DuctTeip: A TASK-BASED PARALLEL PROGRAMMING FRAMEWORK FOR DISTRIBUTED MEMORY ARCHITECTURES*

AFSHIN ZAFARI†, ELISABETH LARSSON†, AND MARTIN TILLENIUS†

Abstract. Current high-performance computer systems used for scientific computing typically combine shared memory compute nodes in a distributed memory environment. Extracting high performance from these complex systems requires tailored approaches. Task based parallel programming has been successful both in simplifying the programming and in exploiting the available hardware parallelism. We have previously developed a task library for shared memory systems which performs well compared with other libraries. Here we extend this to distributed memory architectures. We use a hierarchical decomposition of tasks and data in order to accommodate the different levels of hardware. Our experiments on implementing distributed Cholesky factorization show that our framework has low overhead and scales to at least 800 cores. We perform a comparison with related frameworks and show that DuctTeip is highly competitive in its class of frameworks.

Key words. Task-based parallel programming, distributed memory systems, high performance computing, scientific computing, hierarchical decomposition, data versioning.

AMS subject classifications. 65Y05, 65Y10, 68Q10

1. Introduction.

Parallel programming is a crucial step to utilize the available processing capabilities of a computer system to make a program run faster and/or to solve larger problems. While sequential programming is widely adopted in many industrial and academic fields, parallel programming has reached a more limited audience. This is mainly because parallel programming (historically) has required a deeper technical knowledge of the interactions between the software and the computer architecture. These difficulties increase when a single computer is not sufficient to solve a large problem, and distributed memory computer systems are needed. This is often the case in high-performance scientific computing. A large community of researchers are investigating how to construct programming models that are easy to use for application programmers and that efficiently exploit the hardware capabilities. Differences in applications, variations in computational and storage demands, and evolving technologies make it challenging to define a single optimal approach. In this paper, we focus on task-based parallel programming.

1.1. Motivation.

In scientific computing, there are many large scale applications that need to run in parallel on distributed computer systems due to the large volumes of data and computations that are involved. Developing software for such applications for (heterogeneous) high-performance computer systems requires a large effort, and expert knowledge in parallel and high performance computing. Dependency-aware task based parallel programming, as a method to reduce programming effort while retaining performance, has been very successful for shared memory machines [18, 20, 3, 12, 23]. The key to success of this programming model is that it allows the programmer to write a sequential code expressed in terms of tasks together with information about how these tasks access the shared data. Then a run-time provides efficient parallel execution of the tasks while respecting the data dependencies.

*This work was supported in part by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

†Uppsala University, Department of Information Technology, Box 337, SE-751 05 Uppsala, Sweden (firstname.lastname@it.uu.se)

In the previous paper [23], a highly efficient task parallel framework for shared memory systems called SuperGlue was developed. In this paper, we enable the same type of task based programming style for distributed systems while maintaining similar performance, see also the preliminary work [26].

1.2. Background. In task based parallel programming, there are different ways of handling the dependencies between tasks in different frameworks and/or language extensions. In Cilk [6] or Cilk Plus [11], for example, the user defines and controls the parallelism by writing a sequential program in a fork-join style. In Intel Threading Building Blocks (TBB) [14], a set of C++ templates are provided through which the user can implement the task graph of an algorithm, in which nodes are tasks, and edges show the successors of tasks. The scheduler evaluates the task graph and executes the tasks in parallel. The style of programming in these frameworks supports the creation of recursive and dynamic child tasks. In OpenMP 4.0 [19], regions of a sequential code are marked as tasks by means of compiler directives. Data that are shared or private between tasks have to be explicitly annotated by the user via specific directives. By using the `depend` clause in a `task` directive, the user can specify dependencies of type `in`, `out` and `inout`, but only between sibling tasks.

In distributed memory environments, the partitioned global address space (PGAS) [5, 1] family of frameworks (or languages) provide a global view of the distributed memory to the program, such that the distributed memory space can be accessed as shared memory. The run-time handles the required synchronization and communication transparently. Although the parallel programming effort in this model is reduced to programming for shared memory, the main challenge of writing an efficient parallel program is still present. Ongoing research to address this problem involves incorporating a shared-memory parallel framework into the PGAS system to provide another level of transparency for the user.

In the shared memory task parallel framework SuperGlue [23], the dependencies between tasks are inferred by their accesses to data. To ensure the correct order of task execution, data are equipped with attributes called *versions* in the run-time system. The version number records every data access and is used for handling read-after-write and write-after-read dependencies. Whenever a task is submitted to the run-time, it will wait until the versions of all its input/output data are ready and whenever a task finishes, the versions of the input/output data are updated. All the ready-to-execute tasks can be executed in parallel by the available processors. The same versioning system is used in our extended framework for distributed memory architectures, DuctTeip. The details are explained in Section 2, but here extended to two (or more) levels of tasks and data in a hierarchical setting.

1.3. Related work. In task based parallel programming models, common issues are how to define tasks, track dependencies, schedule tasks, and execute tasks. To address these issues, the existing frameworks use different approaches that can be categorized in terms of static or dynamic regarding task creation and submission, centralized or decentralized with respect to scheduling, and single- or multi-level for the partitioning of tasks and data. There are also more overarching issues such as resource contention [24], energy efficiency [17], and particularly for distributed systems, fault tolerance [9].

Among task parallel efforts with support for distributed memory, StarPU [3, 2], Cluster OmpSs [22], DaGuE [8], and PaRSEC [7], as well as DuctTeip, have a similar view of tasks and task graphs, while the programming models implemented

by Charm++ [16, 15] and Chunks and tasks [21] have some differences regarding communication and data objects.

In StarPU and Cluster OmpSs, tasks can be defined by attaching compiler directives to function definitions in the code. In this case, the tasks and their data accesses are defined statically at compile time. DaGuE and PaRSEC uses an intermediate language for describing an algorithm in terms of tasks working on tiles of data. This description is then given to an optimizer that finds the optimal task schedule and translates the schedule into a C program. Also in these frameworks, the tasks are defined statically

In the StarPU and PaRSEC frameworks, any process can create and submit tasks, whereas Cluster OmpSs employs centralized task creation and submission with one master process that submits tasks to all other processes. StarPU, DaGuE, and PaRSEC are as default single level, but hierarchical partitioning could be implemented with an additional programming effort, while in Cluster OmpSs partitioning of data and tasks can be performed internally by the run-time. In a more direct way, Charm++ and also StarPU (as an alternative method) allow the programmer to use programming constructs (such as classes or APIs) to define tasks and dependencies. This is also the way DuctTeip is implemented. In the Chunks and Tasks framework [21], tasks and chunks (data) are hierarchically defined by inheriting from corresponding C++ classes. A program in this model, is written in sequential form and can be executed using a single or multiple run-time processes.

All these frameworks and approaches successfully hide the complicated efforts of dependency tracking, performing communication, scheduling and executing tasks in parallel. However, maintaining scalability, and performing dynamic task generation and load balancing are still challenging requirements.

To meet these requirements, we propose a new approach for task based parallel programming in distributed systems implemented in the DuctTeip framework. Its potential benefits have also been explored in our earlier paper [26]. Key features of our approach are that it uses data versions for dependency tracking, it uses a hierarchical decomposition of tasks and data, and it uses a local decision model. Similarly to other frameworks, DuctTeip uses overlapping communication and computations and an MPI-Pthreads hybrid parallelization.

The paper is organized in the following way: The programming model is described in detail in Section 2, a description of the framework from the user perspective is given in Section 3. More technical aspects of the framework design are discussed in Section 4. In Section 5 the performance and scalability of DuctTeip is investigated and compared to that of similar frameworks for distributed task based parallel programming. Finally, the results are discussed and some conclusions are drawn in Section 6.

2. The DuctTeip task-based parallel framework. As mentioned in section 1, our proposed method of parallel programming in distributed environments employs a hierarchical approach to extraction of parallelism and parallel execution. The corresponding hierarchical target architecture is a distributed memory system (cluster) of shared memory computational nodes. A key idea in this approach is to use large data partitions for communication and small data partitions for computations. Communicating a single large data block allows for reducing the latency overhead of the communication compared with multiple smaller messages. Then locally, further decomposing the large data into smaller parts increases the cache locality of the data in the local computations.

In order to facilitate the description of the hierarchical model, we introduce the

concept of levels, where level 0 is the top level (root), and level L is the finest level (leaves). The typical data at level 0 consists (symbolically) of the unpartitioned data structure and tasks submitted at level ℓ works on level $\ell+1$ data. Data can of course in general be created and allocated at any level.

In the current implementation, three data levels are used. In the code, the distributed memory framework DuctTeip is used at level 0 (working on level 1 data), and the shared memory library SuperGlue is employed at level 1 (working on level 2 data). For an illustration, see Figure 2.1. In the discussion of data and tasks below, we will describe the general case with L data levels.

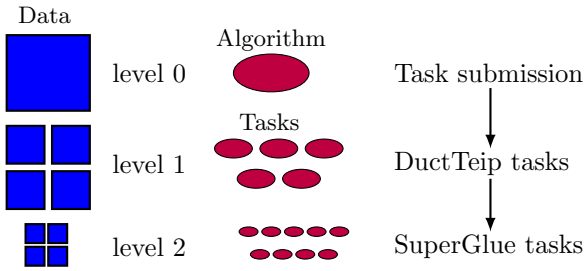


Fig. 2.1: Schematic illustration of hierarchical data and tasks in the currently implemented three level model.

In the following subsections, we explain the hierarchical approach for defining data and tasks and how it affects the dependency tracking, the communication, the task execution and the termination issues in the context of distributed parallel programming.

2.1. Hierarchical decomposition of data and tasks. A shared data structure within the parallel framework is declared at level 0 as $A_0(N, P_G)$, where the first parameter $N = (N_0, \ldots, N_L)$ contains information about the size of the data structure and how it should be split into smaller parts at each level in the data hierarchy, and the second parameter P_G defines the layout of the process grid. Based on the topology of the process grid and a distribution scheme, the ownership of the distributed level 1 data is determined, and each part of the data is allocated locally by its respective owner.

At levels $\ell > 0$, the data $A_\ell(i)$, where the index $i = (i_1, \ldots, i_d)$ distinguishes between the data elements at that level, consist of (repeated) d-dimensional splittings of the level 0 data. The data ownership is inherited from the location of the ancestral level 1 data.

As described in more detail in [23], accesses to the shared data structure are managed through data handles. When a DuctTeip data structure is initiated, handles are created for the data partitions at each level. Handles are objects that track the versions of their data as well as host request queues for particular versions of the data.

For (dense) matrices, defining an efficient storage scheme and selecting methods for splitting into smaller blocks are relatively straightforward. The current implementation assumes matrix data. However, the data interface is intended to work also for more exotic data types such as for example tree structured data. In this case, also packing and unpacking routines are needed for efficient communication of data partitions at the distributed levels. Figure 2.2 shows an example of partitioning a

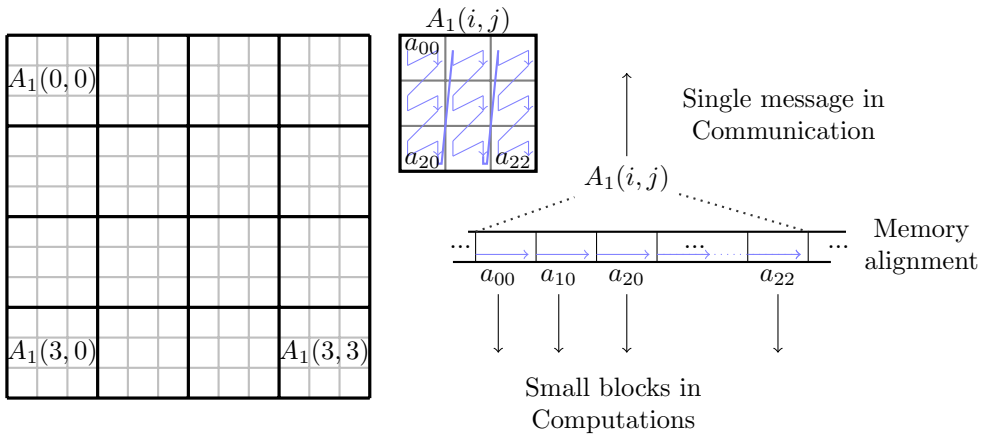


Fig. 2.2: An example of hierarchical data decomposition. A matrix is partitioned into 4×4 large blocks at level 1, each of which contains 3×3 small blocks at level 2. The elements of each block at each level are linearly aligned in contiguous memory. The notation a_{mn} is used for the blocks $A_2(m,n)$ generated from the level 1 block $A_1(i,j)$.

matrix into two hierarchical levels. The matrix elements are ordered such that blocks (tiles) at each level are stored in contiguous memory blocks.

2.2. Hierarchical task generation and task execution. At level 0, the code is traversed by all participating computational nodes. Each task submission statement is inspected with respect to task ownership. If the task is locally owned, it is generated and submitted to the level 0 run-time system. The ownership of a task is decided based on the location of its output data. In the case of several outputs located in different computational nodes, a rule (e.g., first output) is implemented. Due to the hierarchical structure, the number of tasks submitted at the highest level is smaller than in a corresponding one-level approach, thereby reducing the overhead from the global inspection of the tasks in the initial task generation phase.

A task at level ℓ (working on level $\ell+1$ data) contains task submission statements for tasks at level $\ell+1$. When the ℓ level task is ready to run, according to the versioning system described in more detail below, the corresponding $\ell+1$ tasks are generated and submitted to the level $\ell+1$ run-time system. The level ℓ and level $\ell+1$ tasks have a parent-child relationship. The task completion of the parent task depends on the completion of all of its children tasks. However, the run-time system handling the parent tasks can move on to starting the next ready parent task in line without waiting for completion of the previous one. The tasks at level $L-1$, working on the last level data, contain the actual computational kernels. A benefit that follows from the hierarchical task submission is that the submission of lower level tasks is paced in relation to the speed of execution due to the synchronization with the parent tasks. This prevents the number of tasks in the work queues from growing in an uncontrolled way. Figure 2.3 shows an example of a level 0 task graph for a Cholesky factorization and some of the level 1 subgraphs.

In our case with three data levels, the mapping between hardware and software levels in a cluster of multicore computers takes the following form. At level 0, the MPI-based DuctTeip run-time handles task submission at the distributed memory

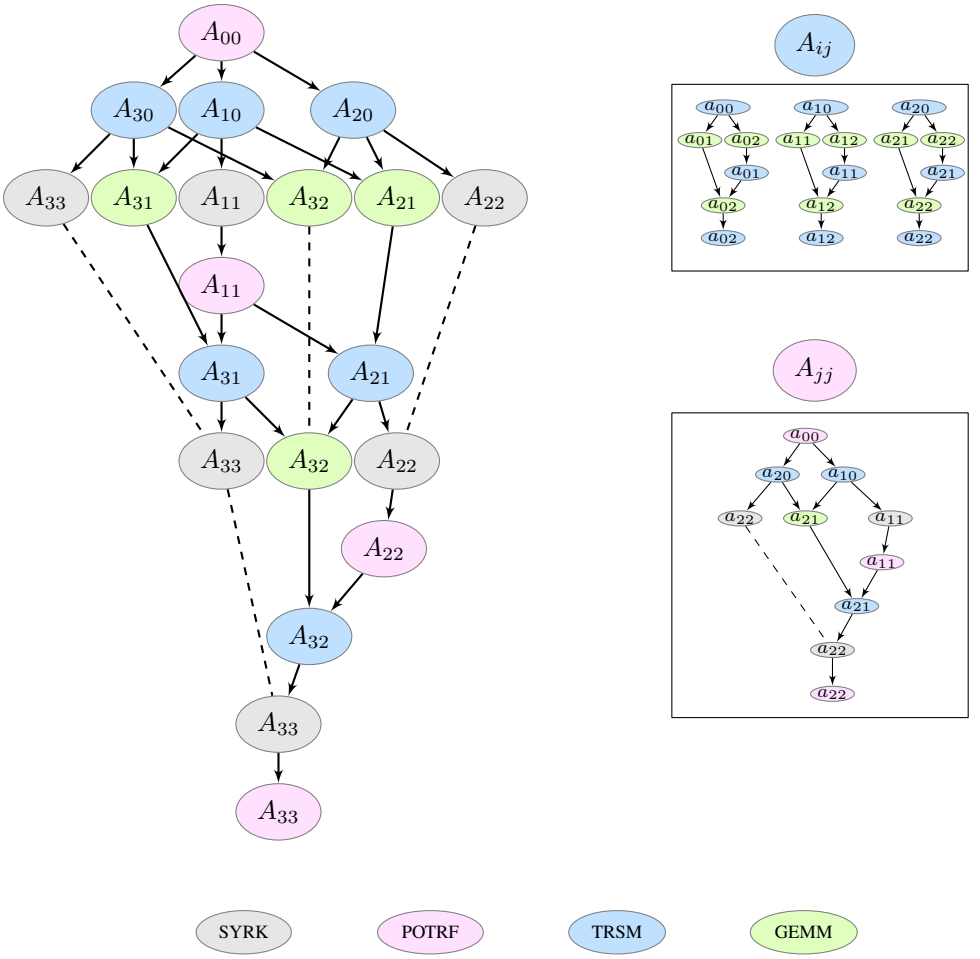


Fig. 2.3: To the left, a level 0 task graph of a 4×4 block Cholesky factorization of a matrix A is shown. The labels of the nodes, A_{ij}, are the output data of the tasks. An arrow between tasks indicates a must execute before dependency, while the dashed lines indicate that the tasks can be reordered, but cannot execute concurrently. To the right, two examples of level 1 task graphs (a block TRSM and a block POTRF) for a 3×3 splitting of the level 1 data are shown.

level. Each computational node runs one MPI process responsible for (among other administrative jobs) running the level 0 tasks. Within each node the level 1 tasks are submitted to the thread-based SuperGlue shared memory run-time system, which executes these tasks in parallel using the local cores.

In the current implementation, the APIs for task submission and task definition with respect to the different run-time systems have some small differences, and code needs to be provided for each participating run-time. Ideally, they should be the same in a general L-level framework. To realize this ideal case is part of an ongoing effort.

2.3. Hierarchical data versioning and dependency tracking. The tracking of data dependencies in the hierarchical task parallel DuctTeip framework is an

extension of the data versioning approach introduced in the SuperGlue shared memory framework. A detailed description of the SuperGlue programming model including data versioning can be found in [23].

Each data handle at each level is equipped with a version counter, which allows the run-time system to order data accesses (at execution time) such that data dependencies are respected. The versioning system is initialized at task submission time based on the assumption that tasks are submitted in a consistent sequential order (at each level). Note that even though tasks at levels $\ell > 0$ are submitted by different nodes, the order is globally preserved because these are submitted at the execution time of the $\ell - 1$ level tasks, which ensures that the dependencies at the parent level are respected and tasks are submitted in a consistent sequential order at each level.

At task submission time, the respective run-time systems count the (future) accesses to each data handle, and for each task, record the *required version number* of each of the accessed data handles. Tasks that require the same version number can be reordered.

During execution, the *run-time version number* of each handle is incremented after each access. Technically, this operation is performed at the completion of the execution of a task. In this way, the run-time system can compare the required version numbers for a certain task with the current run-time version numbers of the involved handles in order to determine if the task is ready to run.

When the arguments of a task are examined to determine if the task is ready for execution and an argument that is not ready is encountered, the task is placed in a queue for that particular data version. This means that as soon as the data is ready, the run-time system knows which tasks to wake up either for execution or for examining of the next data in the argument list. In this way, the need for traversing long lists of waiting tasks is reduced, and tasks can be deployed where the data is locally cached (in particular at the shared memory level). This improves the data locality of the execution, and also reduces the need for a particular scheduling strategy. At the shared memory level, load balancing is achieved dynamically through task stealing. For the distributed memory levels there is so far no dynamical load balancing scheme implemented. The distribution of tasks is, as explained above, determined by the data distribution scheme.

2.4. The distributed communication scheme. In distributed environments, the level 1 data blocks are owned by different computational nodes and must be communicated whenever needed by remote nodes. In our framework, this is solved by the introduction of the *listener* concept.

At task generation time, when data accesses are registered, the location of the data is also examined. In cases where a task requires a specific version of a remote data, a *listener* is sent to (or generated by) the data owner. This means that the remote node knows that the data will be needed later and can send it as soon as it becomes available, see Figure 2.4.

At execution time, when a data handle is upgraded to a new version, tasks that are waiting for that version are examined for execution. Additionally, the listeners waiting for that data version are processed, and the new data is sent to all nodes that have one or more listeners in the queue. Duplicate listeners can occur if several tasks hosted in the same remote node need the data, but then only one message is sent to avoid unnecessary communication traffic. Note that sending the data corresponds to a remote data access, and after completion of the send operation, the version of the data is updated according to the number of listeners that were involved.

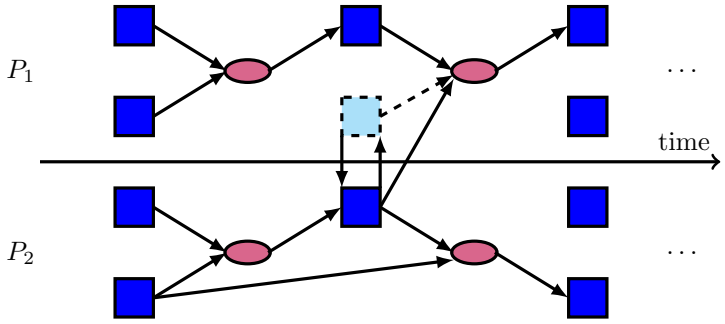


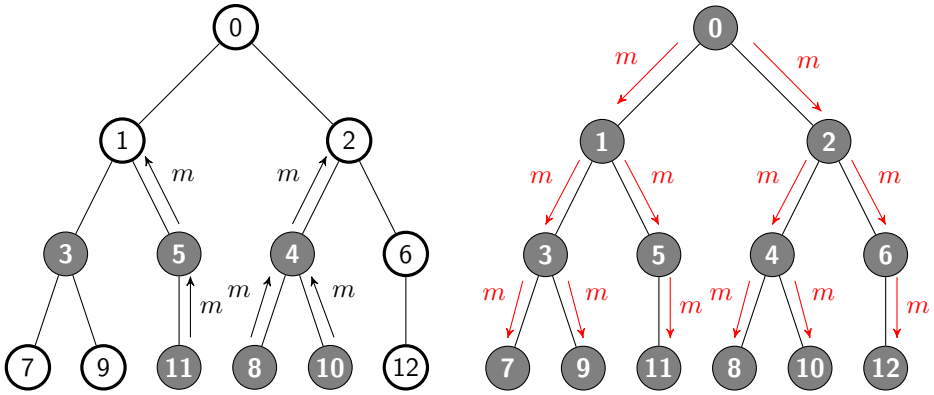
Fig. 2.4: Graphical illustration of the listener concept, where data is represented by boxes, tasks by ellipses, and a local copy of a remote data with a dashed box. In this scenario, process P_1 needs a remote data from process P_2 to run its second task. A listener is generated at task submission time. At execution time, when the required data version is ready, a copy of the data is sent from P_2 to P_1.

2.5. Program Termination. In the DuctTeip framework, no computational node possesses global information about the state of the program. All nodes are working independently based only on their local information about the tasks and data. The only place where global synchronization is needed is at the end of the program. To implement the required global synchronization for termination, we avoid the quadratic $\mathcal{O}(n^2)$ cost of an all-to-all communication among n nodes, by using a binary tree of nodes and sending the termination messages between consecutive levels.

A node is ready to terminate if task submission has ended, there are no tasks in the work queues, and no unprocessed messages or listeners. The first ready-to-terminate messages are collected from the children (starting from the leaf nodes) to their parents, and then termination-OK messages are propagated from the parents (starting from the root node) to the children. The size complexity of this method is $\mathcal{O}(n)$ since every node in the tree sends at most two messages down and one message up; and the time complexity for completing the communication is $\mathcal{O}(n \log n)$.

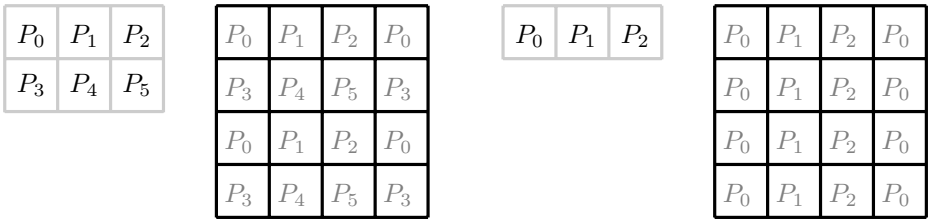
For example, a binary tree of 13 nodes is shown in Figure 2.5a, where gray nodes are ready to terminate. Nodes 8, 10 and 11 are leaf nodes and send the termination message to their parents, nodes 4 and 5 respectively. Nodes 4 and 5 are also ready to terminate and when they have received the message from all their children, they send it to their parents. The non-leaf node 3 is ready to terminate but must wait for its children to become ready to terminate as well. When the root node 0 is ready to terminate and receives termination messages from its children nodes 1 and 2, all the nodes are ready to terminate, Figure 2.5b. Then the root node 0 sends messages to its children and terminates safely. Every node that receives the message from its parent, sends the message to all its children and terminates itself.

3. Programming with the current DuctTeip and SuperGlue implementations. Even if using a programming framework relieves the programmer from many of the more technical aspects of parallel programming, there are still things that the programmer can influence both regarding the algorithm and the program settings that affect the performance of the parallel software.



(a) Upward gathering. (b) Downward scattering.

Fig. 2.5: A hierarchical all-to-all communication of the program termination state.



(a) Block-cyclic ownership. (b) Column-cyclic ownership.

Fig. 3.1: Two different process grids $(p, q) = (2, 3)$ and $(p, q) = (1, 3)$ for the processes P_i, $i = 0, \ldots, 5$, and the resulting block cyclic distributions for a 4×4 block matrix.

3.1. Program configuration. In the current implementation, we assume that shared data are matrices and that we normally have two block levels (there may also be additional types of shared variables such as vectors and even scalar values). We also assume that data at level 1 is distributed and data at level 2 resides within the parent shared memory node. The user needs to specify the block sizes as well as the layout of the process grid, which will be used in the data distribution scheme. These parameters are passed to the DuctTeip framework through the command line as for example

```
mprun -np 32 ductteip -t 16 -p 4 -q 8 -M 48800 4 3,
```

where 32 processes organized in a $p \times q = 4 \times 8$ process grid are created, and a matrix of size (48800×48800) is split into 4×4 level 1 blocks, which in turn are split into 3×3 level 2 blocks. The ownership of the level 1 blocks is block cyclic, with the process grid determining the distribution along rows and columns. An example is shown in Figure 3.1.

3.2. Simulation mode. Clearly, the process grid and block size parameters affect performance, and making the best choice a priori is a non-trivial task. As a support for the user, DuctTeip provides a simulation mode, where the full program is executed in parallel, but with all computational kernels replaced by an empty kernel, and all messages replaced by a one byte message. During the execution, the number of tasks and messages are recorded and relevant measures are provided as output for the user.

The measures that we have found to be most informative are (i) the total communication size, (ii) the work size in terms of computational tasks in one node, and (iii) the total number of messages. Having a large total communication size is likely to be detrimental to performance as the risk of tasks left waiting for remote data increases as well as the risk of congestion of messages.

The work size is measured as the number of level 1 tasks (working on the last level data) multiplied with the typical work complexity. In the Cholesky case, tasks operating on data of size $(n \times n)$ have an $\mathcal{O}(n^3)$ complexity. The work size is a good estimate of the total makespan of performing the actual computations. Note that load imbalance due to the distribution of the data will show up in this measure.

The total number of messages is of less importance than the other two measures. However, large numbers of messages lead to increased communication startup time and can also lead to message congestion and larger overhead.

Figure 3.2 and 3.3 show simulation experiments for Cholesky factorization of a matrix of size $N = 21\,600$ on four computational nodes and of a matrix of size $N = 32\,400$ on nine computational nodes. The size of the level 2 matrix blocks is fixed at a multiple of 180 elements in double precision. This choice is based on experiments performed on one node in order to determine the optimal block size for the particular hardware being used. Each row of subfigures has a constant level 2 block size. The columns show results for different process grid configurations. The horizontal axis shows different splittings $B \times b$, where B is the number of level 1 blocks in each dimension and b is the number of level 2 blocks in each dimension within a level 1 block.

Some conclusions that can be drawn from examining the displayed indicators are that the overall communication size is minimized when using a square process grid, that using too few level 0 tasks (DuctTeip level) can lead to load imbalance (increased work size), and conversely, that using too many level 0 tasks (a consequence of too small level 1 tasks) leads to a large number of messages. Finally, considering the scattered actual execution times, they do not clearly adhere to any of the three measures, but total communication time seems to take precedence over work size.

3.3. Taskifying algorithms. As explained in Section 2.1, an algorithm in the DuctTeip or SuperGlue frameworks is working on (hierarchically) partitioned data. The first step in preparing an algorithm for task-based parallelization is to decide how to partition the shared data. For matrices, blocks or slices are natural partitions. The algorithm then needs to be expressed as working on these data partitions. This could for example correspond to a transition from an element wise Cholesky decomposition algorithm to the block Cholesky factorization. A sequential block Cholesky code is shown as Program 1, and a taskified version is shown as Program 4, in Appendix A.

Shared data can also take other forms. A scalar control variable or a global vector that local results are gathered into can also be shared. However, common for all shared data is that it is protected by a handle, which is a construct provided by the framework. The handle keeps track of the data versions and manages accesses to

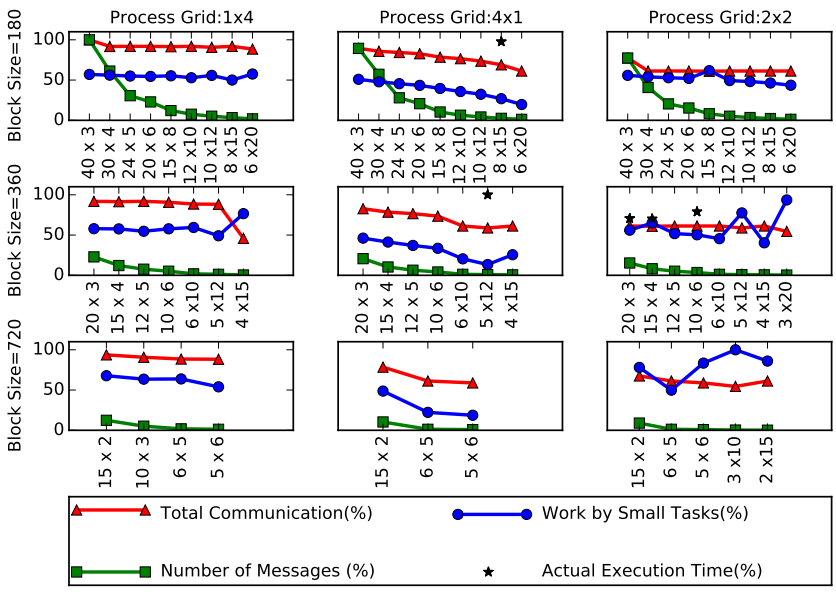


Fig. 3.2: Simulation results for Cholesky factorization of a matrix of size $N = 21\,600$ executed on four computational nodes for different process grids and block sizes. To facilitate making compact graphs, all results are given as a percentage of the maximum value encountered.

the data.

Operations that can become tasks have a clearly defined interface to the shared data in terms of inputs and outputs, and do not modify any other shared data (no global variables). In the taskified code, such operations are replaced by task submission statements. The operations provide the computational kernel of the tasks. APIs for defining and submitting tasks are provided by the framework.

A task submission statement at level 0 (the DuctTeip level) has the form

```
addTask(SYRK, A(i,j), A(i,i));
```

where the first argument indicates the type of the task, and the second and third arguments are handles to the level 1 matrix blocks that are involved in the specific instance of the operation.

A task submission statement at level 1 (the SuperGlue level) looks slightly different. First a new task is created, and then it is added as a subtask to the level 0 parent task.

```
SyrkTask *syrk = new SyrkTask(dt_task, M(i,j), M(i,i));
dt_task->subtask(syrk);
```

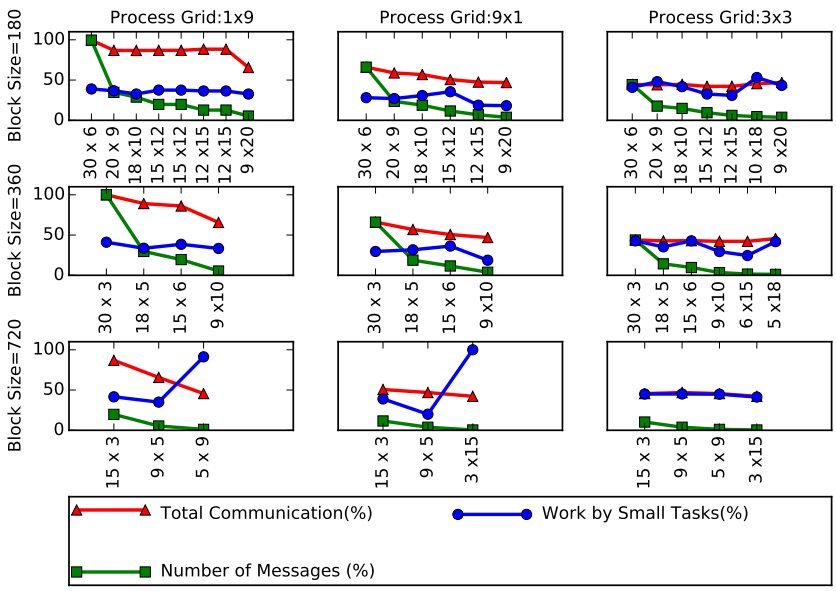


Fig. 3.3: Simulation results for Cholesky factorization of a matrix of size $N = 32\,400$ executed on nine computational nodes for different process grids and block sizes. To facilitate making compact graphs, all results are given as a percentage of the maximum value encountered.

In SuperGlue each type of task has its own subclass, therefore the type is given by the name instead of as an argument. The first argument is here the parent task, and then similarly handles to the level 2 matrix blocks that the task will operate on. Adding the task as a subtask implies that the parent task will wait for it to complete before completing its own execution.

As can be seen here, the syntax differs for task submission at the different levels. This is an effect of combining two different frameworks. This could be done differently in a future version, but here it also gives us the opportunity to explore different ways of implementing tasks. However, this implies that the programmer needs to write two versions of the taskified algorithm, one for each level. Complete examples of taskified algorithms at both levels can be viewed in Appendix A as Programs 4 and 5.

3.4. Implementing tasks and kernels. As briefly touched upon in the previous subsection, in addition to implementing the taskified algorithms, we also need to implement the task objects using the appropriate APIs.

In DuctTeip, an algorithm such as the Cholesky factorization is a class. It contains the taskified algorithm, a method called `runKernels` that connects the name given as an argument to the `addTask` method of the framework with the actual kernel functions, and finally each of the kernel functions used by the algorithm. In the Cholesky

case, these are `POTRF_kernel`, `TRSM_kernel`, `GEMM_kernel`, and `SYRK_kernel`, see Programs 3–5. A kernel function takes a DuctTeip task as its argument, extracts the data from the task, extracts the level 2 data from the level 1 blocks, and submits the SuperGlue subtasks. That is, the DuctTeip kernel function contains the SuperGlue taskified algorithm.

The framework needs to know what type of accesses to the data to expect in order to track the data versions correctly. However, for the linear algebra algorithms that have been considered here, DuctTeip uses an implicit rule that defines all arguments except the last as input, and the last argument as output. Hence, no explicit annotation is provided by the programmer.

At the SuperGlue level, each type of task is a class. The DuctTeip framework provides a slightly simplified layer on top of SuperGlue such that each task class is a specialization of the base class `SuperGlueTaskBase`. The constructor of a specialized task class takes the parent task and handles to the task arguments. The accesses to the data *read*, *write*, or *add* are registered in the constructor. The kernel operation is implemented in the `runKernel` method. In the current implementation, a task at the SuperGlue level works on the last level data. This means that no new tasks are submitted in the kernel. Instead, the kernel operation is the actual computation on the data. In the Cholesky case, this is a call to a BLAS (or LAPACK) routine. A SuperGlue task class is shown as Program 6 in Appendix A.

4. Technical aspects of DuctTeip. This section explains the choices made in DuctTeip to implement the logical aspects described in the previous sections. In particular, the communication and memory usage in the framework are discussed as they have an impact on performance.

4.1. Asynchronous communication and threading models. In order to overlap communication with computations, all send and receive MPI calls should be in asynchronous non-blocking mode using MPI routines that return immediately while the requested operation is carried out internally by MPI. To check completion of the requested operation, the caller program asks MPI about the state of the request object returned by MPI at the calling time.

Checking the state of requests can be performed in blocking or non-blocking mode. Currently, only the non-blocking mode is practical to use, and this is the chosen method in DuctTeip. However, in the non-blocking mode, calls return immediately even if there are no updates in the state. This means that a periodic polling of the state is needed and this results in an overhead.

If we instead use blocking calls to check the state, the main program requires one or more threads that are dedicated to MPI calls, such that computations can continue in parallel also while the communication threads are blocked. Using one or more threads for computations and only one thread for MPI calls, may result in a performance bottleneck, since the computational threads cannot access (unlock) the MPI thread while it is waiting (locked) for completion of previous requests. As a general solution, a buffer for holding the pending MPI requests from the computational threads can be used. The main disadvantage of this solution is a delay between the time of actually sending a message and the time it was ready to send due to buffering.

To avoid the performance issues mentioned above, it is preferable to have more than one thread for MPI calls. Then the threads can be specialized such that one thread manages send requests and one thread manages the receive state. Using more than one thread for MPI calls requires the MPI library be fully compliant with the

MPI.2 specification of threading models, particularly `MPI_THREAD_MULTIPLE` in which many threads can call MPI routines independently.

In MPI implementations that are not fully compliant, the request for using `MPI_THREAD_MULTIPLE` may instead result in one of two other threading models, which again lead to performance issues. Specifically, in the serialized threading model (`MPI_THREAD_SERIALIZED`), multiple threads can call MPI routines, but only one thread at a time is allowed to make progress. The others are blocked until the working thread completes. Similarly, the funnel threading model (`MPI_THREAD_FUNNEL`) allows only one thread to call MPI routines.

At the time DuctTeip was developed, the latest versions of the most common MPI implementations were not fully compliant with respect to multi-threading. Hence, the choice of using the non-blocking polling approach.

4.2. Memory considerations. The memory used for data in DuctTeip is not naively allocated/deallocated, rather it is managed carefully regarding the various uses and states a data may have during run-time. The memory manager in DuctTeip creates a memory pool at start-up time before the task submission begins. Then at the task execution time memory locations are assigned upon request. At run-time, a memory request may occur when a new version of a data is received while an older version is still in use. Memory can also be released at run-time when there are no tasks waiting to use the particular data. By using a memory pool, the time spent for memory allocation and deallocation is reduced as well as shifted out of the task execution phase.

Every item in the memory pool points to contiguous locations of physical memory. This is beneficial both for communication, as data can be sent or received in a single message, and for computations due to improved data locality. The same memory area is used for further partitioning the data into smaller blocks, hence makes it possible for received data to immediately be used by tasks at deeper levels (see Figure 2.2).

This centralized way of managing the memory requests can also allow the framework to adapt the memory allocation to the underlying physical memory structure, for example in NUMA architectures, to efficiently distribute the allocated memory among the NUMA nodes. In Linux operating systems, the `numactl` utility provides policies for NUMA allocation, and the `hwloc` library [10] allows us to probe the architecture of the system. It is possible to connect allocated memory with a specific NUMA node, socket, or core.

In the current version of the DuctTeip framework, there is no general solution for meeting the memory needs of all applications and memory architectures. However, our experiments show significant improvements in performance when the memory for level 1 blocks is allocated as a unit and the memory addresses for the level 2 blocks are derived as an offset to the base address of the level 1 block, compared with allocating memory individually for each level 2 block. In the experiments, allocations are distributed evenly (interleaved in a round-robin mode by the NUMA controller) among the cores of the system.

4.3. Hybrid approaches versus pure MPI approaches. We have chosen to use a hybrid MPI–Pthreads approach in DuctTeip. In this section, we describe the main reasons for this choice from a performance perspective. Whether using a hybrid or a pure MPI approach, the parallel program runs on p distributed computational nodes, each with n cores. In the pure MPI case, pn instances of the program run concurrently, with n instances within each node. In the hybrid MPI–Pthreads case

instead p instances of the program run on the computational nodes, and within every node the work is distributed over n threads.

Using only MPI instances of a program to exploit the parallelism provided by the underlying hardware degrades the performance in comparison with a hybrid approach for the following reasons: First, the memory allocated for each instance of an MPI process within a node is private to that process and can only be accessed by other processes via the main memory of the node, using techniques like inter process communication (IPC). Therefore, communicating even cached data between two processes running on the same CPU, and hence potentially sharing the same cache, requires a round-trip for the data to the main memory. Second, if the cost of communication between q processes of a program is $\mathcal{O}(f(q))$ then the cost of running the program in a pure MPI mode is $\mathcal{O}(f(pn))$. For example, in the worst case of all-to-all communication, with cost $\mathcal{O}(q^2)$, the cost for a pure MPI model is $\mathcal{O}(p^2n^2)$ while for a hybrid model it becomes $\mathcal{O}(p^2)$.

Additionally, in the pure MPI case, the amount of memory that needs to be allocated by one node to receive messages becomes unnecessarily large. For example, if processes P_1 and P_2 running on node i own data A_1 and A_2 respectively, and communicate these data between each other, then we need a receive buffer for each of the two items at node i. If instead using a hybrid approach, no extra buffers are needed for communication within the node. This means that in a pure MPI mode, we run out of memory faster when increasing the node local work size.

5. Experimental results. The Cholesky factorization is used as a benchmark for the performance experiments. The algorithm is well suited for task based parallelization, has non-trivial dependencies, and has been implemented also by other framework developers. We compare the performance of DuctTeip with that of PaRSEC (DPLASMA 1.2.1), ScaLAPACK 2.0.2, Cluster OmpSs and StarPU (1.1.5). All experiments have been performed on the Tintin cluster of the Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). The cluster has 180 dual socket computational nodes with 64 GB/128 GB memory each. Each socket is equipped with an eight core AMD Opteron 6220 (Bulldozer) processor running at 3.0 GHz.

Each framework is compiled and built with the available components that provide the best possible performance for the target hardware. DuctTeip and PaRSEC are compiled and built by the Intel C/C++ 13.1 compiler. The ScaLAPACK Cholesky factorization implementation is written in Fortran calling the DPOTRF subroutine and compiled with the PGI 13.8 compiler. ScaLAPACK itself, is also built by PGI 13.8 and optimized for the target system. StarPU is built using gcc 4.8 and calibrated for the Cholesky factorization program using the dmdar (deque model data aware ready) task scheduling policy. For Cluster OmpSs, the customized compilers Mercury 1.99.7 and Nanos+ 0.9a from the Barcelona Supercomputing Center are built on top of gcc 4.4.7. The Cholesky application is then compiled and built using these.

All the frameworks except StarPU use the ACML 5.3.1 implementation of the BLAS library optimized for the target system. StarPU uses the ATLAS library 3.10.2, also optimized for the target system. The hwloc library 1.9.1 is used by PaRSEC and StarPU and built by gcc 4.4.7. Finally, all applications are linked with OpenMPI 1.6.5.

The most frequent operation in the block Cholesky factorization is a GEMM operation (matrix–matrix multiplication). To have a reference to what could be considered the maximal achievable performance we include what we call the GEMM peak in our

comparisons. The maximum throughput of the `dgemm` BLAS subroutine on one node (16 cores) that is achieved on the Tintin cluster is 162 GFLOPS for square matrices of size $N = 24\,000$. The GEMM peak is then computed as the one-node-result scaled by the number of nodes.

5.1. Strong scaling experiments. For the strong scaling experiments, we first identified the largest problem that can be solved by a single node without performance (throughput) degradation. Then, fixing the problem size, we increase the number of nodes. Theoretically, linear scalability with the number of nodes is expected. The experimental results are shown in the left part of Figure 5.1. The best scalability is shown by PaRSEC, which has a very low run-time overhead due to the optimal schedule that is embedded in the code. The StarPU Cholesky factorization used in the experiment uses single level data and tasks, and the scalability is limited. Cluster OmpSs exhibits the worst performance for this problem. The centralized task submission could be an explanation for this lack of scalability. ScaLAPACK performs better than both StarPU and Cluster OmpSs, but is less scalable than DuctTeip. However, ScaLAPACK is a specialized library, and not a general purpose framework. DuctTeip scales better than the other dynamically scheduled frameworks, which indicates that the programming model with hierarchical partitioning of data and tasks coupled with decentralized task submission and scheduling is appropriate for the architecture we are using. It should be noted though that the frameworks are frequently upgraded to new versions, and later versions are likely to perform better than these results.

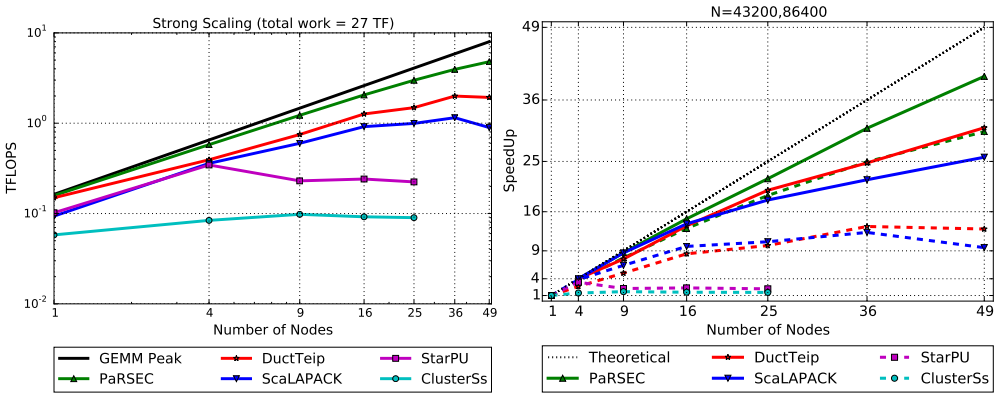


Fig. 5.1: The left subfigure shows the scaling of the different frameworks for Cholesky factorization of a matrix of size $N = 43\,200$. In the right subfigure, the fixed size speedup for problem sizes $N = 43\,200$ (dashed lines) and $N = 86\,400$ (solid lines) is shown.

The right part of Figure 5.1 shows the speedup for the same problem (dashed lines) for up to 49 nodes (784 cores) as well as for a problem with a larger matrix size, optimized to give maximum performance on four nodes (64 cores) (solid lines). The speedup numbers are computed as $S_p = T_1/T_p$ for the first experiment, and $S_p = 4T_4/T_p$ for the second experiment. The relative performance of the frameworks is the same in both experiments, but the second experiment also shows that increasing the workload results in a reduction of the relative size of the overhead, and the speedup

and scaling are thereby improved. Both StarPU and Cluster OmpSs had problems with handling large matrices and/or larger number of nodes, and therefore no results are shown for these frameworks in those cases.

5.2. Weak scaling experiments. For the weak scaling experiments, we have chosen to increase the problem size in such a way that the communication size is constant. As discussed in Section 2.4, communication has a larger impact on the actual performance than the work size itself. The communication size is approximately proportional to N^2/p, where p is the number of processes. Hence, we let the problem size grow from $N = 10\,800$ on one node up to $N = 75\,600$ on 49 nodes (784 cores).

Figure 5.2 shows the experimental result. In absolute numbers, PaRSEC shows the best performance, followed by DuctTeip , and then ScaLAPACK. However, looking at the slope of the scaling results, PaRSEC scales almost perfectly, while DuctTeip scales a little bit worse than ScaLAPACK. This may be expected since DuctTeip is the only framework of the three that employs dynamic scheduling, which would incur a larger overhead.

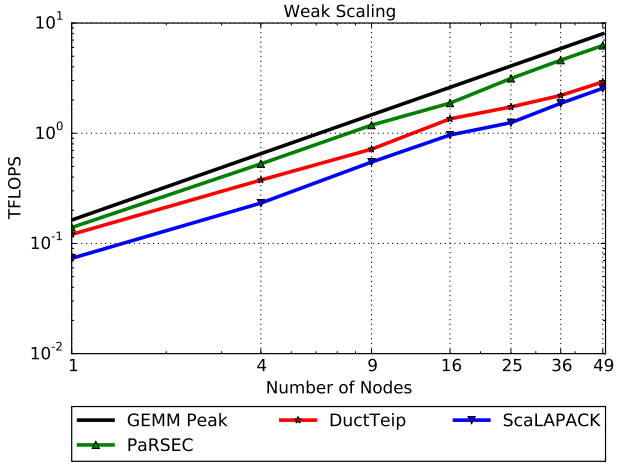


Fig. 5.2: Weak scaling scaling results for the Cholesky factorization when the problem size is scaled up from $N = 10\,800$ one node such that N^2/p, where p is the number of processes, is held constant.

6. Conclusions. We have introduced a dependency-aware hierarchical task-based programming model, where the execution order of the tasks is controlled by a versioning system. The data versioning approach has previously been shown to be very effective in a shared memory setting [23], and here we have shown that it performs equally well for distributed memory systems. In particular, the consistency of the hierarchical task submission is automatically handled by the framework, and tasks at all levels can be executed in parallel. Furthermore, we have proposed an abstract software construct, the listener, which enables the run-time system to control the communication of data. Without this abstraction level, the run-time system would not be able to detect multiple requests of the same data from a single requester or resolve the arrival of multiple versions of the same data by a single receiver.

Using a hierarchical decomposition of data and consequently of tasks, allows us to break down the cost of managing the tasks and data into two (or more) independent levels, and it also allows us to choose suitable block sizes for communication and computations separately. Additionally, the hierarchical break down of tasks reduces the overhead, and naturally introduces an interleaving of the submission and execution of the lowest level computational tasks that helps to prevent memory issues resulting from excessive numbers of tasks being generated.

In the current DuctTeip implementation, the programmer needs to supply the task definitions and algorithms at each level. This is an unnecessary obstacle to the ease of programming in the framework. Therefore, in an ongoing project, we are developing a unified interface, such that the algorithm and the task kernels only need to be defined once and then can be reused for all levels.

A potential obstacle to performance is load imbalance between the computational nodes. The task ownership is an implicit result of the data distribution scheme, the block sizes, the process grid, and the algorithm. Clearly, these factors together may result in load imbalance. In the DuctTeip framework both tasks and data are designed to be movable objects. Hence, the framework is prepared for distributed dynamical load balancing. In another ongoing project, we are investigating whether distributed load balancing can be implemented within the framework in such a way that overall performance is improved.

We have to our knowledge performed the first comparison involving several of the established distributed task parallel frameworks. The experiments show that DuctTeip compares very well with similar frameworks and libraries regarding both absolute performance and relative speedup. Only PaRSEC, which is extremely scalable, but also encodes more information about the task graph in the compiled program, performs better.

Acknowledgments. The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under Project p2009014. Lennart Karlsson at UPPMAX is acknowledged for assistance concerning technical and implementational aspects in making the code run on the UPPMAX resources.

Javier Bueno Hedo, Barcelona Supercomputing Centre, Barcelona, Spain, is acknowledged for assistance with the installation of the Cluster OmpSs compilers and for providing the Cholesky factorization code for Cluster OmpSs.

Appendix A. A sample user code. In this appendix, we show how the Cholesky factorization algorithm can be implemented as a C++ program using the DuctTeip framework. The complexity of the dependencies between tasks and data in this factorization illuminates the flexibility of the framework in handling algorithms that are not straightforward to parallelize, see Figure 2.3 for a task graph of a 4×4 block Cholesky factorization.

Since the DuctTeip framework uses the same logic, concepts and abstracts as the SuperGlue framework does, any algorithm that can be implemented using SuperGlue can similarly be implemented using DuctTeip. For some examples of algorithms implemented with SuperGlue, see [13, 25, 23, 4].

To learn even more about the details of the framework implementations, visit the SuperGlue framework source repository at https://github.com/tillenius/superglue/ and the DuctTeip framework source repository at https://github.com/afshin-zafari/DuctTeip.

A.1. The sequential block Cholesky factorization. Program 1 shows a sequential program that implements the block Cholesky factorization algorithm applied to a matrix A of size $N \times N$. The matrix is partitioned into $nB \times nB$ blocks, each of which contains $nb \times nb$ elements, where $nb = N/nB$.

The algorithm, in LAPACK terminology called `potrf`, can in turn be described in terms of a block version of itself together with the BLAS level 3 functions `syrk`, `trsm`, and `gemm` applied at the block level. The details of these functions are not shown here.

```
1          // For each dimension:
2  int N;  //   Number of elements
3  int nB; //   Number of blocks
4  int nb; //   Number of elements in each block
5  void syrk(double *X, double *Y){...}
6  void trsm(double *X, double *Y){...}
7  void gemm(double *X, double *Y,double *Z){...}
8
9  /* access to the memory of the block at (i,j) */
10 #define A(i,j) A[j*nB*nb+i*nb]
11
12 void potrf(double *A){
13   for(int i = 0; i< nB ; i++){
14     for(int j = 0;j<i;j++){
15       syrk(A(i,j),A(i,i));                    // A_ii = A_ij A_ij^T
16       for(int k = i+1; k<nB ; k++){
17         gemm(A(k,j),A(i,j),A(k,i));           // A_ki = A_kj A_ij
18       }
19     }
20     potrf(A(i,i));                            // A_ii = LL^T
21     for(int j= i+1; j<nB; j++){
22       trsm(A(i,i),A(j,i));                    // A_ji = A_ii^-1 A_ji
23     }
24   }
25 }
26 int main (){
27   N=1000;nB=10;nb=N/nB;
28   double *A=new double[N*N];                  // A := N x N
29   potrf(A);                                   // A -> LL^T
30 }
```

Program 1: A sequential implementation of a block Cholesky factorization.

A.2. The DuctTeip main program. Program 2 shows the `main` function of the block Cholesky factorization using the DuctTeip and SuperGlue frameworks. First the program is configured by passing the command line arguments to the function `DuctTeip_Start()`. Then a matrix A at the DuctTeip level is defined taking its size and blocking information from the configuration parameters. An instance of the `Cholesky` class is created with the matrix as its input. The call to its `taskified` method starts the task submission and the framework executes the tasks in parallel. The call to the function `DuctTeip_Finish()` at the end of the program waits for

completion of all the tasks and finalizes the DuctTeip framework.

```
1 int main(int argc, char * argv[]){
2   DuctTeip_Start(argc, argv);
3   /*Defining data at DuctTeip level
4     with dimensions read from the configuration*/
5   DuctTeip_Data A(config.N,config.N);
6
7   // Create a new instance of Cholesky class
8   //   and call its taskified method
9   Cholesky *C=new Cholesky(&A);
10  C->taskified();
11
12  DuctTeip_Finish();
13 }
```

Program 2: The `main` function of a program using the DuctTeip framework.

A.3. The DuctTeip Cholesky class. The header file of the `Cholesky` class is shown in Program 3. This class is inherited from the `Algorithm` class provided by the DuctTeip framework. In this class, internal data members and the keys for the tasks are defined (lines 15–16). The matrix that the algorithm will be applied to is provided in the constructor (line 19). The `taskified()` method is called for submitting tasks to factorize the input matrix (details are shown in Program 4, lines 2–20).

Whenever any of the tasks get ready to run, the DuctTeip framework calls the `runKernels()` method of the class while passing as argument a pointer to the task, whose key can be retrieved and used for mapping to the actual kernels (see, for example, how the POTRF key is mapped to the kernel in Program 4, line 25).

```
1 #ifndef __CHOLESKY_HPP__
2 #define __CHOLESKY_HPP__
3
4 #include "ductteip.hpp"
5 #include "math.h"
6 #include "potrf.hpp"
7 #include "syrk.hpp"
8 #include "trsm.hpp"
9 #include "gemm.hpp"
10 #include <acml.h>
11
12 class Cholesky: public Algorithm{
13 private:
14   // The input/output data of Cholesky factorization
15   DuctTeip_Data  *M,A;
16   enum KernelKeys {POTRF,TRSM,GEMM,SYRK};
17 public:
18
19   Cholesky(DuctTeip_Data *inOutData = NULL );
20
21   // The taskified version of Cholesky factorization
```

```
22   void taskified();
23
24   // Called by the framework, when a task is ready to run
25   void runKernels(DuctTeip_Task *task );
26
27   // Kernels of the tasks
28   void POTRF_kernel(DuctTeip_Task *dt_task);
29   void  TRSM_kernel(DuctTeip_Task *dt_task);
30   void  SYRK_kernel(DuctTeip_Task *dt_task);
31   void  GEMM_kernel(DuctTeip_Task *dt_task);
32
33   string getTaskName(unsigned long key);
34   void  taskFinished(DuctTeip_Task *task, TimeUnit dur);
35   void  checkCorrectness();
36   void  populateMatrice();
37   void  dumpAllData();
38   ~Cholesky();
39 };
40 #endif //__CHOLESKY_HPP__
```

Program 3: The header file of the `Cholesky` class inherited from the `Algorithm` class.

The algorithm for submitting the DuctTeip tasks of the block Cholesky factorization is shown in Program 4 in the `taskified()` method definition. The taskified implementation of the Cholesky algorithm is the same as the sequential version (Program 1) except for the calls to the functions which are here replaced with their corresponding task submissions. The labels `SYRK`, `GEMM`, `POTRF` and `TRSM` are the user defined keys (Program 3, line 16) linking them to the corresponding kernels. The `addTask` method that submits a task to the framework is inherited from the `Algorithm` class. The first argument is the task key and the rest are the data used by the task. In this simplified interface, all the data arguments are being read by the task except the last one which is being written by the task.

```
1 #include "cholesky.hpp"
2 void Cholesky::taskified(){
3   int Nb = A.getXNumBlocks();
4   for(int i = 0; i<Nb; i++){
5     for(int j = 0; j<i; j++){
6       // submit task for A_ii = A_ij A_ij^T
7       addTask(SYRK, A(i,j), A(i,i));
8       for(int k = i+1; k<Nb; k++){
9         // submit task for A_ki = A_kj A_ij
10        addTask(GEMM, A(k,j), A(i,j), A(k,i));
11      }
12    }
13    // submit task for A_ii -> LL^T
14    addTask(POTRF, A(i,i));
15    for(int j = i+1; j<Nb; j++){
16      // submit task for A_ji = A_ii^-1 A_ji
17      addTask(TRSM, A(i,i), A(j,i));
18    }
```

```
19   }
20 }
21
22 void Cholesky::runKernels(DuctTeip_Task *task )
23 {
24   switch (task->getKey()){
25     case POTRF:   POTRF_kernel(task);      break;
26     case  TRSM:    TRSM_kernel(task);      break;
27     case  GEMM:    GEMM_kernel(task);      break;
28     case  SYRK:    SYRK_kernel(task);      break;
29     default:
30       fprintf(stderr,"invalid task key:%ld.\n",
31               task->getKey());
32       exit(1);
33       break;
34     }
35 }
```

Program 4: Excerpt from `cholesky.cpp` that implements the `Cholesky` class, showing the `taskified` and `runKernels` methods.

In Program 4, lines 22–35 show implementation of the `runKernels()` method in the `Cholesky` class. Since this is the routine that is called by the DuctTeip framework when a DuctTeip taskgets ready to run, the control logic implemented in the `runKernels()` method determines the overall behavior of the program. In the code shown here, the method simply dispatches the control flow to the corresponding task kernels using a mapping scheme. These kernels in turn submit tasks to the SuperGlue run-time system.

A.4. The DuctTeip POTRF kernel. Program 5 shows the `POTRF_kernel()` member method of the `Cholesky` class, which is called by the `runKernels()` method. Inside this method, the argument of the task is retrieved (line 3) and then the corresponding SuperGlue data partitioning within that data is extracted by calling the function `getSuperGlueData()` at line 7. Once again, the POTRF kernel also implements the block Cholesky factorization, but here for the blocks at the SuperGlue level. The implementation is similar to the sequential version of the algorithm except that the calls to the BLAS level 3 routines are replaced by task submissions to the SuperGlue framework. In order to submit tasks corresponding to these BLAS routines, first an instance of a specific SuperGlue task type (`PotrfTask`, `SyrkTask`, `GemmTask`, or `TrsmTask`) is created with their input/output arguments, and then the new task is added as a subtask to the parent DuctTeip task using its `subtask()` method. The `subtask()` method internally submits the subtask to the SuperGlue framework.

```
1 void Cholesky::POTRF_kernel(DuctTeip_Task *dt_task){
2   // Get the argument of the POTRF task
3   DuctTeip_Data  *A = dt_task->getArgument(0);
4
5   // Retrieve the corresponding SuperGlue blocks
6   //   from the DuctTeip data
7   SuperGlue_Data M = A->getSuperGlueData();
8   int n = M.get_block_count();
```

```
9
10   // Blocks of A can be accessed using '(i,j)' indexing
11   // and passed as the SuperGlue tasks' arguments
12   for(int i = 0; i<n ; i++){
13    for(int j = 0; j<i ; j++){
14     // create and submit task for M_ii = M_ij M_ij^T
15     SyrkTask *syrk = new SyrkTask(dt_task,M(i,j),M(i,i));
16     dt_task->subtask(syrk);
17     for (int k = i+1; k<n ; k++){
18      // create and submit task for M_ki = M_kj M_ij
19      GemmTask *gemm=
20        new GemmTask(dt_task,M(k,j),M(i,j),M(k,i));
21      dt_task->subtask(gemm);
22     }
23    }
24    // submit task for M_ii -> L L^T
25    PotrfTask *potrf = new PotrfTask(dt_task, M(i,i));
26    dt_task->subtask(potrf);
27    for( int j = i+1; j<n ; j++){
28      // submit task for M_ji = M_ii^-1 M_ji
29      TrsmTask *trsm = new TrsmTask(dt_task,M(i,i),M(j,i));
30      dt_task->subtask(trsm);
31    }
32   }
33 }
```

Program 5: Excerpt from `cholesky.cpp` that implements the `Cholesky` class, showing the `POTRF_kernel` method.

A.5. The SuperGlue Potrf task class. Program 6 shows the class `PotrfTask` for a task at the SuperGlue level which is supposed to be created and submitted within a kernel of a DuctTeip task. Hence it is inherited from the `SuperGlueTaskBase` class provided by the framework for this purpose. An instance of the `PotrfTask` class is created at line 25 of Program 5 with its actual input data. The input data is passed to the constructor of the class and the input/output direction of the data is registered in the SuperGlue framework (Program 6, line 11). The `PotrfTask` performs the Cholesky factorization in place, i.e., it saves the result of the factorizations in the original memory location. Thus its access to the argument is registered in SuperGlue framework as `write`. The `runKernel()` method of this class is called by the SuperGlue framework when the `PotrfTask` is ready to run. Inside the `runKernel()` method, the `get_argument()` method is used to retrieve the information about the arguments of the task, line 17. This method returns the data at the last level of the data hierarchy (`LastLevel_Data`) whose memory address and size can be retrieved by using the `get_memory()` and `get_rows_count()` methods, respectively (lines 20 and 21). The memory reference is then passed directly to the computational kernel function, which here is `dpotrf` (line 23).

```
1 #include <acml.h>
2 // A SuperGlue task that is submitted
3 //   in the kernel of a DuctTeip task
```

```
class PotrfTask : public SuperGlueTaskBase {
public:
  // At construction time,
  //    gets the parent DuctTeip task
  //    and the input/output SuperGlue data
  PotrfTask( DuctTeip_Task *task_,Handle<Options> &A):
    SuperGlueTaskBase(task_) {
    registerAccess(ReadWriteAdd::write, A);
  }
  // This is called by the SuperGlue framework,
  //   when the task is ready to run.
  void runKernel() {
    // Get the first argument of the task
    LastLevel_Data a = get_argument(0);

    // Get the memory address and its size,
    double *mem = a.get_memory();
    int info, N = a.get_rows_count();

    dpotrf('L',N,mem,N,&info);
  }
};
```

Program 6: The definition of a SuperGlue task within the DuctTeip framework.

REFERENCES

[1] G. Almasi, *PGAS (Partitioned Global Address Space) languages*, in Encyclopedia of Parallel Computing, Springer US, Boston, MA, 2011, pp. 1539–1545.

[2] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, *StarPU-MPI: Task programming over clusters of machines enhanced with accelerators*, in Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23–26, 2012, pp. 298–299.

[3] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures*, Concurrency Computat.: Pract. Exper., 23 (2011), pp. 187–198.

[4] P. Bauer, S. Engblom, and S. Widgren, *Fast event-based epidemiological simulations on national scales*, Int. J. High Perform. Comput. Appl., (2016). Electronic publication ahead of print.

[5] F. Blagojevic, P. Hargrove, C. Iancu, and K. A. Yelick, *Hybrid PGAS runtime support for multicore nodes*, in Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS 2010, New York, NY, USA, October 12-15, 2010. Article 3.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, in Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995, pp. 207–216.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, *PaRSEC: Exploiting heterogeneity to enhance scalability*, Comput. Sci. Eng., 15 (2013), pp. 36–45.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra, *DAGuE: A generic distributed DAG engine for high performance computing*, Parallel Comput., 38 (2012), pp. 37–51.

[9] A. Bouteiller, T. Herault, G. Bosilca, P. Du, and J. Dongarra, *Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy*, ACM Trans.

Parallel Comput., 1 (2015), pp. 10:1–10:28.

[10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, *hwloc: A generic framework for managing hardware affinities in HPC applications*, in Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17–19, 2010, pp. 180–186.

[11] Cilk Plus, *Intel® C++ Compiler 16.0 User and Reference Guide: Intel® Cilk™ Plus*. https://software.intel.com/en-us/intel-cplusplus-compiler-16.0-user-and-reference-guide-cilk-plus, June 2016.

[12] J. Dongarra and P. Luszczek, *PLASMA*, Springer US, Boston, MA, 2011, pp. 1568–1570.

[13] M. Holm, S. Engblom, A. Goude, and S. Holmgren, *Dynamic autotuning of adaptive fast multipole methods on hybrid multicore CPU and GPU systems*, SIAM J. Sci. Comput., 36 (2014), pp. C376–C399.

[14] Intel® TBB, *Intel® threading building blocks documentation*. https://www.threadingbuildingblocks.org/, June 2016.

[15] L. V. Kalé, *Charm++*, in Encyclopedia of Parallel Computing, Springer US, Boston, MA, 2011, pp. 256–264.

[16] L. V. Kalé and S. Krishnan, *CHARM++: A portable concurrent object oriented system based on C++*, in Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Washington, DC, USA, September 26 - October 1, 1993, pp. 91–108.

[17] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras, *Towards more efficient execution: a decoupled access-execute approach*, in Proceedings of the International Conference on Supercomputing, ICS'13, Eugene, OR, USA, June 10–14, 2013, pp. 253–262.

[18] J. Kurzak and J. Dongarra, *Implementing Linear Algebra Routines on Multi-core Processors with Pipelining and a Look Ahead*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 147–156.

[19] OpenMP Architecture Review Board, *OpenMP 4.0 Complete specifications*. http://openmp.org/wp/openmp-specifications/, July 2013.

[20] J. M. Pérez, R. M. Badia, and J. Labarta, *A dependency-aware task-based programming environment for multi-core architectures*, in Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan, 2008, pp. 142–151.

[21] E. H. Rubensson and E. Rudberg, *Chunks and tasks: A programming model for parallelization of dynamic algorithms*, Parallel Comput., 40 (2014), pp. 328–343.

[22] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, *ClusterSs: a task-based programming model for clusters*, in Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8–11, 2011, pp. 267–268.

[23] M. Tillenius, *SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization*, SIAM J. Sci. Comput., 37 (2015), pp. C617–C642.

[24] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell, *Resource-aware task scheduling*, ACM Trans. Embedded Comput. Syst., 14 (2015), pp. 5:1–5:25.

[25] M. Tillenius, E. Larsson, E. Lehto, and N. Flyer, *A scalable RBF-FD method for atmospheric flow*, J. Comput. Phys., 298 (2015), pp. 406–422.

[26] A. Zafari, M. Tillenius, and E. Larsson, *Programming models based on data versioning for dependency-aware task-based parallelisation*, in Proceedings of the 15th IEEE International Conference on Computational Science and Engineering, CSE 2012, Paphos, Cyprus, December 5–7, 2012, pp. 275–280.