Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System

Marc Sergent*†‡, David Goudin†, Samuel Thibault‡ and Olivier Aumage*
*Inria Bordeaux first.last@inria.fr
†CEA/CESTA first.last@cea.fr
‡University of Bordeaux first.last@u-bordeaux.fr

***Abstract*—The ever-increasing supercomputer architectural complexity emphasizes the need for high-level parallel programming paradigms. Among such paradigms, task-based programming manages to abstract away much of the architecture complexity while efficiently meeting the performance challenge, even at large scale. Dynamic run-time systems are typically used to execute task-based applications, to schedule computation resource usage and memory allocations. While computation scheduling has been well studied, the dynamic management of memory resource subscription inside such runtimes has however been little explored. This paper studies the cooperation between a task-based distributed application code and a run-time system engine to control the memory subscription levels throughout the execution. We show that the task paradigm allows to control the memory footprint of the application by throttling the task submission flow rate, striking a compromise between the performance benefits of anticipative task submission and the resulting memory consumption. We illustrate the benefits of our contribution on a *compressed* dense linear algebra distributed application.**

***Keywords*-memory control; task-based run-time systems; compressed linear algebra; distributed computing**

I. Introduction

Efficiently parallelizing an application involves the two fundamental steps of *exposing* the application parallelism, and of *mapping* the exposed parallelism onto the available computing resources. Both steps require large amounts of expertise from programmers, in all but the simplest cases. Task-based parallel programming models and their associated runtime systems are becoming popular among such programmers, due to their ability to transparently handle the mapping step, while reducing the burden of the parallelism exposition step. Such programming environments rely on the programmer to define elementary pieces of work (the tasks), and to detail how such tasks relate to each other (the dependencies). Still, expressing task dependencies by hand can be error prone. The Sequential Task Flow (STF) task parallelism model [1], [2], [3] therefore simplifies the programmer job even further by automatically building a graph of task nodes and dependence edges from the sequential algorithmic flow of task submissions to the runtime system, by detecting when a task accesses pieces of data modified by previously submitted tasks, in which case the submission ordering between those tasks should be enforced. The resulting graph of tasks is then mapped on the platform to schedule the execution of the application. We have shown in previous work [4], [5] that this can be efficiently extended to distributed-memory by using MPI for instance.

Most of the research work on dependent tasks has until now been dedicated to the execution stage. However, the task submission stage impacts the application performance as well: if the application submits tasks as short, interspersed sequences, the runtime system does not get sufficiently far look-ahead to perform good task scheduling and to anticipate on data transfer requirements to overlap data transfers time with computations. Conversely, if the application submits tasks as long, bulk sequences, the runtime system may itself waste valuable resources especially in terms of memory, notably in the case of distributed memory, for which reception buffers have to be allocated to receive contributions from other nodes, resulting in sub-optimal performances, and even reaching fatal out-of-memory condition. This paper hence studies the possible cooperation between the application and the runtime system to control the task submission flow from a memory consumption point of view, to allow for sufficient parallelism while keeping the memory footprint under control. The main idea is to integrate a throttling mechanism on the task submission flow, and to drive it by monitoring the memory consumption level. When the memory usage exceeds a predefined upper threshold, the throttling temporarily blocks the task submission routine. Once the memory usage falls below a lower threshold as the result of previously submitted tasks running to completion, the task submission routine is unblocked again. This control flow on task submission is made possible thanks to the property of the STF model to be immune to dependence-related deadlocks by design. This does not prevent memory-related deadlocks (as typically prevented by the Banker's algorithm), but as discussed in section V-C, in our MPI distributed-memory context, this is not a problem in practice because temporary buffers used to store contributions from different MPI nodes easily provide memory to release, and the real concern is to control the submission rate to avoid out of memory conditions.

We experiment our proposal by considering a Boundary Element Methods (BEM) [6] code designed at CEA, which

uses a *compressed* dense linear algebra solver where the compression ratio of each matrix tile cannot be estimated *a priori*. This application emphasizes the need for a control mechanism between the task submission flow and the memory consumption level, to be able to run large test cases to completion without wasting prohibitive amounts of memory into grossly over-estimated safety margins.

The contributions of this paper are the following:

- A mechanism to monitor the level of memory subscription induced by a flow of submitted tasks.
- A mechanism to throttle the flow of tasks submission to smooth out the memory subscription peaks induced by burst task submission sequences, drived by the memory subscription monitor.
- The evaluation of these combined mechanisms on a compressed dense linear algebra application, running successfully to completion with a much narrower tolerance gap in terms of memory consumption than without these mechanisms.

This paper is structured as follows: Section 2 presents the STF model and its properties that make such a task submission flow control possible, as well as the STF-based StarPU runtime system we used for our experiments. Section 3 exposes the state of the art. Section 4 introduces our contribution about the control of the memory subscription of StarPU. Section 5 describes the distributed, *compressed* dense linear algebra application we used as the target for the study and the consequences on memory control. Section 6 presents our performance results. Section 7 concludes the paper and gives some prospects of this work.

II. Sequential Task Flow

The Sequential Task Flow class of parallel programming models characterizes task parallelism environments involving two stages running concurrently: a *sequential tasks submission stage*, and a *parallel tasks execution stage*. The submission stage is performed by the application core part, which unfolds its "taskified" algorithm sequentially, submitting the tasks constituting the algorithm one at a time, for subsequent asynchronous accomplishment by the execution stage.

For each submitted task, the application also specifies the pieces of data accessed by the task (`R`: *Read-Only*), as well as those possibly modified by the task (`R/W`: *Read-Write* or `W`: *Write-Only*). Taking into account the pieces of data, their so-called *access modes*, and the sequential task submission order, the execution stage implemented as a runtime system can build a graph of task dependence edges (usually a Directed-Acyclic Graph (DAG)) and use this graph to decide which sets of tasks may safely run in parallel, and which sets of tasks should strictly be executed following the submission order, to preserve the expected application semantics. For instance, Figure 1 shows a StarPU version of the tiled Cholesky factorization algorithm. The *for* loops only submit tasks, without waiting for completion of any of them.

```
for (j = 0; j < N; j++) {
 POTRF(RW,A[j][j]);
 for (i = j+1; i < N; i++)
   TRSM(RW,A[i][j],R,A[j][j]);
 for (i = j+1; i < N; i++) {
   SYRK(RW,A[i][i],R,A[i][j]);
   for (k = j+1; k < i; k++)
     GEMM(RW,A[i][k],R,A[i][j],R,A[k][j]);
 }
}
starpu_task_wait_for_all();
```

Figure 1. Source for the tiled Cholesky factorization in the STF model

A key property of STF is that by construction, no submitted task may depend on the result of an as-yet unsubmitted task. Consequently, the submission and execution stages are loosely coupled: The submission stage may temporarily stop submitting tasks at any point in the task flow without any risk of causing deadlocks; Also, the submission stage may wait for the completion of some batch of submitted tasks at any time, but is not conceptually required to do so, except at the very end of the application execution.

A. The StarPU Task-Based Programming Environment

StarPU is a task-based parallel programming environment and associated runtime system developed by the STORM research team in Bordeaux, France. It targets multicore and heterogeneous, accelerated, computing platforms. StarPU exposes a sequential task flow API to the application. It dynamically schedules and executes submitted tasks on the available cores, following the programmer selected stock scheduling algorithm or a specifically tailored one. When running on an accelerated node, a distributed shared memory (DSM) embedded within StarPU transparently handles data transfers between the main memory space and the accelerators (or GPUs) memory spaces, and manages data replica consistency. The scheduling and DSM engines of StarPU also cooperate together to perform data prefetching and overlap transfers with computations.

Beyond intra-node parallelism, StarPU also provides two supports for interacting with MPI in a distributed context [4], [5]. The *explicit* MPI support defines a set of `starpu_mpi_*` wrappers on top of common MPI routines such as `MPI_Isend` or `MPI_Irecv` that insert proper data dependence edges between computation tasks and MPI requests. When using StarPU's *implicit* MPI support instead, the application has to provide an initial data distribution over the computing nodes. Then, each node submits the *whole* sequence of application tasks identically, and every node is responsible for effectively executing a *subset* of the global set of tasks in accordance with data ownership; for instance, a node executes a given task if and only if it owns the piece of data the task writes to. Since the whole set of tasks

```
struct starpu_data_interface_ops {
  ssize_t allocate ();
  ssize_t free ();
  int pack_data(void **ptr, size_t *count);
  int unpack_data(void *ptr, size_t count);
  ...
};
```

Figure 2. StarPU data interface structure.

is known globally, the StarPU execution engine is able to determine data dependence edges linking two distinct nodes, and then to insert `MPI_Isend` and `MPI_Irecv` calls on each side of such edges as appropriate, in a transparent fashion from the application point-of-view. As only the inter-node dependence edges really matter for StarPU's implicit MPI support to work, the constraint of global knowledge of the task graph can in practice be relaxed significantly: A node really only needs to submit its own tasks, as well as the neighbor tasks at the other end of its own tasks' dependence edges. We have shown in previous work [5] that thanks to these optimizations, while exposing a programming model which is simple, our approach exhibits performance that is competitive with state-of-the-art distributed environments such as ScaLAPACK or the DPLASMA framework [7], [8] over clusters of more than a hundred hybrid nodes, and scalability perspectives seem to reach at least thousands of nodes.

In order to support various data types (vectors, matrix blocks, sparse matrix blocks, etc.), StarPU uses one *data interface* per type of data. This interface defines methods for dealing with the data. Figure 2 shows the methods which will be of interest for this article. The `allocate` and `free` interfaces are used to allocate and release a piece of data. In them, the application can allocate data with the underlying memory allocator of his choice (*malloc*, *hoard* or any other third-party allocator). Some *private data* also allows the interface to store information for its own use, such as the tile size, leading dimension, number of vector elements, etc. For the most simple cases such as vector or matrix blocks, sending and receiving a piece of data is implemented natively inside StarPU with direct `MPI_Isend` and `MPI_Irecv` calls. For more complex cases such as sparse matrix blocks or the BLR-compressed blocks we will discuss in Section V, the `pack_data` and `unpack_data` methods can be defined to describe respectively how to *pack* the piece of data into a contiguous block of bytes (before sending over MPI), and how to *unpack* such block of bytes (after reception from MPI) into the original data content.

Such kind of *interfaces* are usually defined once for good: StarPU provides interfaces for most basic data types, and knowledgeable users can define their own interfaces, which can then easily be reused as such by other users in various applications without additional development. In Section V, we describe how we implemented an interface for BLR-compressed matrix tiles, which can now be easily used by various BLR-based applications.

III. State of the art of distributed task-based runtimes

Beside StarPU, numerous task-based runtime systems have been proposed to drive applications on top of distributed HPC platforms. Some runtimes such as OmpSs/StarSS [2], Kaapi [3], Qilin [9] or others [10] also follow the principle of full run-time task discovery of the sequential task-flow paradigm. Some environments such as Charm++ [11] are based on the message-driven parallelism instead. Others, such as the DPLASMA framework [7], [8], associate a compilation stage to build a parametric representation of the task graph with an execution stage to map the resulting parametric graph on computation units.

The problem of memory-aware task scheduling or workflow mapping has been the subject of theoretical research works [12] as well as some implementations such as Charm++ [13], or the Bounded Memory Scheduling (BMS) model of dynamic task graphs recently implemented in the concurrent-collection based Qt-CnC runtime system [14]. A prototype for a more general resource-aware task scheduling was developed in OmpSs [15]. The topic was also studied with keen interest in the domain of embedded computing, where memory resources may drastically be limited [16], [17]. We refer to Section 4.3 of [18] for a survey of theoretical and practical memory-aware algorithms.

Involving the application in the memory footprint control was explored on the algorithmic side [19], and as a collaborative approach between the application and a runtime system [20]. However, to the best of our knowledge, no support fully handled on the runtime side, in a transparent manner, has yet been proposed. This is the purpose of our contribution.

Scope of our contribution

A well-known issue of task-based runtime systems when scaling up on distributed supercomputers is to regulate the memory resource subscription without a prohibitive performance penalty.

An attempt of collaborative approach between an application and a task-based runtime system to control the memory subscription has been studied by Agullo et *al.* in [18]. However, that work is focused on an application which does not fit in the memory of a *single* computer. It proposes an application-level memory subscription control coupled with the StarPU runtime system, which provides memory consumption feedback to the application thanks to its memory manager engine. Instead, our work aims to control the memory subscription of applications at the runtime system's level and to extend the memory control feature to distributed applications thanks to the STF model.

We discuss in the following section a method based on the STF paradigm to tackle that issue and how we implemented it into the StarPU runtime system.

IV. Controlling the memory subscription

With the property of the STF model that decouples the submission step and the execution step, and guarantees that no submitted task depends on the result of unsubmitted tasks, it is possible to temporarily put the submission step on hold without risking dependence-related deadlocks: a submitted task will always be able to complete. The principle of addressing the memory subscription issue is to anticipatively compute, at the task submission time, the memory space that needs to be booked to guarantee its successful completion when its execution time comes later. If this memory space requirements cannot be known exactly at submission time, an upper bound overestimation should be used instead. The runtime system then adds up this amount to the memory subscribed for all the already submitted and not yet completed tasks, and may temporarily block the task submission flow when the estimation of memory booked by the runtime system reaches a maximum subscription threshold. Once the runtime system detects that enough memory space has been freed by tasks completions, it may resume the task submission flow. We implemented our contribution in the StarPU runtime system which implements the STF model, but any runtime system which can hold the task submission flow without introducing deadlocks (e.g. by respecting the sequential order of the task submission) and can track the memory consumption of each submitted task can implement the mechanism we introduce in this paper.

A. Implementation in StarPU with the memory manager

Our implementation extends the use of the memory manager engine of StarPU for the control of the task submission by introducing the `starpu_memory_allocate` and `starpu_memory_deallocate` functions, to be called by the data interface (as defined in Section II-A) in its `allocate` and `free` methods, i.e. along with actually allocating or deallocating any piece of data. These calls enables StarPU's accounting of the amount of data allocated by the interface. That separation of allocation concern vs accounting concern allows the user to allocate data with his memory allocator, and declare the allocation to StarPU. We then introduce some SLEEP and WAKEUP thresholds for stopping and restarting the task submission. When the submission thread of the application submits a task, an estimation of the memory subscription needed for its execution is computed from the data interfaces of each piece of data involved with that task. The memory manager is then advised of this memory booking with a call to the `starpu_memory_allocate` routine before actually submitting the task. If the memory manager detects that the memory subscription of StarPU would exceed the SLEEP threshold, the submission thread blocks, thus avoiding to overrun the threshold.

While the submission thread is blocked, already-submitted tasks still proceed progressively to execution and eventually free some memory as the result of their completion. The runtime system is notified of this by calls to `starpu_memory_deallocate` from the `free` method of the data interface. When the memory manager detects that the current memory subscription of StarPU has reduced below the WAKEUP threshold, it wakes up the submission thread, which thus resumes submitting tasks. We use different thresholds for SLEEP and WAKEUP to reduce the number of wake-ups and thus the overhead of this mechanism. The SLEEP threshold can be defined by default to the size of available memory on the machine minus a couple of GiB for the operating system, and the WAKEUP threshold be defined to 90% of the SLEEP threshold.

The `starpu_memory_allocate/deallocate` functions have been used by [18] to ensure that enough memory space is available for the execution of a given set of tasks that need to allocate temporary pieces of data of known size. That was however integrated into the application loop, while we here propose to encapsulate the implementation inside the *data interface*, thus relieving the application from code changes. Moreover, in the case of our targeted compressed dense linear algebra application, there is no temporary data when executing on a single node: data is assembled completely before starting submitting tasks, and computation is performed in place. However, when employing several distributed nodes instead, tasks do indeed expect contributions from other nodes, which we will call *remote data* in this paper. The resulting incoming messages have to be stored in temporary buffers, whose memory footprint should be accounted for as well, as discussed in the next section.

B. The distributed case: memory footprint of incoming remote contributions

To store remote data on the receiving node which will execute the task, StarPU automatically allocates a buffer. Since the `allocate` method of the data interface calls `starpu_memory_allocate` along with allocating the buffer, the StarPU memory manager is made aware of this additional consumption of memory, and will thus eventually *block* the submission thread once a lot of buffers have been allocated this way.

By default, StarPU will *cache* the received remote data [4], so that when several tasks need the same remote data, only one communication is performed. As a consequence, the corresponding buffers will be kept allocated. The application can, while submitting tasks, make a call to `starpu_mpi_cache_flush` to specify when a given piece of data will probably not be used by the application algorithm in the close future. For instance, Figure 3 shows

```
for (j = 0; j < N; j++) {
 POTRF(RW,A[j][j]);
 for (i = j+1; i < N; i++)
   TRSM(RW,A[i][j],R,A[j][j]);
 starpu_mpi_cache_flush(A[j][j]);
 for (i = j+1; i < N; i++) {
   SYRK(RW,A[i][i],R,A[i][j]);
   for (k = j+1; k < i; k++)
     GEMM(RW,A[i][k],R,A[i][j],R,A[k][j]);
   starpu_mpi_cache_flush(A[i][j]);
 }
}
starpu_task_wait_for_all();
```

Figure 3. Source code of tiled Cholesky with flush notifications.

how such calls can be quite naturally added to the tiled Cholesky factorization. In that case the buffer containing the cached remote data will be released once all the previously-submitted tasks referencing this data have completed. In the Cholesky example of Figure 3, diagonal `A[j][j]` tiles can for instance be flushed once the corresponding TRSM tasks have completed. The `free` method of the data interface calls `starpu_memory_deallocate` along with the actual release so that the memory manager can then possibly wake the submission thread since memory has been released.

To summarize, this mechanism allows to accurately control the flow of task submission from the application, according to how much memory gets used for the buffers required for receiving remote data from MPI. In the end, StarPU lets the application submit as many tasks as possible (and thus achieve most parallelism), while taking into account the entailed memory consumption.

C. Dealing with remote data of unknown size

In our compressed dense linear algebra case, the size of the remote data received *via* MPI may be unknown to the receiving side, since the receiving node does not know the compression ratio achieved by the sending node for a given tile. In that situation, the data interface can announce an overestimation to StarPU: for instance, a possible worst case upper bound estimation may be obtained from considering that a compressed tile cannot get larger than an uncompressed tile. The data interface will thus pass the size of an uncompressed tile to `starpu_memory_allocate` at submission time, to remain on the safe subscription side. When `MPI_Irecv` completes, the actual size of the compressed tile gets known, and in the `unpack` method the data interface calls `starpu_memory_deallocate` with the difference, to account for the amount of memory which did not get used in the end.

As a result, since initially a lot of MPI requests are pending on the receiver side, we first get an overestimation of the memory that will be required to receive them. That overestimation however gets progressively fixed as MPI requests complete, thus releasing memory subscription, and thus allowing more tasks to be submitted by the application for better parallelism, which entail more MPI requests with over-estimations of the memory required to receive them, and so on.

D. Fragmentation issues

Allocating data with varying sizes would often lead to allocation fragmentation, which would waste memory, and thus make our memory accounting underestimate the actual memory usage. In our study, fragmentation was very limited and has thus not posed problems. If fragmentation becomes a concern, a way to deal with it would be to periodically request statistics from the allocator, more precisely the ratio between the amount of allocations vs the corresponding memory usage, and multiply the StarPU memory accounting by this ratio, thus fixing the estimation of memory use for the allocated buffers.

E. Discussion

To summarize, using the `starpu_memory_allocate` and `starpu_memory_deallocate` memory accounting functions allows StarPU to guarantee a safe execution of the application, by blocking and restarting its submission loop thanks to thresholds.

For all applications for which the size of the data does not change during the execution (the dense Cholesky factorization falls within this scope for example), StarPU simply needs to deal with the remote data received *via* MPI, where the size of each piece of data is completely known. Since the behaviour of the memory consumption is similar to the one presented on Figure 4 (and will be discussed in Section VI-B), we do not present those experimental results due to lack of space.

However, some applications may manipulate data which grows in size over the execution, e.g. matrix fill-in. In some cases, this fill-in can be easily estimated, and a static overestimation exposed to StarPU instead of the growing real size, to keep on the safe side.

In other cases, the fill-in can not be predicted accurately, and overestimation would have to be very gross, leading to a poor use of available memory. This is the case we discuss in the next section, taking a tiled Cholesky factorization with *BLR* compression as a case study, to show how our contribution allows to strike a compromise between performance and memory consumption for such applications.

V. Task-Based BLR Solver

A. BLR: dense linear algebra and compression techniques

The need for scalable direct solvers in dense linear algebra faces the memory footprint issue. Nowadays, the necessity of storing the full dense matrix in memory limits the maximum size of problem that can be computed at least as much as the computing time.

An algorithmic solution has been developed by Bebendorf to approximate boundary element matrices in [21]. The idea

is that dense blocks of matrices can be represented as a product of two tall and skinny matrices. Some of the numerical precision is lost in the process, but the loss is controlled by a cut factor called ϵ, used during the compression step. A dedicated arithmetic for such compressed matrices has also been introduced by Bebendorf on this paper. This arithmetic relies on re-compression steps during the update of the trailing matrix to keep it as compressed as possible during the execution. However, after the re-compression step, the size of each updated piece of data data can change, and thus the pair of matrices that represents each dense block must be reallocated to match that new size.

An implementation of this type of solver exists at CEA for their applications. For coupling this solver with the StarPU runtime system, we use the Chameleon software, which is a collection of solvers on top of task-based runtime systems developed by the HiePACS research team at Bordeaux, France. We developed in Chameleon a StarPU-compliant data interface (as defined in Section II-A) for compressible matrices. This allowed an easy coupling of CEA's solver with the StarPU runtime system, which is one of the runtime systems supported by Chameleon. We then simply kept the same tiled Cholesky algorithm source code, as shown on Figure 3, but extended the underlying kernel into using the CEA-provided BLAS kernels which can operate on various combinations of uncompressed and BLR-compressed data.

B. Issues entailed by using BLR

The size of the compressed tiles will typically tend to grow in an unpredictable way while the application algorithm progresses, due to data contributions from other tiles, notably from the tiles which could not be compressed. This means that `starpu_memory_allocate` will have to be called by the data interface to account for the additional required space, to delay the submission of tasks which would have used that space. In compressed linear algebra algorithms, the contributions coming from other MPI nodes are not modified, so only the local tiles will grow. We may still end up overflowing the memory, if the inflation of local tiles as tasks complete is bigger than the amount of the cached MPI remote data that can be released when tasks complete.

This issue can be avoided altogether safely by overestimating the amount of memory needed for local tiles, by assuming for instance that they are all uncompressed and making the data interface announce the corresponding size instead of the actual size of the compressed block. That will however make a poor use of the available memory, and StarPU will have to let the application submit fewer tasks at the same time, thus reducing the available parallelism.

On the other hand, since tiles have various compression ratios, tasks will have varying completion time: a GEMM (matrix-matrix product) task with two dense tiles as input and a compressed tile as output will take roughly one order more time to execute than a GEMM task with only compressed tiles. It is thus crucial to let the application submit enough tasks in advance for the runtime scheduler to be able to reorder them to compensate the irregularity.

The user could increase the amount of available memory by running the computation on more MPI nodes, but in practice this is not reasonable because the goals of using compression is not only to reduce the amount of computation, but also to reduce the number of nodes needed to fit the computation in memory, as will be discussed in Section VI-B. Assuming that tiles are uncompressed would for instance lead typically to a $10\times$ overestimation of the required number of MPI nodes.

We are thus aiming at a compromise between achieving good performance by submitting enough tasks in advance, while using as few MPI nodes as possible, and guaranteeing the termination of the execution as much as possible.

C. A compromise approach

Generally speaking, it is not possible to know how much fill-in will happen overall, because matrices can be arbitrarily complex and it is possible that the computation ends up filling the matrix in completely. Users can thus prefer to decide to play on the safe side by assuming that local tiles are uncompressed when making memory subscription, even if that will probably lead to unused memory. They can then choose between executing on few MPI nodes (and thus little available memory, thus little available parallelism as well as lower achieved performance), or spending more MPI nodes on the computation to get the result faster.

With the matrices used in practice, the computation however does not behave so badly, and application users may have a rough idea of how much fill-in will happen overall with the matrices being worked on, and assume e.g. that local tiles will not grow more than a $2\times$ factor. This is for instance true in the experimentation conducted in Section VI (ratio between *initial memory* and *current memory*). We can then simply apply this factor on the size initially announced by the data interface for the compressed tiles, it is a much better overestimation than using the uncompressed size mentioned in the previous paragraph. If there can be a guarantee on this inflation factor, execution is then guaranteed to complete without overflowing the memory. The application may even refine the estimation during the execution: when it knows a better estimation for the maximum future size of a tile, the data interface can be made to call `starpu_memory_deallocate` to release the difference, thus allowing for more parallelism.

In the case of compressed linear algebra, we however do not have any strong guarantee since the inflation of tiles over the whole application execution is not formally bound. In practice, though, the average inflation factor per task is very small. There may be tiles which inflate noticeably, but on average the tile expansion is smaller than the amount of

data released when flushing the cached remote data from other MPI nodes once tasks complete.

In that case, we can get bolder, and not overestimate the size of local compressed tiles any more, and when their size increases, the data interface calls `starpu_memory_allocate` to declare the additional usage. This will make StarPU prevent the application from submitting more tasks for yet more time, until that increase in local tile memory consumption gets compensated by the release of cached remote data from other MPI nodes. We just have to lower the SLEEP/WAKEUP thresholds slightly to have enough memory margin in between. In practice this works really well, as will be seen in the experimental section, e.g. on Figure 4: the amount of subscribed memory nicely evolves between the two SLEEP/WAKEUP thresholds. StarPU can then make fairly good use of all the available memory for caching MPI remote data and allowing the application to submit a lot of tasks at a time.

If users underestimated the inflation factor, i.e. they did not use enough MPI nodes for the computation, the inflation of the local tiles may consume more and more memory until filling the total memory. In that case StarPU will warn the user, and still try to continue the execution, but a single task at a time, to minimize the memory footprint. Resorting to out-of-core mechanisms would be the only way for the runtime system to support this case, and while supported in StarPU, it is not an option on the platform used for the studied test case and is outside the scope of this paper.

In the end, since the memory subscription mechanism is encapsulated once for good inside the data interface, application code can remain as simple as Figure 3, and the user can decide between full safety, better performance, and the number of MPI nodes that will have to be reserved for the computation.

VI. Experiments

To validate the compromise approach described in the previous section, we analyze the memory behaviour of a tiled Cholesky factorization on BLR-compressed matrices, typically used at CEA for simulations, on a cluster of several dozens of nodes. We show that the amount of memory consumed by the application during the execution is successfully constrained inside the memory bounds given to the runtime system by the user. We show that the throttling of the task submission flow does not alter the performance of the application compared to other runs of the same application without any limitations on the task submission flow. Finally, we show that our memory subscription control mechanism allows a better memory filling of computing nodes without risking out-of-memory scenarios, which allows to use fewer nodes to perform the computation. We have also applied our approach on various linear algebra applications which use another compression strategy, H-matrices, and a similar approach on sparse linear algebra [18] for the QR factorization with the same conclusions; we here only present results for the BLR-compressed Cholesky case which is much simpler to analyze.

Table I
Parameters used for each setup

	Matrix size	SLEEP threshold	WAKEUP threshold
Setup 1	201k	20 GB	18 GB
Setup 2	450k	18 GB	16 GB

A. Experimental context

We conducted our experiments on the TERA100 Hybrid cluster [22] of the CEA. Each computing node is composed of 2 Quad-Core Intel Xeon E5620 CPUs running at 2.4 GHz, and we used between 9 and 81 nodes for our experimental campaign.

For our experiments, we coupled the distributed multicore implementation of the BLR-based dense double-precision complex tiled Cholesky direct solver of the CEA with the Chameleon software.

The application's parameters have been tuned as follows : the size of a block is 512, the chosen MPI data layout is the two-dimensional block cyclic distribution, and we only use square grids of processes. The block size choice is a trade-off between small blocks to compress blocks as much as possible and big blocks to fully exploit the computing units. Square grids of processes with the two-dimensional block cyclic distribution allow to minimize the number and volume of communications and to balance them over all the nodes for dense linear algebra [23]. However, the BLR-based Cholesky factorization holds most of its dense blocks on the diagonal of the matrix. This means that extra-diagonal nodes will receive more dense blocks, and thus as much amount of data, than diagonal nodes. In summary, square grids of processes with the two-dimensional block cyclic distribution is a rather challenging case in terms of volume of communication for the BLR-based Cholesky factorization.

We used the *prio* scheduler of StarPU which allows the application to give priorities to tasks at submission time. We designed a set of priorities for each task of the BLR-based Cholesky factorization to enhance the performance of the application, but how we designed them is out of the scope of this paper.

We define two different setups that use different matrix sizes and memory thresholds in Table I. The first setup is used to show the behaviour of the memory control of StarPU and prove its feasibility, while the second is used to study the scalability of our contribution.

B. Results

Figure 4 shows the behaviour of the memory subscription of StarPU and the resulting memory footprint of the process (which is shown by the *RSS* curve) with and without the

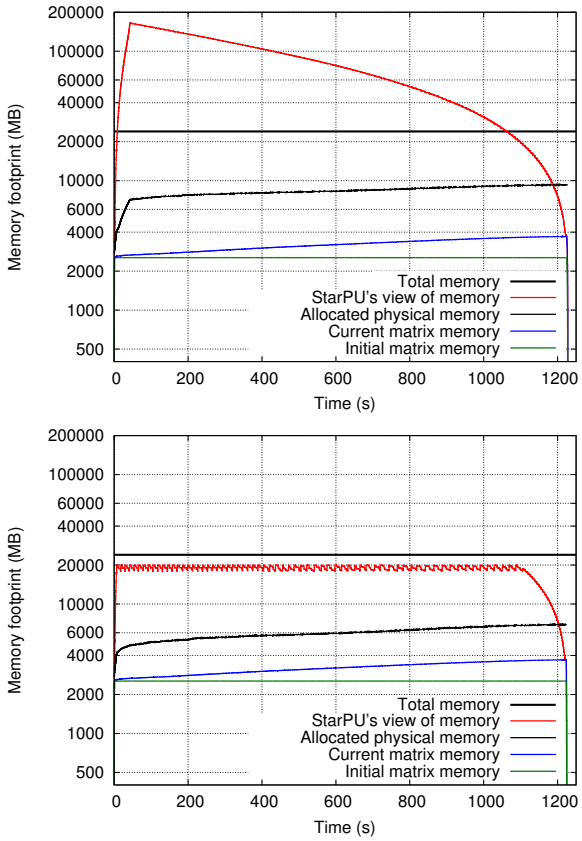

Figure 4. Memory state of node 1 during the execution on 9 nodes on Setup 1, without (top) and with (bottom) memory control

control of the submission task flow on the Setup 1 and 9 computing nodes. We chose to show the plot of node 1 since such extra-diagonal node receives more remote data than diagonal nodes on this BLR-based solver, and thus pressures the memory as much. The *initial memory* curve shows the amount of memory registered to StarPU at the beginning of the application. The *current memory* curve shows how the memory consumption of local data fluctuates during the execution. Thus, the gap between this curve and the *initial memory* curve represents the memory consumption caused by the filling of the matrix. The *overestimation* curve shows the memory subscription known by the memory manager of StarPU, with the overestimation of not-yet-received remote data. The *total memory* line shows the amount of memory available on the node, which must not be reached by the *RSS* curve which represents actual memory use.

We see on the top graph of Figure 4 that without memory control, the *overestimation* curve greatly exceeds the total memory of the machine. Thus, there is no guarantee that the memory will not overflow until the end of the execution. We can also observe that the *RSS* curve of the top graph stops around 9.5 GB of memory consumption while the one of the bottom graph only goes up to 7 GB.

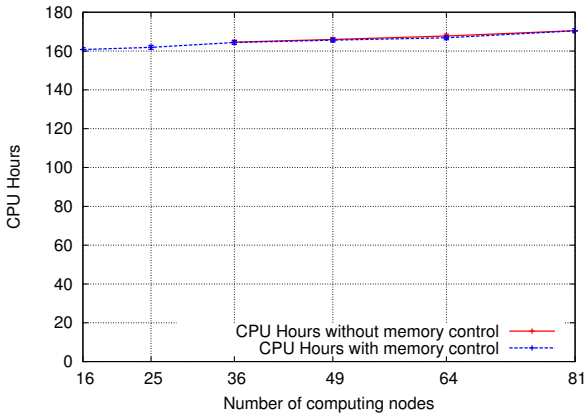

Figure 5. CPU Hours from 16 to 81 nodes on Setup 2

In the top (non-controlled) case, all the memory allocations needed for remote data are performed as soon as possible, which is why the *RSS* curve's gradient is steep at the beginning of the execution. After that, the gradient of the curve follows the growth of local data, which is caused by the filling phenomenon of the local blocks of the compressed matrix.

Instead, the bottom graph shows that with the memory subscription control and well-chosen thresholds (as discussed in IV-A), the memory footprint of the process is kept in check. The growth of the memory footprint of the process is smoother because the memory allocations for remote data are spread throughout the execution. This allows the memory allocator to re-use the memory space used by previously received pieces of data for newly received ones.

Finally, we note that the execution times are almost the same, which lets us assume that the memory control does not affect the performance of the execution.

This assumption is confirmed on Figure 5, which presents the performance results in terms of CPU.hours from 16 to 81 computing nodes on Setup 2.

On the first hand, we can observe that the performance results with and without memory control nicely scale similarly. The amount of consumed CPU.hours for an execution on 81 nodes is only about 6% higher than for an execution on 16 nodes, and the effect on performance of using memory control, which would be thought to harm parallelism, is negligible. This shows that the throttling of the task submission is sufficiently relaxed not to hinder the performance of the computation.

On the other hand, we can see in this plot that the curve without memory control has no points for 25 and 16 nodes. This means that the runs reached an out-of-memory condition, which will be discussed in the next paragraph. Furthermore, those runs with memory control required slightly less CPU.hours than those with 36 nodes or more. This shows that our contribution allows the user

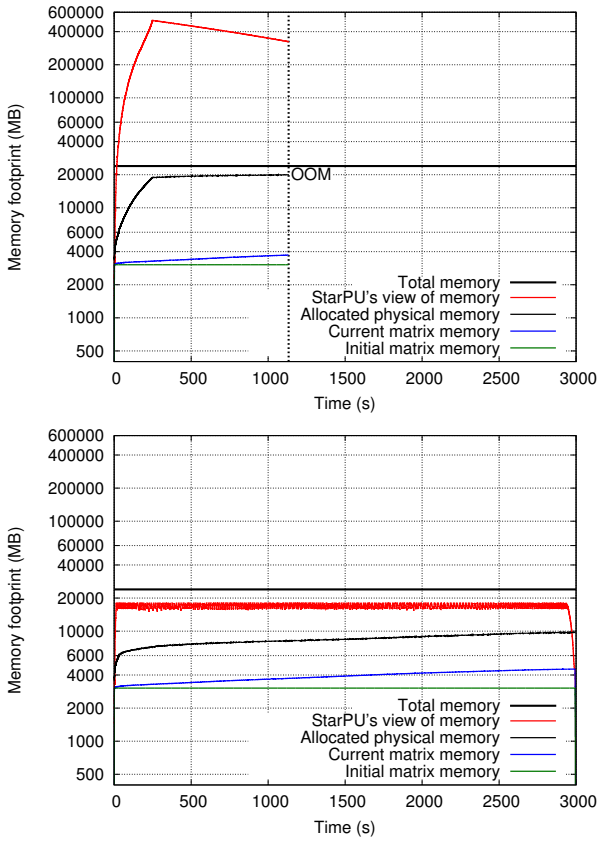

Figure 6. Memory state of node 1 during the execution on 25 nodes on Setup 2, without (top) and with (bottom) memory control

to run their solvers on fewer nodes than before while consuming less CPU.hours, which is really interesting in an industrial context where access to computational resources is constrained, in particular large cluster reservations.

Figure 6, which is formatted the same way as Figure 4, shows what happens to the memory subscription when the execution faces an out-of-memory condition without memory control, and how the execution successfully goes to its end with memory control. These runs have been made with 25 computing nodes on Setup 2. On the top plot, StarPU submits all the tasks as soon as possible, and allocates the memory space needed to receive all the remote data that those tasks will need in the future, which causes a quick growth of the *RSS* curve. When all the data is finally allocated, the growth of local data caused by the filling of the compressed matrix slowly fills up the remaining memory space until the out-of-memory killer is called when the RSS reaches up to 20.8 GB. With the memory control, the memory allocations for receiving remote pieces of data are diluted throughout the execution. Thus, the allocator can re-use the memory space of previously allocated pieces of data for newly allocated ones in a more effective way than without memory control. The bottom plot shows that, indeed, the execution with memory control goes to its end, with a maximum memory footprint of 10GB.

C. Experimental conclusion

These experimental results enlightens three important points of our contribution. 1) Memory control efficiently bounds the memory consumption and limits the memory footprint of the application, as shown in Figures 4 and 6. 2) Memory control does not cause any performance decay at high scales, as shown in Figure 5. 3) Memory control allows the successful execution of bigger test cases on a small number of nodes while slightly reducing the required CPU.hours to perform the run, as shown in Figure 5.

VII. Conclusion and Prospects

This paper proposed a runtime-level control of the memory subscription growth during application execution, based on throttling the task submission flow. The runtime system monitors the memory subscription of the application by maintaining an account of the memory space required for each piece of data of each submitted task. When the exact memory footprint of a piece of data is not yet known at the time of submission, the accounting is temporarily given an overestimated quantity, later revised when the exact data size gets settled. With that information, the runtime system can decide to stop the task submission flow when the memory subscription reaches a maximum threshold, and to restart submitting tasks when enough memory space has been freed by the application, without risking dependence-related deadlocks thanks to the STF model which imposes a sequential ordering for task submission. This mechanism guarantees the successful execution of the application within the memory bounds given by the user.

To deal with a compressed dense linear algebra application for which the size of most of the pieces of data tend to grow in ways unpredictable prior to the execution, we proposed a compromise approach between performance and successful execution within the memory bounds. We validated this approach by coupling a BLR-based tiled Cholesky direct solver of the CEA with the StarPU runtime system. We demonstrated on a real-life compressed matrix of the CEA that the memory consumption growth is constrained thanks to the runtime memory control instead of risking out-of-memory. Our results also showed that throttling the task submission flow has no performance impact when using dozens of nodes. Furthermore, the memory control allows to execute this application on fewer nodes than before, while slightly decreasing the amount of consumed CPU.hours. This approach has also been successfully used for dealing with other compressed linear algebra applications (H-matrices).

The contribution described here considered applications for which the amount of data allocated on each node does not exceed the available memory on the machine. Using

out-of-core techniques to overcome this limitation is part of ongoing work in StarPU. Another way we intend to explore to address this issue would be to combine memory-aware task scheduling heuristics with the runtime-level memory control we proposed.

In this paper, we suppose that the sequential submission order of the tasks by the application guarantees that the memory consumption of the application cannot reach an out-of-memory condition if all the tasks are executed sequentially. This is not always the case, as many applications divide their work in assembling (allocating) and computation (deallocating) phases. To address this issue, an idea would be that, instead of blocking the task submission flow when there is no memory space available, to force allocation tasks to wait for memory and continue to submit tasks until deallocation tasks are submitted and executed, which then unlocks the waiting allocation tasks.

REFERENCES

[1] 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, Special Issue: Euro-Par 2009*, 2011.

[2] J. Bueno, X. Martorell, R. M. Badia, E. Ayguad, and J. Labarta, "Implementing ompss support for regions of data in architectures with multiple address spaces," in *International conference on Supercomputing*, 2013.

[3] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Locality-aware work stealing on multi-cpu and multi-gpu architectures," in *Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, 2013.

[4] C. Augonnet, O. Aumage, N. Furmento, S. Thibault, and R. Namyst, "StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators," INRIA, Rapport de recherche RR-8538, May 2014. [Online]. Available: http://hal.inria.fr/hal-00992208

[5] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Harnessing Supercomputers with a Sequential Task-based Runtime System," INRIA, Tech. Rep. [Online]. Available: http://starpu.gforge.inria.fr/starpu-mpi.pdf

[6] S. Sauter and C. Schwab, "Boundary element methods," in *Boundary Element Methods*, ser. Springer Series in Computational Mathematics, 2011, vol. 39.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczezek, A. YarKhan, and J. Dongarra, "Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA," ICL, UTK, Tech. Rep., 2010.

[8] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical dag scheduling for hybrid distributed systems," in *International Parallel and Distributed Processing Symposium*, 2015.

[9] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar, "Scheduling concurrent applications on a cluster of cpu-gpu nodes," in *International Symposium on Cluster, Cloud and Grid Computing*, 2012.

[10] T. Beri, S. Bansal, and S. Kumar, "A scheduling and runtime framework for a cluster of heterogeneous machines with multiple accelerators," in *International Parallel and Distributed Processing Symposium*, 2015.

[11] C. Mei, G. Zheng, F. Gioachin, and L. V. Kal, "Optimizing a parallel runtime system for multicore clusters: A case study," in *TeraGrid'10*, 2010.

[12] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien, "Parallel scheduling of task trees with limited memory," *TOPC*, vol. 2, no. 2, 2015.

[13] I. Dooley, C. Mei, J. Lifflander, and L. V. Kale, "A study of memory-aware scheduling in message driven prallel programs," in *International Conference on High Performance Computing*, 2010.

[14] D. Sbîrlea, Z. Budimlić, and V. Sarkar, "Bounded memory scheduling of dynamic task graphs," in *International Conference on Prallel Architectures and Compilation*, 2014.

[15] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell, "Resource-aware task scheduling," in *Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*, 2013.

[16] P.-A. Arras, D. Fuin, E. Jeannot, A. Stoutchinin, and S. Thibault, "List scheduling in embedded systems under memory constraints," in *International Symposium on Computer Architecture and High Performance Computing*, 2013.

[17] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Memory analysis and optimized allocation of dataflow applications on shared-memory mpsocs," in *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 2014.

[18] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems," IRIT, Tech. Rep., 2014.

[19] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplicatoin and lu factorization algorithms," in *International conference on Parallel processing Euro-Par*, 2011.

[20] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Multifrontal qr factorization for multicore architectures over runtime systems," in *Euro-Par 2013 Parallel Processing*, 2013.

[21] M. Bebendorf, "Approximation of boundary element matrices," *Numerische Mathematik*, vol. 86, no. 4, 2000.

[22] "TERA-100 Hybrid," Website, 2015, http://www.top500.org/system/177460.

[23] J. Dongarra, R. Van De Geijn, and D. Walker, "A look at scalable dense linear algebra libraries," in *Scalable High Performance Computing Conference, 1992. SHPCC-92, Proceedings.*