Using Dynamic Broadcasts to improve Task-Based Runtime Performances

Alexandre Denis[1,2], Emmanuel Jeannot[1,2], Philippe Swartvagher[1,2], and Samuel Thibault[2,1]

[1] Inria Bordeaux – Sud-Ouest, 33405 Talence, France
{alexandre.denis,emmanuel.jeannot,philippe.swartvagher}@inria.fr
[2] LaBRI, University of Bordeaux, 33405 Talence, France
samuel.thibault@labri.fr

Abstract. Task-based runtimes have emerged in the HPC world to take benefit from the computation power of heterogeneous supercomputers and to achieve scalability. One of the main bottlenecks for scalability is the communication layer. Some task-based algorithms need to send the same data to multiple nodes. To optimize this communication pattern, libraries propose dedicated routines, such as `MPI_Bcast`. However, `MPI_Bcast` requirements do not fit well with the constraints of task-based runtime systems: it must be performed simultaneously by all involved nodes, and these must know each other, which is not possible when each node runs a task scheduler not synchronized with others.
In this paper, we propose a new approach, called dynamic broadcasts to overcome these constraints. The broadcast communication pattern required by the task-based algorithm is detected automatically, then the broadcasting algorithm relies on active messages and source routing, so that participating nodes do not need to know each other and do not need to synchronize. Receiver receives data the same way as it receives point-to-point communication, without having to know it arrives through a broadcast.
We have implemented the algorithm in the StarPU runtime system using the NewMadeleine communication library. We performed benchmarks using the Cholesky factorization that is known to use broadcasts and observed up to 30% improvement of its total execution time.

Keywords: task-based runtime systems · communications · collective · broadcast

1 Introduction

Scalability of applications over clusters is limited, among other things, by synchronizations, an extreme example being *Bulk Synchronized Parallelism* (BSP). To increase performance, task-based runtime systems try to avoid any synchronization through asynchronicity in the way they schedule tasks on nodes. To follow this scheduling model in order to ensure scalability, communications have also to support asynchronicity.

For communications, task-based runtime systems often rely on MPI. However, MPI libraries and the MPI interface are not designed with such use in mind. A problem arises when a piece of data produced by one task is a dependency for several other tasks on several nodes. A natural way to send the same data to multiple nodes would be to use the `MPI_Bcast` or `MPI_Ibcast` primitives. However, this approach assumes all nodes know in advance that this data will arrive through a broadcast instead of a point-to-point operation, and that they know each other. This assumption is not met in the case of a dynamic task-based runtime system, where nodes ignore the state of the task scheduler on other nodes, and thus they typically use a naive broadcast algorithm with linear complexity.

In this paper, we propose an algorithm for a dynamic broadcast, where only the root knows the list of all recipients, and recipients do not have to know in advance whether data will arrive through a broadcast or a point-to-point operation, while still being able to leverage optimized tree-based broadcast algorithms.

In short, this paper makes the following contributions: we propose a dynamic broadcast algorithm, suitable for use by task-based runtime systems; we implemented the mechanism in our NewMadeleine [5] communication library, and modified StarPU [4] to take benefit from it; we performed benchmarks to show the performance improvement.

The rest of this paper is organized as follows. Section 2 details why broadcasts using `MPI_Bcast` are not suitable for task-based runtime systems. Section 3 introduces our algorithm for dynamic broadcasts. Section 4 presents its implementation in NewMadeleine and StarPU. In Section 5, we evaluate our solution using micro-benchmarks and a Cholesky factorization kernel. In Section 6 we present related works, and Section 7 concludes.

2 Broadcasts in Dynamic Task-Based Runtime Systems

With task-based runtime systems, the application programmer writes applications decomposed into several tasks with dependencies. Each task is a subpart of the main algorithm. All tasks with their dependencies form a DAG (Direct Acyclic Graph); tasks are vertices, and edges represent a data dependency between two tasks: the child task needs data produced by its predecessor(s). In order to get task-based runtime systems to work on distributed systems, tasks are distributed among available nodes. When dependent tasks are not located on the same node, an edge spans across two different nodes and the runtime system automatically handles the data transfer.

A given piece of data may be a dependency for multiple tasks (a vertex with several outgoing edges). If the receiving tasks are located on different nodes, the same data will have to be sent to multiple nodes. This communication pattern is generally known as a *multicast*, or a *broadcast* in MPI speaking, which is a kind of *collective* communication.

The naive way to perform a broadcast is to send data from the root to each node using independent point-to-point transfers. With such an implementation, the duration of a broadcast is *linear* with the number of nodes. MPI libraries

usually implement much better algorithms [12, 15, 13] for `MPI_Bcast`, such as binary trees, binomial trees, pipelined trees, or 2-trees, which exhibit a *logarithmic* complexity with the number of nodes. It is thus strongly advised to use `MPI_Bcast` to broadcast data when possible.

However, for task-based runtime systems that dynamically build the DAG (such as StarPU [3] or Quark [9]), nodes do not have a global view of data location and do not synchronize their scheduling. This makes the use of `MPI_Bcast` or `MPI_Ibcast` difficult and inefficient, for several reasons:

detection – all the information the runtime system knows about data transfers is the DAG. A broadcast appears as a task whose result is needed by multiple other tasks. However, in general the whole DAG is not known statically but generated while the application is running. Therefore, the runtime system cannot know whether the list of recipient is complete or if another recipient will be added later.

explicit – function `MPI_Bcast` has to be called explicitly by the sender and the receivers. Therefore, each receiver node have to know in advance whether a given piece of data will arrive through an `MPI_Bcast` or a point-to-point `MPI_Recv`. Application programmer cannot give any hint, since communications are driven by the DAG, and thus depends on where tasks are mapped during the execution.

communicator – function `MPI_Bcast` works on a *communicator*, a structure containing all nodes taking part in the broadcast (sender node and recipients). The construction of a communicator is also a collective operation: to build it, each node participating in a communicator must know the list of all nodes in the communicator. Thus, if we build a communicator containing a specific list of nodes for a given broadcast operation, all nodes have to know the list of all nodes participating in the broadcast.

Yet, the runtime system on a node only has a local view of the task graph: receiver nodes know which node will send them the data, but they do not know all other nodes which will also receive the same data. Hence, building an MPI communicator is impossible without first sending the list of nodes to all nodes, but that would mean we need a broadcast before being able to do a broadcast!

synchronization – even if we use a non-blocking `MPI_Ibcast` instead of a blocking `MPI_Bcast`, it works on a communicator. The creation of a communicator with the precise set of nodes is a blocking operation and has to be performed by all nodes at the same time. This constraint is somewhat alleviated by the non-blocking flavor of communicator creation in the future MPI 4.0 version. Nonetheless, a single communicator creation may take place at the same time. This means broadcasts, and their associated communicator creation, must nonetheless be executed in the same order by all nodes, which implies some kind of synchronization to agree on broadcast scheduling, thus hindering one of the most important feature of distributed task-based runtime system: its ability to scale by avoiding unnecessary synchronization.

As a consequence, the mechanisms needed to actually use an `MPI_Bcast` to broadcast data in a task-based runtime system are likely to cost more than the benefit brought by the use of an optimized broadcast.The general problem is being able to use desynchronized and optimized broadcasting algorithms, without all nodes of the broadcast know each other. We present in this paper the solution we developed to achieve this goal.

3 Our Solution: Dynamic Broadcasts

3.1 Detection of Collectives

As explained in Section 2, the detection of broadcast patterns is not straightforward since the DAG is dynamic.

From the dependency graph view, a broadcast is a set of outgoing edges from the same vertex and going to tasks executed on different nodes. During task graph submission, the runtime system creates a send request for each of these edges, even before the data to send is available. When the data becomes available, the requests are actually submitted to the communication library.

The detection of broadcast consists in noticing on creating a send request that one already exists for the same data, and aggregating them into a single request with a list of recipients. When the data becomes available, if the list contains more than one recipient, a broadcast is submitted to the communication library.

This method may miss some send requests if they are posted after the data became ready, *i.e.* if a task is submitted after the completion of the task that produces the data it depends on. This happens if the task graph submission takes longer than the task graph execution (which is not supposed to happen in general), or if the application delays submission of parts of the task graph for its own reasons, in which case the runtime system did not need to send this data sooner anyway. Code instrumentation showed that 98 % of broadcasts were detected with the correct number of recipients for the Cholesky decomposition described in Section 5. These missed broadcasts correspond only to communications performed during the very beginning of the algorithm, when the application has only started submitting the task graph, and thus task execution has indeed caught up quickly and made some data available before the application could submit all inter-node edges for them. Quickly enough, tasks submission proceeds largely ahead of tasks execution, and all broadcasts are detected.

To avoid redundant transfers of the same data between two nodes, a cache mechanism is used [3]. If two tasks scheduled on the same node need a piece of data from another node, only one communication will be executed. Hence, the recipient list does not contain duplicates.

3.2 Dynamic Broadcast Algorithm

We propose here a broadcast algorithm, that we call *dynamic broadcast*, that fulfills the requirements to be used by task-based runtime systems, namely: use

optimized broadcast algorithms; all recipients of the broadcast do not have to know each other; have a seamless integration for the receiver who is expecting a point-to-point communication.

Optimized broadcast algorithm Several optimized algorithms for broadcast exist [16]. The main idea of all these optimized algorithms is that after a node received data, it sends this data to other recipients, so that the root node has less communications to execute, which shortens the global execution time. For most algorithms, routing is organized as a tree: the source node sends to a set of nodes, each of these nodes then sends to a set of other nodes, and then recursively until all recipients get the data. Tree-based algorithms have usually a logarithmic complexity in the number of nodes. The choice of a broadcast algorithm depends mainly on the number of recipients and the size of data to transmit.

We choose to implement binomial trees because this broadcast algorithm is the best trade-off for a single all-purpose algorithm to get good performance on a wide range of data sizes and numbers of nodes. Other optimized algorithms [15, 13, 12] could be used in our dynamic broadcast, following the same approach.

In the binomial tree algorithm, each node receiving data contributes to the diffusion by sending data to next nodes, and keep sending data to other recipients until all nodes received the data. The Figure 1 illustrates a broadcast to 6 recipients: node 0 starts by sending to node 4, then 0 sends to 2 and at the same time 4 sends to 6 and finally while 0 is sending to 1, 2 is sending to 3 and 4 is sending to 5. The binomial tree has a logarithmic complexity in the number of nodes.

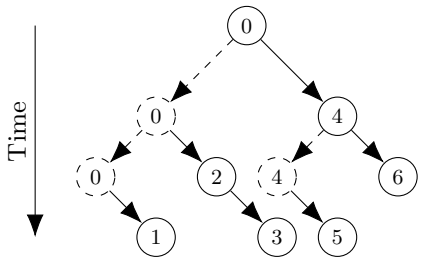

Fig. 1. Example of binomial tree with six recipients. Levels in the tree are steps in the algorithm.

Self-contained messages Since nodes do not know in advance whether they will be participating in a broadcast, our algorithm is based on self-contained messages. They are active messages, processed outside of the application flow, without requiring the application to call specific primitives in the communication

library. The message contains all the information needed to unroll the collective algorithm.

Only the root of the broadcast knows the complete list of recipients. Recipients themselves only need to know to which nodes they will need to forward the data, *i.e.* the sub-tree below them. We send this list of nodes together with data, in the header of the active message. When a node forwards data to other nodes, it trims the list of nodes so as to include only the nodes contained in the relevant sub-tree.

In the case depicted in Figure 1, the list of nodes sent by node 0 to 4 is $\{5, 6\}$, the list sent to 2 is $\{3\}$ and the list sent to 1 is empty.

The runtime system sets a *priority* level for each communication request, depending on task priorities, defined by the application (during the submission of tasks, the user can define the priority of each task, by specifing an integer). This information may be used by a communication library that is able to schedule packets by priority like NewMadeleine. We reorder the list of nodes of broadcasts so that higher-priority requests are closer to the root of the tree, for them to get data earlier. Moreover, in addition to the list of nodes, we transmit the list of priorities. This way, when inner nodes of the tree have to forward messages, they get inserted in their local packet flow with the right priority.

The general idea behind this mechanism is that routing information are transmitted with the data itself, and are not assumed to be prior knowledge, as `MPI_Bcast` would otherwise require.

Transparent receive When a request which is part of a broadcast is received, the data is forwarded to nodes contained in the list, following the binomial tree algorithm, and data is delivered locally. Since nodes cannot predict whether data will arrive through point-to-point communication or through a broadcast, on the receiver side our algorithm injects data received by a broadcast in the path of point-to-point receive. The runtime system posts a regular point-to-point receive request, and when data arrives through a dynamic broadcast, it is actually received by this point-to-point request for a seamless integration.

We called our algorithm *dynamic broadcast* because nodes realize they take part in a broadcast in a dynamic fashion, on the fly at the same time when data arrives.

4 Implementation

Our implementation was made within the StarPU task-based runtime and the NewMadeleine communication library. This Section introduces both libraries and presents implementation details of our dynamic broadcast algorithm.

4.1 StarPU

StarPU [4] is a task-based parallel and distributed runtime system. In its single-node form, StarPU lets HPC applications submit a sequential flow of tasks, it

infers data dependencies between tasks from that flow, and it schedules tasks concurrently while enforcing these dependency constraints. The distributed version of StarPU [3] extends the Sequential Task Flow model. The application gives an initial distribution of data on the participating nodes, and every node submits the same flow of tasks to its local StarPU instance. Each StarPU instance then infers whether to execute a task or not from the piece of data the task writes to. The instance that owns the piece of data written to, executes the task. Moreover, each StarPU instance infers locally when to generate send and receive communications to serve inter-instance data dependencies. This distributed execution model does not involve any master node or synchronization. Instead, all instances are implicitly coordinated by running the same state machine from the sequential task flow.

Two back-ends are implemented in the distributed version of StarPU. One relies on MPI standard and has to be used with an MPI implementation (such as OpenMPI). The other one uses NewMadeleine routines to take benefit from its specific features beyond the MPI interface.

4.2 NewMadeleine

NewMadeleine [5] is a communication library which exhibits its own native interface in addition to a thin MPI layer called MadMPI. The work described in this paper is located in the NewMadeleine native interface. The originality of NewMadeleine compared to other communication libraries and MPI implementations is that it decouples the network activity from the calls to the API by the user. In the interface presented to the end-user, primitives send and receive *messages*. NewMadeleine applies an optimizing strategy so as to form *packets* ready to be sent to the network. A *packet* may contain multiple *messages* (aggregation), a *message* may be split across multiple *packets* (multi-rail), and *messages* may be actually sent on the wire out-of-order depending on packet scheduler decision and priorities. NewMadeleine core activity is triggered by the network. When the network is busy, *messages* to be sent are simply enqueued; when the network becomes ready, an optimization strategy is called to form a new *packet* from the pending *messages*. A receive is always posted to the driver, and all the activity is made of up-calls (event notifiers) triggered from the lowest layer when the receive is completed, which make active messages a natural paradigm for NewMadeleine.

4.3 Dynamic collectives implementation

Dynamic broadcasts were implemented as a new interface of NewMadeleine, and the NewMadeleine backend of StarPU was adapted to exploit this new interface.

The detection of broadcasts is implemented in StarPU. When the application submits a task B which depends on data produced by a task A mapped on a different node, an inter-node communication request is issued. If a previous request or collective was already detected for this data, the new request

is merged in to get a bigger collective. Most often, task submission proceeds quickly, and thus the submission front is largely ahead of the execution front. As a consequence, when task B is submitted, task A will probably not have been executed yet, and similarly for all tasks which depend on A. This is why our approach catches most potential for broadcasts. Once task A is completed and thus the data available for sending, the whole collective request is handed to NewMadeleine.

The dynamic broadcast itself is implemented in NewMadeleine, using its non-blocking `rpc` interface for active messages. They use a dedicated communication channel that is separate from the channel used for point-to-point communications. Thus, the library distinguishes broadcasts, which needs special processing, from regular point-to-point messages. The library is always listening for `rpc` requests and is thus able to always process dynamic broadcasts for all tags and from all nodes.

To manage seamless receive of a broadcast by a point-to-point request, point-to-point requests are registered by the dynamic broadcast subsystem. Conversely, if the data for a receive comes by the point-to-point way, the request is removed from the table in the dynamic broadcast subsystem.

When a broadcast is received, the matching point-to-point receive is searched and data is received in-place in the buffer of the point-to-point request, forwarded to nodes in sub-trees, and the point-to-point request is notified completion. If the matching point-to-point receive was not posted yet, the broadcast request is locally postponed until the matching point-to-point receive is posted. To be able to match a message arriving through a broadcast with a point-to-point request, the original source node (root of the broadcast) is also sent together with data, the list of nodes, and their associated priority.

5 Evaluation

In this section, we present the performance results we obtain for mechanisms presented in this paper. We executed micro-benchmarks to ensure the broadcast performances are as expected and then we evaluated the impact on a real computing kernel, the Cholesky factorization.

The benchmarks were carried out on two different clusters: `inti` from CEA and `plafrim`. `inti` nodes are dual Xeon E5-2680 at 2.7 GHz, with 16 cores and 64 GB RAM, and equipped with Connect-IB *InfiniBand* QDR (MT27600). Default MPI on the machine is OpenMPI 2.0; since this version is ancient, we compiled the latest OpenMPI 4.0. `plafrim` nodes are dual Xeon Gold 6240 at 2.6 GHz with 36 cores and 192 GB RAM, and equipped with Intel *Omni-Path* 100 series network. Default MPI on `plafrim` are OpenMPI 3.0 and OpenMPI 4.0.

5.1 Micro-benchmarks

To be sure our algorithm and its implementation have the expected performances, we conducted micro-benchmarks of the dynamic broadcast and compared its performance against `MPI_Bcast` and `MPI_Ibcast` of MadMPI, and a

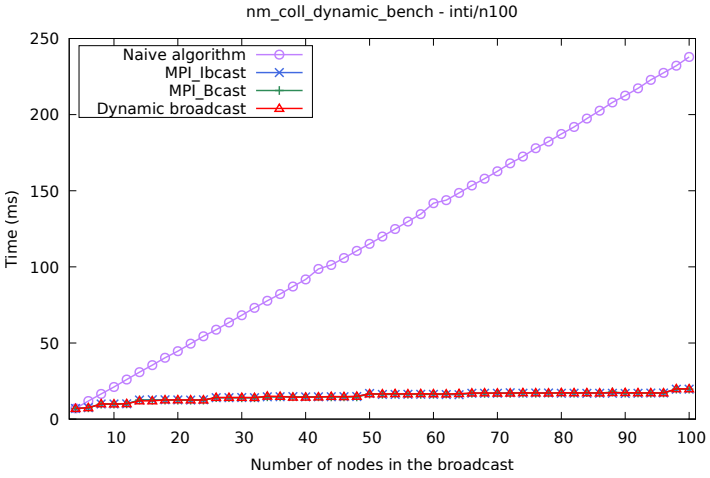

Fig. 2. NewMadeleine micro-benchmark on cluster `inti` on 100 nodes (1600 cores), comparing broadcast algorithms

naive broadcast (a loop of point-to-point requests to the recipient nodes). The duration of the broadcast is defined as the time difference between the start on the root node and the last received data on the last node.

The result of this micro-benchmark on 100 nodes of the `inti` cluster is depicted in Figure 2 for 8 MB of data. As expected, naive broadcast exhibits a linear complexity with the number of recipients and both dynamic and regular broadcasts have a logarithmic complexity. The performance difference between dynamic broadcast and regular MPI broadcast is insignificant, which shows that the additional routing data and the treatment when receiving data is negligible.

5.2 Cholesky Factorization

To evaluate the gain brought by dynamic broadcast, we have evaluated its performance on a Cholesky factorization.

Description In Algorithm 1, we depict the tiled version of the Cholesky Factorization algorithm. For a given symmetric positive definite matrix A, the Cholesky algorithm computes a lower triangular matrix L such that $A = LL^T$. In the tiled version used here, the matrix is decomposed in $T \times T$ square tiles where $A[i][j]$ is the tile of row i and column j. At each step k it performs a Cholesky factorization of the tile on the diagonal of panel k (`POTRF` kernel) then it updates the remaining of the tiles of the panel using triangular solve (`TRSM` kernel). The trailing sub-matrix is updated using the `SYRK` kernel for tiles on the diagonal and matrix multiply (`GEMM` kernel) for the remaining tiles.

Algorithm 1: Tiled version of the Cholesky factorization.

```
1 for k = 0...T − 1 do
2     A[k][k] ← POTRF(A[k][k])
3     for m = k + 1...T − 1 do
4         A[m][k] ← TRSM(A[k][k], A[m][k])
5     for n = k + 1...T − 1 do
6         A[n][n] ← SYRK(A[n][k], A[n][n])
7         for m = n + 1...T − 1 do
8             A[m][n] ← GEMM(A[m][k], A[n][k], A[m][n])
```

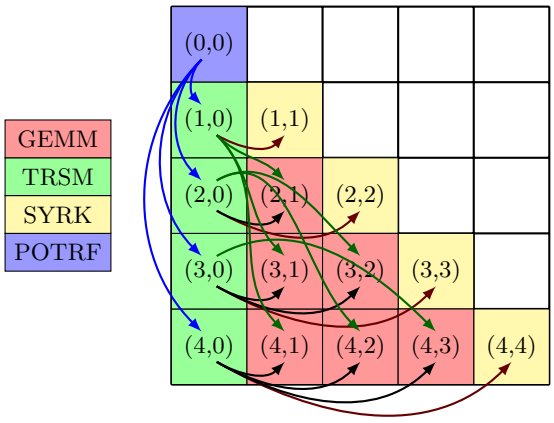

Fig. 3. The 2 different types of broadcasts for the Cholesky factorization for $T = 5$ and $k = 0$. Blue arrows : from 1 `POTRF` to $T - k - 1 = 4$ `TRSM`. Green and black arrows from 1 `TRSM` to $T - k - 2 = 3$ `GEMM` and red arrows to 1 `SYRK`.

This algorithm is a good use-case for the dynamic broadcast problem. Indeed, as shown in Fig. 3, the $A[k][k]$ tile computed by the `POTRF` kernel is broadcasted to the $T - k - 1$ `TRSM` kernels of the same panel (blue arrows). Moreover, each $A[m][k]$ $(m > k)$ tile generated by the `TRSM` kernels (line 4), is used by one `SYRK` kernel (to update the tile $A[m][m]$, red arrows) and $T - k - 2$ `GEMM` kernels (to update the tiles $A[m][n]$ $(k < n < m)$, black arrows ; and the tiles $A[m][n]$ $(m < n < T)$, green arrows). As seen in Section 3, in StarPU, all the communication and especially the collective communication are inferred at runtime by the system based on the dependencies that are described in the task graph generated from the program. Furthermore, since for both cases, the same tile is broadcasted to all the kernels and several kernels are executed by a same node, the runtime system is able to factorize the communication by giving the list of compute nodes that require the considered tile. In practice, as nodes are layout using squared 2D

Block-cyclic distribution, the maximum number of nodes involved in a broadcast is $\mathcal{O}(\sqrt{P})$ where P is the total number of nodes.

We used the CHOLESKY factorization from the CHAMELEON library [2], which can use STARPU as task-based runtime system.

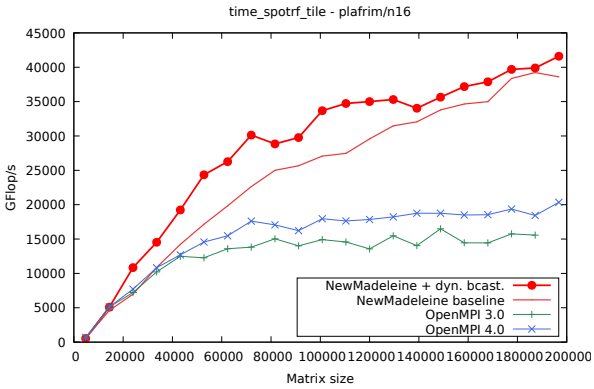

Fig. 4. CHAMELEON CHOLESKY factorization performance on cluster `plafrim` on 16 nodes (512 cores)

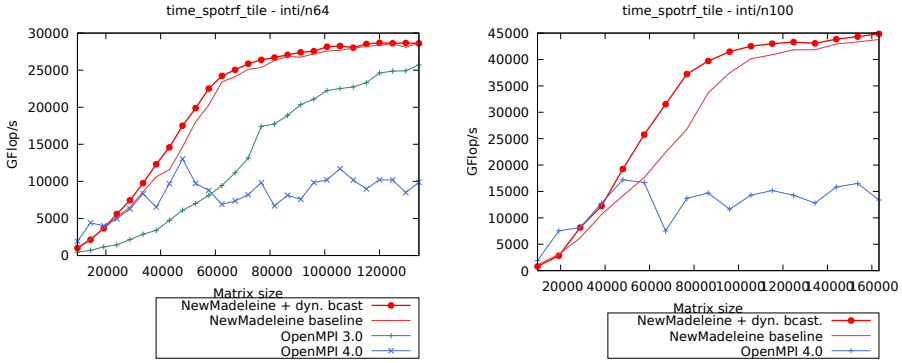

Fig. 5. CHAMELEON CHOLESKY factorization performance on cluster `inti`: on left, on 64 nodes (1024 cores); on right, on 100 nodes (1600 cores)

Results Results of the CHOLESKY benchmark on cluster `plafrim` on 16 nodes is depicted in Figure 4. Results for machine `inti` on 64 and 100 nodes is shown

in Figure 5. There is one MPI process per node and each point on graphs is the average of two runs. We compare the *baseline* NewMadeleine version without dynamic broadcast against NewMadeleine with dynamic broadcast. Additionally we represent the performance we obtain with MPI as a reference. The performance difference between NewMadeleine and MPI is explained [8] by other mechanisms beyond the scope of this paper.

On all 3 cases, dynamic broadcasts improve performances[3] of the Cholesky factorization, mainly on small matrices. On `plafrim` on 16 nodes (Figure 4), the best performance improvement is 25 %. On `inti` (see Figure 5) on 64 nodes the improvements is up to 20 % and on 100 nodes up to 30 %. Since the number of nodes in broadcasts increases with the total number of nodes, the more nodes are used, the more the broadcast takes time, thus dynamic broadcasts improve overall scalability with the number of nodes. For larger matrices, communications have less impact since there are always enough ready tasks to execute before having to wait for data coming from the network, hence it is not surprising to observe the best performance improvement for small matrices.

The impact of using *dynamic broadcasts* in Cholesky execution needs to be more studied and requires a deep analysis of runtime system and communication library internal behaviours. Since this analysis is not straightforward, we consider it as out of the scope of this paper.

6 Related Works

Broadcasting algorithms have already been discussed a lot [16, 12, 15, 13], but are an orthogonal problem to work described in this paper which can rely on any tree-based algorithm.

The idea to optimize collective communications by sending only one message per receiving node when multiple tasks with the same input share the same node has been proposed in early task-based runtime systems [10]. However, in this work, no optimization was performed in the way the data was broadcasted to the different nodes.

Parsec [7] is a task-based runtime system, based on a *Parameterized Task Graph* (PTG), an algebraic representation of the dependency graph. Such kind of graph can be entirely stored in the memory of each node since the memory used for its representation is linear in the number of task types, and not in the number of tasks. Since all nodes know the full task graph, they can easily know all nodes involved in a broadcast and the entire graph being known at the beginning of the execution, explicit call to broadcast routines can be made. In practice, Parsec uses binomial or chained trees, on the top of MPI point-to-point requests. Broadcasts are identified directly from the algebraic representation of the task graph, which the application programmer thus has to provide, while our approach can be introduced in most task-based runtime systems, which use a dynamic task submission API.

[3] It is important to note that the improvement is measured on the total performance and not on the communication part only.

ClusterSs [14] is a task-based runtime system built with a master-slave model: only a master node knows the whole task graph and distributes tasks to slave nodes. Thus the master node can easily detect broadcasts and tells to slave nodes how to handle them. However, no information is published about the optimization of broadcasts.

Charm++ [1] is a parallel programming model relying on tasks called *chares*. It comes with the TRAM subsystem for collective communications, but it is supposed to be used explicitly by the application, which makes its constraints different from our use case.

Legion [6] is a task-based runtime system focused on data locality. Its default scheduling policy is work-stealing, even to another node. No detail is given about a potential communication optimization, but with work-stealing there is not synchronization between different nodes that request the same data.

HPX [11] is a runtime system which executes task on remote nodes *via* active messages. Its API contains routines to explicitly invoke a broadcast involving several nodes.

All in all, other task-based runtime systems either do not optimize broadcasts, or have an API or a DAG representation that allows for explicit use of broadcasts, which are different constraints than dynamic task submission.

7 Conclusion and Future Works

Task-based runtime systems are used to program heterogeneous supercomputers in a scalable fashion. In their DAG, a situation may appear where a given piece of data needs to be sent to multiple nodes. The use of an optimized broadcast algorithm is desirable for scalability. However, the constraints of relaxed synchronization and asynchronous schedulers on nodes make it difficult to use `MPI_Bcast`.

In this paper, we have introduced a *dynamic broadcast* mechanism which makes it possible to use an optimized tree-based broadcast algorithm without needing all the participating nodes know all the other nodes, and without even needing them know they are involved in a broadcast at all. The integration is seamless and nodes receive data with a regular point-to-point receive API. We have implemented the algorithm in NewMadeleine, used it in StarPU, and evaluated its performance on a Cholesky factorization. Results show that our dynamic broadcast may improve overall performance up to 30 % and that it improves scalability.

In the future, we will work on integrating different broadcast algorithms (binary trees, pipelined trees) to get the best performance for all message sizes. We study the implementation of similar algorithm using a generic MPI library, by emulating active messages with a communication thread. Finally, the biggest remaining challenge consists in analyzing finely the global performance of StarPU with regard to networking.

Acknowledgements

This work is supported by the Agence Nationale de la Recherche, under grant ANR-19-CE46-0009.

This work is supported by the Rgion Nouvelle-Aquitaine, under grant 2018-1R50119 *HPC scalable ecosystem*.

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Universit de Bordeaux, Bordeaux INP and Conseil Rgional d'Aquitaine (see https://www.plafrim.fr/).

This work was granted access to the HPC resources of CINES under the allocation 2019- A0060601567 attributed by GENCI (Grand Equipement National de Calcul Intensif).

The authors furthermore thank Olivier Aumage and Nathalie Furmento for their help and advice regarding to this work.

References

1. Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., et al.: Parallel programming with migratable objects: Charm++ in practice. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 647–658. IEEE (2014)
2. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., Tomov, S.: Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In: mei W. Hwu, W. (ed.) GPU Computing Gems, vol. 2. Morgan Kaufmann (Sep 2010), https://hal.inria.fr/inria-00547847
3. Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.: Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. IEEE Transactions on Parallel and Distributed Systems (2017), https://hal.inria.fr/hal-01618526
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 **23**, 187–198 (Feb 2011), http://hal.inria.fr/inria-00550877
5. Aumage, O., Brunet, E., Furmento, N., Namyst, R.: NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks. In: Workshop on Communication Architecture for Clusters (CAC 2007). Long Beach, California, United States (Mar 2007), https://hal.inria.fr/inria-00127356
6. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (Nov 2012). https://doi.org/10.1109/SC.2012.71
7. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Luszczek, P., Dongarra, J.: Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach. Scalable Computing and Communications: Theory and Practice pp. 699–735 (2013-03 2013)
8. Denis, A.: Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests. In: CCGrid 2019 - 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing. Larnaca, Cyprus (May 2019), https://hal.inria.fr/hal-02103700

9. Dongarra, J.: Architecture-Aware Algorithms for Scalable Performance and Resilience on Heterogeneous Architectures (3 2013). https://doi.org/10.2172/1096392
10. Jeannot, E.: Automatic multithreaded parallel program generation for message passing multiprocessors using parameterized task graphs. In: International Conference 'Parallel Computing 2001' (ParCo2001). Naples, Italy (Sep 2001)
11. Kaiser, H., Brodowicz, M., Sterling, T.: Parallex an advanced parallel execution model for scaling-impaired applications. In: 2009 International Conference on Parallel Processing Workshops. pp. 394–401 (Sep 2009). https://doi.org/10.1109/ICPPW.2009.14
12. Pješivac-Grbović, J., Angskun, T., Bosilca, G., Fagg, G.E., Gabriel, E., Dongarra, J.J.: Performance analysis of mpi collective operations. Cluster Computing **10**(2), 127–143 (2007)
13. Sanders, P., Speck, J., Trff, J.L.: Two-tree algorithms for full bandwidth broadcast, reduction and scan. Parallel Computing **35**(12), 581 – 594 (2009). https://doi.org/https://doi.org/10.1016/j.parco.2009.09.001, selected papers from the 14th European PVM/MPI Users Group Meeting
14. Tejedor, E., Farreras, M., Grove, D., Badia, R.M., Almasi, G., Labarta, J.: A highproductivity taskbased programming model for clusters. Concurrency and Computation: Practice and Experience **24**(18), 2421–2448 (2012). https://doi.org/10.1002/cpe.2831
15. Trff, J.L., Ripke, A.: Optimal broadcast for fully connected processor-node networks. Journal of Parallel and Distributed Computing **68**(7), 887 – 901 (2008). https://doi.org/https://doi.org/10.1016/j.jpdc.2007.12.001
16. Wickramasinghe, U., Lumsdaine, A.: A survey of methods for collective communication optimization and tuning. CoRR **abs/1611.06334** (2016), http://arxiv.org/abs/1611.06334