DAGuE: A generic distributed DAG engine for high performance computing

George Bosilca*, Aurelien Bouteiller*, Anthony Danalis*†, Thomas Herault*, Pierre Lemarinier*, Jack Dongarra*†
*University of Tennessee Innovative Computing Laboratory
†Oak Ridge National Laboratory
‡University Paris-XI

***Abstract*—**
The frenetic development of the current architectures places a strain on the current state-of-the-art programming environments. Harnessing the full potential of such architectures has been a tremendous task for the whole scientific computing community.

We present DAGuE a generic framework for architecture aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. Applications we consider can be represented as a Direct Acyclic Graph of tasks with labeled edges designating data dependencies. DAGs are represented in a compact, problem-size independent format that can be queried on-demand to discover data dependencies, in a totally distributed fashion. DAGuE assigns computation threads to the cores, overlaps communications and computations and uses a dynamic, fully-distributed scheduler based on cache awareness, data-locality and task priority. We demonstrate the efficiency of our approach, using several micro-benchmarks to analyze the performance of different components of the framework, and a Linear Algebra factorization as a use case.

I. Introduction and Motivation

The past few years have witnessed a persistent increase in the number of cores per CPU and in the use of accelerators. This trend can only be expected to continue, as hardware vendors announce chips with as many as 80 cores, multi-GPU capable compute nodes and potentially a tighter integration between the accelerators and the processors. While, from a pure performance viewpoint, this additional performance is welcome, from a programming perspective it is difficult to extract additional performance from the available hardware.

While alternative programming paradigms have been emerging, explicit message passing, using MPI, is currently the dominant approach to creating parallel applications. However, MPI alone does not provide mechanisms to fully harness the potential performance of multi-core applications. To achieve that, a hybrid programming model is a commonly proposed solution, with MPI processes running across nodes and multiple threads running on each node.

Unfortunately, programming hybrid applications is difficult and error prone. The application programmer is required to address several low level problems, such as explicit communications, mutual exclusion, load balancing, memory distribution, cache reuse and memory locality on non-uniform memory access (NUMA) architectures. These issues are hard to address and yet are orthogonal to the algorithm design and are fundamental issues of the computational research domain scientists and engineers are mostly interested in.

In this paper, we present *DAGuE*, a framework for parallel application developers, that moves the task of addressing the system specific performance issues from the application developer to the DAGuE run-time system developer. DAGuE is a Direct Acyclic Graph (DAG) scheduling engine, where the nodes of a DAG are sequential computation tasks and the edges are data communications. Therefore, designing a parallel application with this framework consists of encapsulating computation tasks into sequential kernels and defining, through a DAGuE specific language, how these kernels interact with each other.

DAGuE schedules tasks in a fully distributed and dynamic fashion. It enables local tasks to make progress waiting only on data dependencies to other tasks, and no process has a global knowledge of the execution progress of remote processes. Each process runs its own instance of the scheduler using a representation of the DAG that is problem size independent. The DAGuE engine utilizes all cores of each node enabling work stealing between cores of the same node. To reduce overhead, work stealing is implemented in an architecture-aware fashion and communications are made asynchonously to overlap them with computation. Communications are implicit, thus they are managed by the run-time rather than the application developer. They follow data dependencies of the DAG and do not require global synchronization, thus enabling scalability. A DAGuE user focuses on expressing the algorithm as a DAG of tasks, and defining how the tasks should be distributed over the computing resources. Tools of the framework help her in this task.

The remainder of the paper is organized as follows. Section II describes the related work, Section III contains a detailed description of the DAGuE framework. Finally, Section IV gives the experimental results and Section V provides the conclusion and future work.

II. Related Works

DAGs have a long history [1] of expressing parallelism and task dependencies in distributed systems. Previously, they have often been used in grids and peer-to-peer systems to schedule large grain tasks, mostly from a central coordinator organizing the different task executions and data movements. [2], [3]

present a taxonomy of DAGs that have been used in grid environments.

More recently, many projects have proposed to use them as an approach to address the challenge of harnessing the computing potential of multi-core computers, especially in the Linear Algebra field. In [4], [5], the authors demonstrate that DAGs enable the scheduling of tasks for tile algorithms on multi-core CPUs, reaching performances inaccessible to traditional approaches for the same problem sizes. [6] demonstrates how such an approach can also be used to address hybrid architectures, with computers augmented with accelerators like GPUs. [7] defines codelets, a task description language to enable the execution of same tasks on different hardware, and [8] uses DAGs to schedule tasks on heterogeneous computers.

We distinguish three approaches to build and manage the DAG during the execution: [3] reads a concise representation of the DAG (in XML), and unrolls it in memory before scheduling it. [9], [6], [10] modify the sequential code with pragmas, to isolate tasks that will be run as an atomic entity, and run the sequential code to discover the DAG. Optionally, these engines use bounded buffers of tasks to limit the impact of the unrolling operation in memory. The third approach consists of using the concise representation of the DAG in memory, to avoid most of the impact of unrolling it at runtime. Using structures like Parameterized Task Graph (*PTG*) proposed in [11], the memory used for DAG representation is linear in the number of task types and totally independent of the total number of tasks.

Only a few projects have tried to use DAG scheduling in distributed memory environments. Scheduling DAGs on clusters of multi-cores introduces new challenges the scheduler should be dynamic to address the non determinism introduced by communications and in addition to the dependencies themselves, data movements must be tracked. In the context of Linear Algebra, three projects are prominent: in [12], [13], the authors propose a first centralized approach to schedule computational tasks on clusters of SMPs using a PTG representation and RPC calls based on the pm2 project. [14] proposes an implementation of a tiled algorithm based on dynamic scheduling for the LU factorization on top of UPC. [15] uses a static scheduling of the Cholesky factorization on top of MPI to evaluate the impact of data representation structures. All of these projects address a single problem and propose ad-hoc solutions.

The framework described in this paper, DAGuE, takes advantage of a concise representation of the DAG; it is fully distributed, i.e. no centralized components, and avoids unrolling the DAG in memory at any given moment. Moreover, as shown in the rest of this paper, it is a general tool not dedicated to a single application.

III. The Direct Acyclic Graph Environment

DAGuE consists of a runtime engine and a set of tools to build, analyze, and pre-compile a compact representation of a DAG. The internal representation of Direct Acyclic Graphs used by DAGuE is called JDF. It expresses the different types of tasks of an application and their data dependencies.

Applications may be expressed directly as a JDF. Alternatively, most applications can also be described as a sequential SMPSS-like code, as shown in Figure 7. This sequential representation can be automatically translated in the JDF representation (described below) using our tool, H2J, which is based on the integer programming framework Omega-Test [16]. The JDF representation of a DAG is then precompiled as C-code by our framework and linked in the final binary program, with the DAGuE library.

The DAGuE library includes the runtime environment that consists of a distributed multi-level dynamic scheduler, an asynchronous communication engine and a data dependencies engine. The user is responsible for expressing the task distribution in the JDF (helping the H2J tool to translate the original sequential code in a distributed version), and distributing and initializing the original input data accordingly. The runtime environment is then responsible for finding an efficient scheduling of the tasks, detecting remote dependencies and automatically moving data between distributed resources. Below, we present in detail the input JDF format and the mechanisms involved in the scheduler as well as the communication engine to unleash the maximum amount of parallelism with dynamic and asynchronous distributed scheduling.

A. The JDF Format

The JDF is the compact representation of DAGs in DAGuE; a language used to describe the DAGs of tasks in a synthetic and concise way. A realistic example of JDF, for the Cholesky factorization that we use to evaluate the engine in Section IV, is given in Figure 1. The Cholesky Factorization consists of four basic task types: DPOTRF, DTRSM, DSYRK, DGEMM. For each operation, we define a function (lines 1 to 9 for DPOTRF) that consists of 1) a definition space (DPOTRF is parametrized by k that takes values between 0 and $SIZE-1$); 2) a task distribution in the process space (DPOTRF(k)) runs on the process that verifies the predicates of lines 5 and 6); 3) a set of data dependencies (lines 7 to 9 for DPOTRF(k): single data element); and 4) a body that holds the effective C-code that is going to be executed when this task is selected by the scheduling engine for execution (this code is omitted in the Figure 1 due to space constraints).

Dependencies beginning with a left arrow are IN dependencies for this data element; they describe how this data has been produced or how it can be found. Dependencies beginning with a right arrow are OUT dependencies for this data; when the body of this task will be completed, this data has to be transmitted to the specified task or memory location. The main goal of the scheduling engine is to select a task for which all the IN dependencies are satisfied and selects a core to run the body of the task when it is scheduled, which will enable all the OUT dependencies of this task, thus making more tasks ready to be scheduled.

Dependencies apply on data that is necessary for the execution of the task, or that is produced by the task. For example,

```
1  DPOTRF(k)  (high_priority)
2    // Execution space
3    k = 0..SIZE-1
4    // Parallel partitioning
5    : (k / rtileSIZE) % GRIDrows == rowRANK
6    : (k / ctileSIZE) % GRIDcols == colRANK
7    T <- (k == 0) ? A(k, k) : T DSYRK(k-1, k)   [TILE]
8      -> T DTRSM(k, k+1..SIZE-1)                [TILE]
9      -> A(k, k)
10
11 DTRSM(k,n)  (high_priority)
12   // Execution space
13   k = 0..SIZE-1
14   n = k+1..SIZE-1
15   // Parallel partitioning
16   : (n / rtileSIZE) % GRIDrows == rowRANK
17   : (k / ctileSIZE) % GRIDcols == colRANK
18   T <- T DPOTRF(k)
19   C <- (k == 0) ? A(n, k) : C DGEMM(k-1, n, k)
20     -> A DSYRK(k, n)
21     -> A DGEMM(k, n+1..SIZE-1, n)
22     -> B DGEMM(k, n, k+1..n-1)
23     -> A(n, k)
```

```
36 DSYRK(k,n)  (high_priority)
37   // Execution space
38   k = 0..SIZE-1
39   n = k+1..SIZE-1
40   // Parallel partitioning
41   : (k / rtileSIZE) % GRIDrows == rowRANK
42   : (n / ctileSIZE) % GRIDcols == colRANK
43   A <- C DTRSM(k, n)
44   T <- (k == 0) ? A(n,n) : T DSYRK(k-1, n)
45     -> (n == k+1) ? T DPOTRF(k+1) : T DSYRK(k+1,n)
46
47 DGEMM(k, m, n)
48   // Execution space
49   k = 0    .. SIZE-2
50   m = k+2  .. SIZE-1
51   n = k+1  .. m-1
52   // Parallel partitioning
53   : (m / rtileSIZE) % GRIDrows == rowRANK
54   : (n / ctileSIZE) % GRIDcols == colRANK
55   A <- C DTRSM(k, n)
56   B <- C DTRSM(k, m)
57   C <- (k == 0) ? A(m, n) : C DGEMM(k-1, m, n)
58     -> (n == k+1) ? C DTRSM(k+1, m) : C DGEMM(k+1, m, n)
```

Fig. 1. JDF representation of Cholesky

the task DPOTRF uses a single data element as input: T; the execution of the task modifies it, and it is also output data for this task. How the data is retrieved is described by the lines after the left arrow, while where they should be sent is described by the lines after the right arrows (there may be one left arrow and multiple right arrows per data). The input T can come either from an input memory (local to the node on which the task executes or located in a remote node), or from the output of another task (that executed locally or remotely). For example, when k=0, T of DPOTRF(k) comes from the memory in the input array A(0, 0) otherwise, T comes from the output of the task DSYRK(k-1, k). Even though the POTRF operation works in-place and overwrites A(0,0), this tile is never referenced as an input matrix. Instead, any subsequent dependent tasks refers to it as the output T from DPOTRF(0).

Output dependencies, following the right arrow, work similarly. One might notice that for output dependencies the language supports ranges of tasks as targets for a data. In this example, T is sent to the input T of DTRSM(k, k+1), and to the input T of DTRSM(k, k+2), and so on until DTRSM(k, SIZE-1). A dependency can also be a final output, meaning that no other task will modify its value before the end of this DAG. However, subsequent tasks are allowed to use this data read-only. Explicit marking of initial and final data is a desirable feature, enabling the composition of several JDFs to build complex algorithm.

The DAGuE engine is responsible for moving data from one processor to another when necessary; tasks are enabled only when all data marked as IN is locally available. Dependencies of the JDF may be marked with a modifier at their end (like [TILE] at the end of a dependency line). This modifier is a type qualifier it tells the communication engine how to transfer data from a remote location to another. By default, the communication engine uses a default data type, defined by the user to fit the basic data bloc unit (for the tiled Cholesky, this default datatype describes a single tile). Sometimes, the algorithm manipulates several different data; the user can

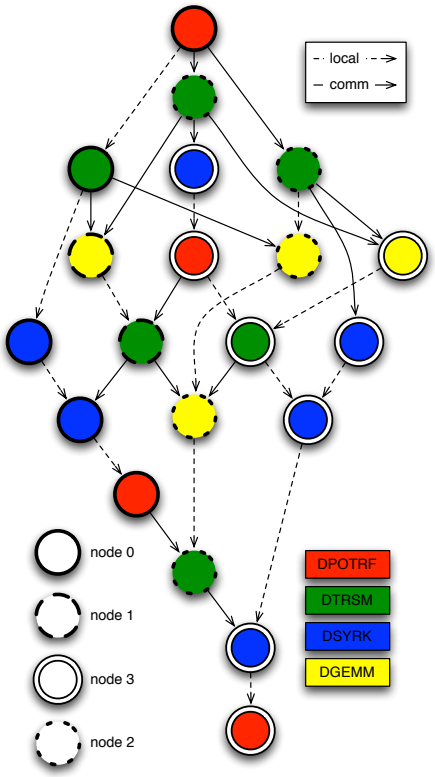


Fig. 2. DAG of Cholesky for a 4x4 tile matrix on a 2x2 grid of processors.

specify different datatypes for each in and out dependencies. For special cases, the language is flexible enough to let the user transfer the same data with different types for several tasks. This is useful to spare communication bandwidth in some Linear Algebra kernels, where typically a tile is divided in a lower and a upper triangle that flows to different tasks independently.

The internal representation of the JDF used by the DAGuE engine maps this language. The representation of Figure 2

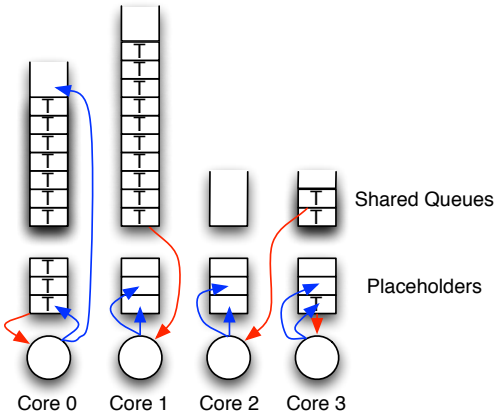


Fig. 3. Illustration of the multi-core scheduling

describes the conceptual representation of the unrolling of the JDF as a DAG. The size of the fully unrolled DAG is a function of the global parameters of the JDF representation (SIZE, GRIDrows, etc...), and its memory requirements will explode with the combination of all possible tasks. However, this is not a concern for DAGuE, as the engine never unfolds the DAG, but instead uses a symbolic interpretation to schedule tasks. The JDF is never unrolled in memory at any given time, and thus spares computation cycles to walk the DAG and memory to keep a global representation.

The IN and OUT dependencies, accessible from any task to any task, ascendant or descendant, are sufficient to implement a fully distributed scheduling engine for the underlying DAG, based only on local knowledge. When a node learns that some task has been completed remotely, it can locally compute in $O(1)$ operations what local tasks are enabled by this completion (using the global knowledge of the JDF, descending links, and process distribution predicates). To simplify scheduling operations, the JDF compiler transforms the IN dependencies into a prologue function, and the OUT dependencies into an epilogue function, around the body of the task.

B. The DAGuE engine: a fast, distributed, architecture-aware dynamic scheduler of DAG

In DAGuE, the scheduler is partly in the library and partly in the file compiled from the JDF representation. DAGuE creates one thread per core on the local machine and binds them on each core. Each thread runs its own version of the scheduler. Figure 3 represents critical structures used to do this scheduling; each thread owns a waiting queue and a bounded buffer of tasks called *placeholder*. When a thread completes a task, it executes the epilogue derived from the JDF, that determines which tasks may have been enabled by the execution of the completed task. Iterating on the outgoing dependencies of the task, the thread atomically marks in a shared structure which incoming dependencies are enabled. If some tasks become marked as enabled, the thread schedules them.

To improve locality and data reuse on NUMA architectures, the schedule function (part of the DAGuE library) favors the queuing of the new enabled tasks in the placeholder of the calling thread. A task that is enqueued in the placeholder always executes on the same thread, maximizing cache and memory locality. If the placeholder is full, the tasks are put at the beginning (for high priority tasks) or at the end (for other tasks) of the thread waiting-queue. When this epilogue is done, the thread looks up for the next task to run. The first task in the placeholder is chosen; if no task is found in the placeholder, the thread tries to pop from the beginning of its own queue. If this queue is also empty, it tries to pop from the end of the other threads' waiting-queues. The order in which the queues are considered for job stealing depends on the distance between cores. The DAGuE environment uses the HWLOC library [17] to discover the NUMA architecture of the machine at run time and adapts the stealing strategy.

In addition to dependencies tracking, DAGuE tracks data flows between tasks. When a task is completed, the epilogue stores pointers to each of the data produced by the task in a structure shared between the threads. When a new task starts, the prologue of the task uses the IN dependencies and this shared structure to retrieve each of the IN data. When all tasks depending on a particular output have completed, the engine stops tracking this data, and releases all internal resources associated with.

At the end of the epilogue, the thread has noted, using the parallel partitioning of the JDF, which tasks, if any, will execute remotely, and which data from the completed task they require. The epilogue ends with a call to the Asynchronous Communication Engine to trigger the movement of the output data to the requesting nodes. A producer/consumer approach is taken here, the orders are pushed into a queue, and the communication thread will serve them as soon as possible.

The JDF language and its internal representation at runtime, including both the generated code and the dynamic data structures, are specifically tailored to handle DAGs that enable simultaneously a large number of dependencies, called ranges. The internal dynamic structures are designed for memory efficiency and can support millions of activations with very small overheads, as we will demonstrate in Section IV.

The scheduling is fully distributed; all nodes run the scheduling engine. If necessary, each process can parse the concise JDF representation to find information about any tasks in a memory constrained space. Each computing thread runs for itself the scheduling functions, thus alleviating the need for a centralized approach of scheduling. To handle load imbalance between threads, the scheduling is dynamic and threads are allowed to steal work from one another on the same process, in a NUMA-aware way. The work-stealing approach is, however, controlled using placeholders that hold tasks that cannot be stolen from a thread, to increase data reuse.

C. Asynchronous Communication Engine

In DAGuE, communications are implicitly inferred from the data dependencies between tasks, according to the predicates defining the parallel partitioning. Asynchrony and dynamic scheduling are the key concepts of DAGuE, meaning that

the communication engine has to also exhibit those same advantages in order to effectively achieve communication/-computation overlap and asynchronous progress of tasks in a distributed environment. As a consequence, in DAGuE, communications are handled by a separate thread, which takes commands from all the computing threads and issues the corresponding network operations. Upon completion of a task, the dependency resolution function is executed on the same core that handled that computation. The distribution predicates of the downstream tasks are evaluated, and if the predicates are verified, the dependency is satisfied by the local scheduler. Otherwise, an *activate* message is sent to the process that verifies the predicate. From the compute thread perspective, this is a *fire and forget* operation. Regardless of the network congestion status, the compute threads are able to focus on the next available compute task as soon as possible to maximize communication overlap.

An activate message contains information about the task that completed (the task identifier and the values of the parameters) and the index of the output data variables needed by all the dependent tasks on the destination expressed as a single integer bit mask. During the epilogue of the task, activate messages targeting the same processor are coalesced and a single command is sent to every destination process. Only processes that will run tasks depending on the completed task are notified. As an example, on the ping-pong program presented in figure 5, when finishing PING(2), the activate message from rank 0 to rank 1 contains {PING, 2, 1}, because T is the first output of PING.

Upon the reception of an activate message, the destination process schedules the reception of the relevant output data from the parent task according to the variable mask. A single control message is sent to the originating process to initiate the data transfers; all output data needed by the destination are received by different rendezvous messages. When one of the data transfers completes, the receiver invokes locally the dependency resolution function associated with the parent, inside the communication thread, with a specific restricted mask to satisfy only the dependencies related to that particular variable. Remote dependencies resolutions are variables specific, not tasks specific, in order to maximize asynchrony. In the case of tasks enabling different tasks with different data, tasks that have received their inputs can run, regardless of the status of other outputs of the parent task. The remote dependency resolution function queues all released tasks in the ready queue of the first compute thread. As those can be stolen by any other thread, and are not already loaded into any cache, this is not detrimental to load balancing or data reuse strategies of the scheduling.

In the current version, the communications are performed using MPI. To increase asynchrony, data messages are using non-blocking, point-to-point operations to allow for several tasks to concurrently release remote dependencies. However, an uncontrolled number of concurrent messages simultaneously progressing into the MPI library leads to various issues, ranging from catastrophic aggregate bandwidth to exhaustion of the available requests of the MPI library. As a consequence, only three concurrent remote dependencies are allowed to progress at any given time, a value that was tuned to preserve the aggregate bandwidth on major MPI implementations. The MPI thread pre-posts enough persistent receives to handle the control messages for the maximum number of concurrent tasks completion. There is no limit to the number of control messages that can be sent, to avoid deadlocks. This can generate unexpected messages, but only for small size messages, and does not consume requests. Due to the rendezvous protocol described in the previous paragraph, the data payload of the variables are never unexpected, thus reducing memory consumption from the network engine and ensuring flow control.

The MPI thread is not intended to be running on its own physical core. Therefore, in order to decrease the level of noise it generates on the computing threads, the MPI thread periodically invokes `nanosleep` in order to yield the processor to real computation. The trade-off for this lower overhead on computation lies in the inability of this approach to benefit from the smallest possible latency. The rationale of this choice comes from the granularity of the tasks in the target applications, whose performance are more bandwidth rather than latency constrained. While MPI is a portable communication environment, it doesn't fit well with totally asynchronous progressions, where a model such as Active Messages fits better. This will be addressed in future work.

IV. Performance Evaluation

A. Experimental conditions

The Griffon cluster is one of the clusters of the Grid'5000 experimental grid [18]. It is a 648 core machine composed of 81 dual socket Intel Xeon L5420 quad core processors at 2.5GHz with 16GB of memory, interconnected by a 20Gbs Infiniband network. Linux 2.6.24 (Debian Sid) is deployed.

The Dancer cluster is a small 8 quad core node cluster, based on a Intel Q9400 2.5Ghz processor, each node with 4GB of memory. All nodes are connected using a dual Gigabit Ethernet links, and four additionally sports Myricom MX10G. Linux 2.6.31.2 (CAoS NSA) is deployed.

On Dancer and Griffon, the software is compiled using gcc and gfortran 4.4 with -O3 flags, and uses the OpenMPI 1.4.1, Plasma 2.1.0 and Intel MKL-10.1.0.015 libraries.

B. Micro benchmarking

1) Scheduling Performance: The first results evaluate the overheads of the scheduling engine of DAGuE on a single node architecture. Two different simple benchmarks compute Nb repetitions of a simple task, consisting of a $N \times N$ double precision matrix-matrix multiply. The first benchmark is a sequential program composed of four nested loops (one loop around Nb, then the three loops of the matrix-matrix multiply). The second benchmark is a simple JDF file that generates Nb parallel tasks consisting of the three inner loops of the matrix multiplication.

The Figure 4 plots the ratio between the time taken by the sequential program with ideal scaling (hence $1/p$ of the

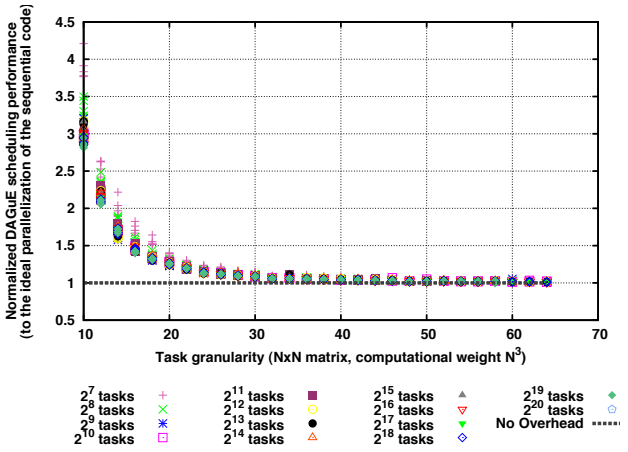


Fig. 4. Ratio between the time taken by the DAGuE engine to schedule Nb matrix-matrix multiply of size $N \times N$ and the time taken by the similar sequential code divided by the number of cores (ideal parallelization)

```
1  PING(k)
2   // Definition domain of the parameters
3   k = 0 .. NT
4
5   // Parallel partitionning
6   : 0 == rowRANK
7
8   T <- (k==0) ? A(0) : I PONG(k-1)      [ATYPE]
9     -> (k==NT) ? A(0) : I PONG(k)       [ATYPE]
10
11 PONG(k)
12  // Definition domain of the parameters
13   k = 0 .. NT-1
14
15  // Parallel partitionning
16  : 1 == rowRANK
17
18   I <- T PING(k)                       [ATYPE]
19     -> T PING(k+1)                     [ATYPE]
20
```

Fig. 5. JDF representation of the ping pong

time measured, where p is the number of cores), and the time taken by the DAGuE engine for the same number of tasks Nb and matrix size N. We did all measures on the dancer platform, five times, and divided each measurement of the DAGuE engine by the fastest sequential run for the same parameters. A similar benchmark using a simple Posix threads parallelization, not presented here due to lack of space, supports the same conclusions.

The embarrassingly parallel matrix-matrix multiply is a stress test for the scheduling engine of DAGuE. An extremely large number of tasks (up to 2^{20}) can be scheduled at the same time. Thus, the waiting queue of the engine is rapidly filled with ready tasks that have to be scheduled. Thanks to not unfolding the complete graph, the engine is able to manage millions of simultaneous tasks without impacting the memory consumption or the computation time. For very small task (in the order of microseconds), the overheads due to dynamic scheduling can take up to twice the ideal execution time, suggesting that DAGuE is best fitted for tasks of a coarser grain. However, the overheads due to the scheduling infrastructure become rapidly negligible; for a relatively small work size (a matrix-matrix multiply of 30×30 doubles takes $44\mu s$ on the dancer platform), DAGuE reaches the ideal parallelization performance projection.

2) Communication Performances: The second benchmark aims at evaluating the communication performance of the DAGuE engine. We have designed a very simple ping pong benchmark where a message of variable size is sent from one node to another node, a variable number of times. The JDF representation of this ping pong is located in Figure 5. Node 0 (identified by the predicate `0 == rowRANK`) is the only one to execute the `PING(k)` task, transmitting a data to the `PONG(k)` task that can execute on node 1 only. This data is typed with the non-default type `ATYPE`, that is allocated to the desired size by the main program. `PING(0)` reads its data locally while `PING(k)` $(k > 0)$ uses the data sent by `PONG(k-1)`.

We measure the total time t taken to execute this JDF on two machines, interconnected with 2 Gb/s ethernet cards, then with the Myricom MX-10G high-speed network, and finally on the Griffon platform with two machines interconnected with Infiniband 20 Gb/s. From this time t we compute the latency $(t/(2 \cdot NT))$ and the bandwidth $(2 \times 8 \cdot NT \cdot S/t)$ of the DAGuE engine, where NT is the number of iterations and S is the size of the data in bytes. In Figure 6 we compare these measurements with the NetPIPE [19] benchmark using directly the same MPI library as DAGuE.

Figure 6(a) demonstrates a high overhead on latency for the DAGuE benchmark, for all kinds of networks: from a factor of 10 on the double-1G Ethernet network to a factor of 90 on the MX-10G network. The implementation of the RTT benchmark in DAGuE consists of placing a string of tasks alternatively between two nodes, and allowing the DAGuE engine to move the data for the tasks. The current implementation of DAGuE uses a 3-way rendezvous protocol to move all data; the emitter first signals the completion of the task to the nodes that will run a task depending on this completion. The receiver node, when notified of a completion, allocates resources to receive the actual data, then requests the data from the emitter, that finally sends the data. For very small messages, this multiplies the latency by at least a factor of 3. Moreover, the goal of the DAGuE engine is to resolve data dependencies and move data for the upper layer application. To do this, the engine introduces an accounting of data and allocates memory to receive the new data. So, all network data are received in a newly allocated buffer that will be garbage collected by the system. Furthermore, the communications and the treatment of the tasks are done on different threads, addingfour4 to six thread context switches to the latency. This is a different behavior than the NetPIPE benchmark, which receives and sends data "in-place" and does not use threads. For high-speed networks this introduces a significant overhead that explains the observed difference.

However, the DAGuE system is not designed to move small data, but data of the order of magnitude of a matrix tile. Figures 6(a) and 6(b) also show that for medium-size messages (64KB), the difference between NetPIPE and DAGuE is small

```
FOR k = 0..TILES-1
  A[k][k] ← DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    A[m][k] ← DTRSM(A[k][k], A[m][k])
  FOR n = k+1..TILES-1
    A[n][n] ← DSYRK(A[n][k], A[n][n])
    FOR m = n+1..TILES-1
      A[m][n] ← DGEMM(A[m][k], A[n][k], A[m][n])
```

Fig. 7. Pseudocode of the tile Cholesky factorization (right-looking version).

for the Ethernet network, and it becomes small at 512KB for high-speed networks. For the tested applications, the tile size resulting from tuning varies from 200×200 (320KB) to 350×350 (≈1MB), which is in the high network efficiency range.

C. Application Benchmarking

Cholesky Factorization: The Cholesky factorization (or Cholesky decomposition) is mainly used for the numerical solution of linear equations $Ax = b$, where A is symmetric and positive definite. This factorization of an $n \times n$ real symmetric positive definite matrix A has the form

$$A = LL^T,$$

where L is an $n \times n$ real lower triangular matrix with positive diagonal elements. Due to its large recognition, we used this factorization as a first use case for the environment. We have implemented a tiled algorithm version of the Cholesky factorization. As described in [20], a single step of the algorithm is implemented by a sequence of calls to the LAPACK and BLAS routines: DSYRK, DPOTF2, DGEMM, DTRSM. Due to the symmetry, the matrix can be factorized either as upper triangular matrix or as lower triangular matrix. The tile Cholesky algorithm is identical to the block Cholesky algorithm implemented in LAPACK, except for processing the matrix by tiles. Otherwise, the exact same operations are applied. The algorithm relies on four basic operations implemented by four computational kernels:

DPOTRF: The kernel performs the Cholesky factorization of a diagonal (triangular) tile T and overrides it with the final elements of the output matrix.

DTRSM: The operation applies an update to a tile A below the diagonal tile T, and overrides the tile A with the final elements of the output matrix. The operation is a triangular solve.

DSYRK: The kernel applies an update to a diagonal (triangular) tile B, resulting from factorization of the tile A to the left of it. The operation is a symmetric rank-k update.

DGEMM: The operation applies an update to an off-diagonal tile C, resulting from factorization of two tiles A to the left of it. The operation is a matrix multiplication.

Figure 7 shows the pseudocode of the Cholesky factorization (the right-looking variant).

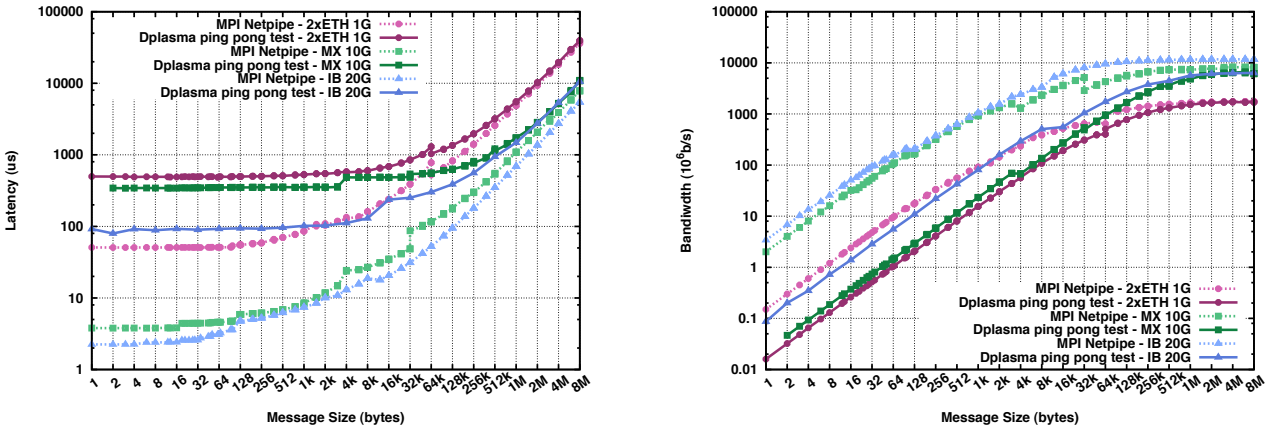
A parallel Cholesky factorization implementation is controlled by several parameters: N defines the size of the input matrix ($N \times N$ doubles), while NB defines the size of the tiling (or blocking). A $N \times N$ matrix is divided in $NT \times NT$ tiles (or blocks) where $NT \times NB = N$. When NB does not divide N, the last tile of each row or column is padded with zeroes. No computation happens on the padding but complete tiles are transferred over the network nonetheless. Two other parameters, P and Q, control the process grid used to map the block cyclic distribution of the tiles (or blocks) on the computing resources. According to [21] and to our experiments, the best performance is achieved when using a process grid that is square or closest to square with $P \leq Q$. Consequently, for all the results presented in this paper, the process grid follows this rule. NB has been tuned experimentally for each software, the results are generated using the best overall performing NB.

In the rest of the paper, for all figures that present performances in GFLOP/s, we provide the theoretical performance of the platform computed as the frequency of a core, times the depth of the pipeline of the core, times the number of cores. We also provide the *GEMM peak* performance of the platform. GEMM peak is measured as the best performance obtained by a single core to compute a double precision matrix-matrix multiply using the same numerical library as the Cholesky factorization (BLAS), while the other cores are computing independent, identical, GEMMs. This is considered as the practical peak performance of the platform, and this is the operation that dominates the Cholesky factorization. All benchmarks that follow only consider double precision operations.

ScaLAPACK and DSBP: We compare the performances of the Cholesky factorization with two other implementations. ScaLAPACK [22] is the reference implementation for distributed parallel machines of some of the LAPACK routines. Like LAPACK, ScaLAPACK routines are based on block partitioned algorithms to improve cache reuse and reduce data movement. We used the vendor ScaLAPACK and BLAS implementations (from MKL).DSBP [15] is a tailored implementation of the Cholesky factorization using 1) a tiled algorithm, 2) a specific data representation suited for Cholesky, and 3) a static scheduling engine. We used DSBP version 2008-10-28[1].

1) Impact of task granularity: In In Figure 8, we investigate the effect of task granularity on the performance of the DAGuE Cholesky Factorization at different node scales and input matrix sizes. For each run, we took the smallest matrix size that is bigger than a target T and still divisible by the block size. For one node, the target T_1 is $13,600$; for four nodes, the target T_4 is $26,880$; for 81 nodes, the target T_{81} is $120,000$. Each of these sizes is chosen to exhibit the peak performance of the DAGuE Cholesky implementation on the different setups. To compare all runs in a normalized way, the figure represents the efficiency as a percentage of the theoretical peak for each setup.

[1] available online at http://www8.cs.umu.se/~larsk/index.html



(a) Latency (b) Bandwidth

Fig. 6. Round-Trip benchmark – comparison of DAGuE and NetPIPE on Ethernet, Myricom and Infiniband networks.

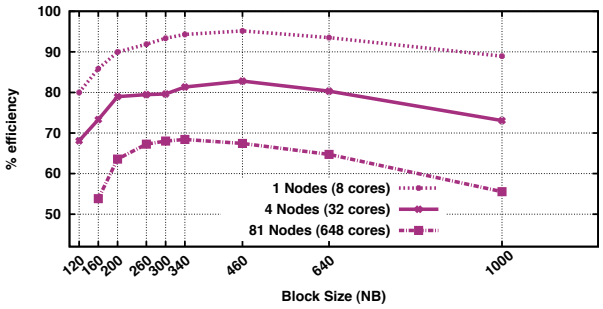


Fig. 8. Performance (relative to the theoretical peak) of the DAGuE Cholesky Factorization as function of the Tile Size (Griffon platform).

All curves present the same general shape: the performance first increases with the block size until a peak, then decreases slowly when the block size increases. For a single node, this is the effect of the optimization of cache effects in the BLAS kernel. For a distributed run, the optimal block size is the result of a trade-off between an ideal size for optimizing the cache effects in the kernel, and network efficiency. As seen in Figure 6, starting at 1MB, the DAGuE engine reaches network saturation. Thus, for blocks of 360×360 elements and larger, the transfer time increases linearly with the amount of data (thus as the square of the block size). Smaller block sizes experience a lower network efficiency. However, when the size of the matrix is large, there are enough tasks ready to be scheduled at all times to overlap communication costs with computation, and as a consequence, block size tuning mostly depends on the BLAS kernels.

One can see however that for 81 nodes, the best NB value is 340, while it is 460 for one node. A distributed runs require communications which themselves introduce memory copies, that pollute parts of the cache. Since the cache is not used exclusively by the BLAS kernels, the best block size decreases slightly and thus increases the probability that a tile will fit in some cache of the computing nodes, even if the MPI thread is using part of this to handle communications.

2) Problem Scaling: Figure 10 presents the performance of the Cholesky Factorization when scaling the problem size. We ran the different Cholesky Factorizations on the Griffon platform, with 81 nodes (648 cores), and for a varying problem size (from $13,600 \times 13,600$ to $130,000 \times 130,000$). We took the best block value for each of the implementations; block sizes were tuned as demonstrated in Figure 8 for the DAGuE implementation. We kept the best value of the runs for each plot in the figure.

When the problem size increases, the total amount of computation increases as the cube of the size, while the total amount of data increases as the square of the size. For a fixed block size, this also means that the number of tiles in the matrix increases with the square of the size, and so does the number of tasks to schedule. Therefore, the global performance of each benchmark increases until a plateau is reached. On the Griffon platform, the amount of available memory was not sufficient to reach the plateau with neither of the implementations.

The figure shows that for small size problems, DSBP obtains a better performance than DAGuE. DSBP is using a data format specifically tailored for the Cholesky factorization (exploiting the symmetry of the matrix). As a consequence, DSBP does not require as much parallelism as DAGuE to overlap the communications with computation. When DAGuE has enough data per node to overlap all communication with computation, the dynamic scheduling of DAGuE utilizes the computing resources and the network better, up to 70% of the theoretical peak (75% of GEMM-peak).

3) Impact of intra-node versus inter-node communication: Figure 9 presents the performance per core, for a fixed total number of cores, when varying the repartition between distributed memory and shared memory accesses. Even using the inefficient Ethernet network, the performance per core decreases only slightly when replacing shared memory computation by MPI distributed messaging, outlining the nearly perfect overlap achieved by the communication engine.

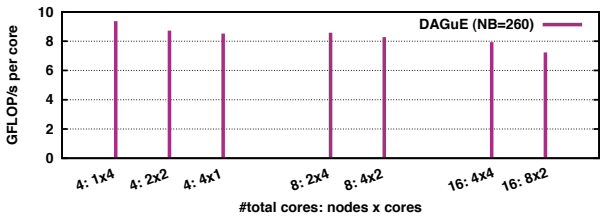


Fig. 9. Performance comparison at fixed total number of cores between distributed and shared memory performance with N=18200 (Dancer platform, 2xGEthernet).

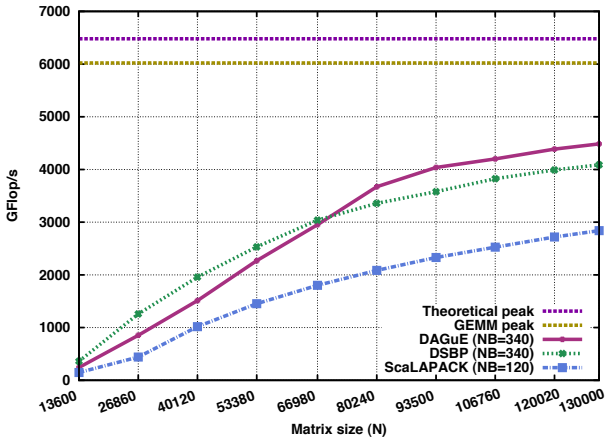


Fig. 10. Problem Scaling of the Cholesky Factorization, on 81 nodes (Griffon platform).

4) Weak Scalability: Figure 11 presents the weak scalability study of the Cholesky Factorization. The initial workload for a single node (8 cores) experiment is a $13,600 \times 13,600$ matrix. This matrix size is scaled up accordingly to the number of nodes to keep the per core workload constant, up to $N = 120,000$ for an 81 node (648 cores) deployment.

As one can see, all benchmarks scale almost perfectly, attaining 49% of the GEMM peak for ScaLAPACK, 66% for DSBP, and up to 78% for the DAGuE engine. All runs in the figure are done with a square process grid, which is the best process grid for Cholesky factorization. The only exception is the point at 384 cores (48 nodes, 8 cores per node). In this case, we used a process grid of 6×8 for the DAGuE engine, and 16×24 for DSBP and ScaLAPACK. This measurement was added to demonstrate that all benchmarks suffer from a similar downgrade of performance when the grid is not perfectly square.

5) Strong Scalability: Figure 12 presents the strong scalability study for the Cholesky factorization (i.e., evolution of the performance for a given matrix size, when increasing the number of computing resources participating in the factorization). For Figure 12(a), we used the largest available matrix size for the smallest number of nodes ($93,500 \times 93,500$) and the most efficient block size after tuning (340×340). For Figure 12(b), we always used the same number of nodes (81), but varied only the number of cores, so we chose the smallest matrix size for which benchmarks were able to obtain the best

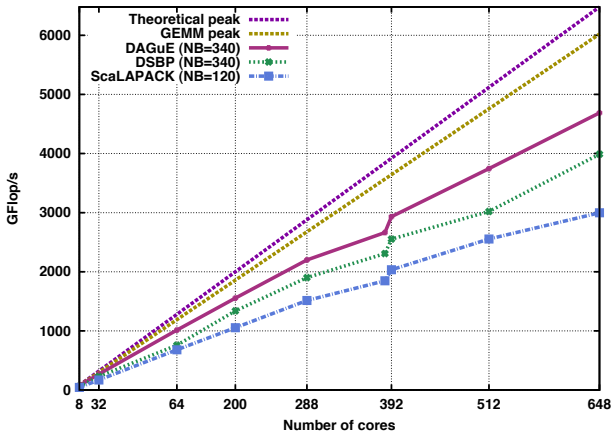


Fig. 11. Weak Scalability of the Cholesky Factorization, starting from N=13,600 for 8 cores (Griffon platform).

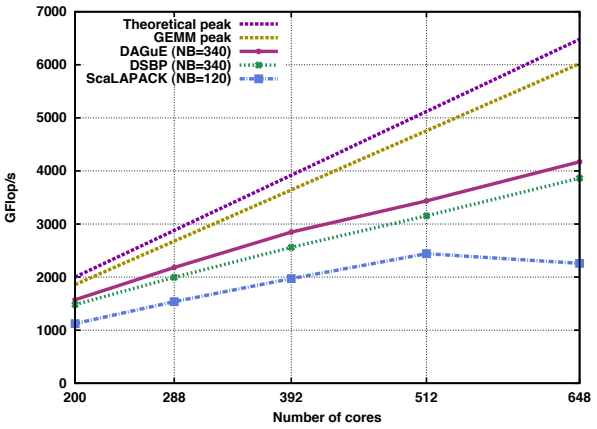
performances ($120,020 \times 120,020$).

The figure shows that, for a fixed matrix size, the performances of both tiled factorization implementation (DAGuE and DSBP) scale almost linearly. Because the same matrix is distributed on an increasing number of nodes, the ratio between computations and communications decreases with the number of nodes. As a consequence, the efficiency of the benchmark decreases when the number of cores increases. ScaLAPACK seems to suffer more from this effect, and is consequently unable to continue scaling after 512 cores for this matrix size.

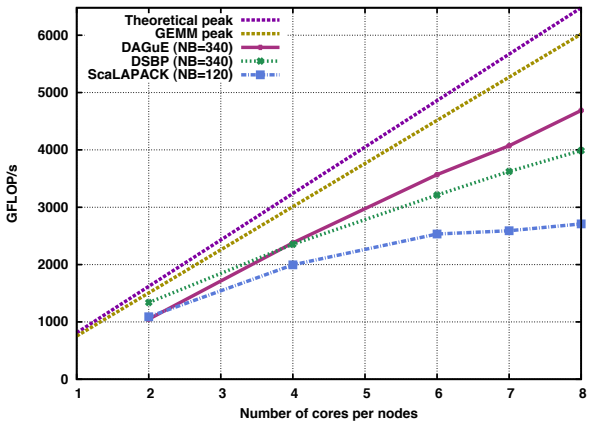
Figure 12(b) illustrates that the DAGuE and DSBP approaches are best fitted for clusters with many cores. We were able to run on a larger matrix because even at 2 cores per node, the whole memory of the 81 nodes is available. As shown in [15], DSBP data representation enables it to outperform ScaLAPACK. Because DAGuE is designed as a hybrid system, it scales linearly with the number of cores, as long as enough parallelism enables to feed all the threads. At 2 cores per node, the ad-hoc data representation of DSBP is more beneficial than the scaling provided by the hybrid and more generic approach of DAGuE. However, for larger core counts per node, the dynamic scheduling of DAGuE exhibits a better use of the local computing resources, allowing it to surpass DSBP.

6) Generality of DAGuE: Because the existence of DSBP gives a comparison point against a similar tiled factorization algorithm, but using a static scheduling, we mostly focused our results on the Cholesky factorization. However, the DAGuE framework has also been used to implement two other well known Linear Algebra factorization algorithms: the tiled version of QR [23] and the tiled version of LU [24]. To demonstrate the generality and applicability of the DAGuE framework, the Table I presents early results obtained with those different algorithms at large scale on the Kraken XT5[2] machine of the University of Tennessee and Oak Ridge National Laboratory. Studying these algorithms is outside the scope of this paper, however the full study may be found in [25].

[2]http://www.nics.tennessee.edu/computing-resources/kraken



(a) Varying the number of nodes for N=93,500.



(b) Varying the number of cores per node, with 81 nodes and N=120,020

Fig. 12. Strong Scalability of the Cholesky Factorization (Griffon platform)

	QR	LU	Cholesky
Number of cores	$5,808$	$3,072$	$3,072$
Size of the problem	$633,600^2$	$454,000^2$	$454,000^2$
Time of execution	8,039s	3,348s	2,013s
Efficiency (% theoretical peak)	69.8%	58.3%	48.5%

TABLE I
LARGEST RUN AND EFFICIENCY OF OTHER APPLICATIONS WRITTEN USING THE DAGUE FRAMEWORK (KRAKEN XT5)

V. CONCLUSION

With the emergence of massively multicore architectures, the classical approach based on MPI SPMD programming model tends to become inefficient. Problems with memory bandwidth, latency and cache fragmentation will, therefore, tend to become more severe, resulting in communication imbalance. Furthermore, network bandwidth (between parallel processors) and latency are improving, but at significantly different rates than the increase of operations per second performed by the CPU. Specificaly, network bandwidth and latency improve by 26%/year and 15%/year respectively, while processing speed increases by 59%/year. Therefore, the shift in algorithm properties, from computation-bound toward communication-bound is expected to become even more evident in the near future. This is demonstrated by our experiments by the fact that ScaLAPACK, a very efficient, but 20 year old software package, underperforms on modern architectures. The DAGuE engine proposed in this paper tackles this problem by proposing a generic DAG engine to express task dependencies at a finer granularity. By specificaly targeting clusters of multi-cores, with a hybrid programming model mixing explicit message passing and multi-threaded parallelism DAGuE automatically extracts more asynchrony from the algorithms, and therefore brings the application performance closer to the physical peak. Moreover, algorithms expressed as DAGs have the potential to alleviate the user from focusing on the architectural issues, while allowing the engine to extract the best performance from the underlying architecture.

In this paper, the DAGuE engine performance has been investigated using synthetic benchmarks, underlining a very good efficiency from a task granularity of a few microseconds. The Cholesky factorization has been implemented using the JDF representation to demonstrate the performance of the system on a realistic workload. The performance of this algorithm has been compared to the classical approach for distributed systems programming, represented by the Cholesky ScaLAPACK algorithm, and a similar optimized version of the tiled Cholesky algorithm called DSBP. The DAG/Tiled algorithm approach clearly outperforms ScaLAPACK, both in terms of scalability and performance, with an efficiency almost doubled in certain instances. Besides being generic, because it benefits from more asynchrony from its dynamic and cache aware scheduling, in most cases the DAGuE engine compares favorably in terms of performance against the Cholesky specific DSBP tiled algorithm implementation.

Some of the experimental results suggest that even more performance could be achieved with a better handling of collective communications. Especially at small matrix sizes, the ratio between the volume of communications from the source of a broadcast and the amount of avaliable computations to overlap it is not always sufficient. Expressing the collective communication as a JDF task is a very elegant and promising way of allowing asynchronous progress of the communications. Features such as this will be investigated in our future work. While solving linear systems is of extreme importance to the community, other families of important algorithms can benefit from DAGuE, such as stencil algorithms, sparse linear algebra, FFT, etc. We are hopeful that the DAGuE engine can provide similar performance improvement for those types of problems as it did for dense linear algebra.

REFERENCES

[1] J. A. Sharp, Ed., *Data flow computing: theory and practice*. Ablex Publishing Corp, 1992.

[2] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," Journal of Grid Computing, Tech. Rep., 2005.

[3] O. Delannoy, N. Emad, and S. Petiton, "Workflow global computing with yml," in *7th IEEE/ACM International Conference on Grid Computing*, september 2006.

[4] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The impact of multicore on math software," in *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA*, ser. Lecture Notes in Computer Science, B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, Eds., vol. 4699. Springer, 2006, pp. 1–10.

[5] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 116–125.

[6] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, 2009.

[7] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par 2009 Euro-par'09 Proceedings*, ser. LNCS, Delft Pays-Bas, 2009. [Online]. Available: http://hal.inria.fr/inria-00384363/en/

[9] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE International Conference on*, 29 2008-oct. 1 2008, pp. 142 –151.

[10] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11, DOI: 10.1145/1654059.1654079.

[11] M. Cosnard and E. Jeannot, "Automatic Parallelization Techniques Based on Compact DAG Extraction and Symbolic Scheduling," *Parallel Processing Letters*, vol. 11, pp. 151–168, 2001. [Online]. Available: http://dx.doi.org/10.1142/S012962640100049Xhttp://hal.inria.fr/inria-00000278/en/

[12] M. Cosnard, E. Jeannot, and T. Yang, "Compact dag representation and its symbolic scheduling," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 921–935, August 2004.

[13] E. Jeannot, "Automatic multithreaded parallel program generation for message passing multiprocessors using parameterized task graphs," in *International Conference 'Parallel Computing 2001' (ParCo2001)*, september 2001.

[14] P. Husbands and K. A. Yelick, "Multi-threading and one-sided communication in parallel lu factorization," in *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*, B. Verastegui, Ed. ACM Press, 2007.

[15] F. G. Gustavson, L. Karlsson, and B. Kågström, "Distributed SBP cholesky factorization algorithms with near-optimal scheduling," *ACM Trans. Math. Softw.*, vol. 36, no. 2, pp. 1–25, 2009.

[16] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 1991, pp. 4–13.

[17] 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 *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE, Ed., Pisa Italy, 02 2010. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00429889/en/

[18] R. Bolze, F. Cappello, E. Caron, M. J. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *IJHPCA*, vol. 20, no. 4, pp. 481–494, 2006.

[19] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, "Netpipe: A network protocol independent performance evaluator," in *IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[20] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, 2008.

[21] J. Choi, J. Demmel, I. S. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley, "Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance," in *Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, Second International Workshop, PARA '95, Lyngby, Denmark, August 21-24, 1995, Proceedings*, ser. Lecture Notes in Computer Science, J. Dongarra, K. Madsen, and J. Wasniewski, Eds., vol. 1041. Springer, 1995, pp. 95–106.

[22] L. S. Blackford, J. Choi, A. J. Cleary, E. F. D'Azevedo, J. Demmel, I. S. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley, "ScaLAPACK: A linear algebra library for message-passing computers," in *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1997.

[23] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled qr factorization for multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 13, pp. 1573–1590, 2008.

[24] ——, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, 2009.

[25] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Distributed-memory task execution and dependence tracking within DAGuE and the DPLASMA project," Innovative Computing Laboratory, University of Tennessee, Technical Report ICL-UT-10-02, 2010.