy," *Proceedings of the International Conference on Supercomputing*, pp. 161–170, 2013.