Parallel Computing 38 (2012) 37–51

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

DAGuE: A generic distributed DAG engine for High Performance Computing ☆

George Bosilca [a], Aurelien Bouteiller [a,*], Anthony Danalis [a], Thomas Herault [a], Pierre Lemarinier [b], Jack Dongarra [a,c]

[a] Innovative Computing Laboratory, The University of Tennessee, United States
[b] IRISA, Université de Rennes 1, France
[c] Oak Ridge National Laboratory, United States

ARTICLE INFO

Article history:
Available online 19 October 2011

Keywords:
HPC
Micro-task DAG
Heterogeneous architectures
Architecture aware scheduling

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 is 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 expressed 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.

Published by Elsevier B.V.

1. 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 accelerators and 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.

To achieve this, an MPI/threads hybrid programming model is the 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. Instead of allowing application programmers to focus on algorithmic issues, it encumbers their task with several low level architectural issues such as load balancing, memory distribution, cache reuse and memory locality on non-uniform memory access (NUMA) architectures, and communications/computations overlapping. From a performance

☆ This work is an extended version of an article presented at the 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2011).

* Corresponding author.
E-mail addresses: bosilca@eecs.utk.edu (G. Bosilca), bouteill@eecs.utk.edu (A. Bouteiller), danalis@eecs.utk.edu (A. Danalis), herault@eecs.utk.edu (T. Herault), lemarini@eecs.utk.edu (P. Lemarinier), dongarra@eecs.utk.edu (J. Dongarra).

0167-8191/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.parco.2011.10.003

portability point of view these issues are hard to address in a generic way, and are yet orthogonal to the algorithm design computational scientists are interested in.

In this paper, we present *DAGuE*, a framework for parallel application developers, that transfers 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 movements between tasks. Designing a parallel application with the *DAGuE* framework consists of encapsulating computation tasks into sequential kernels and defining, through a DAGuE specific language, how the flow of data circulates from one kernel into the next. The algorithm and the data distribution are decoupled, the runtime system is responsible of mapping the algorithm on the data at runtime.

DAGuE schedules tasks in a fully distributed and dynamic fashion among all computing resources (cores, GPUs, accelerators). It enables local tasks to make progress waiting only on data dependencies with 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. Communications are implicit, depicted solely by he data flow between tasks executing on different nodes, 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 via the data distribution. Tools of the framework help her in this task.

The remainder of the paper is organized as follows. Section 2 overviews related work, Section 3 contains a detailed description of the DAGuE framework. Section 4 describes the algorithms ported to DAGuE. Finally, Section 5 gives the experimental results and Section 6 provides the conclusion and future work.

2. Related work

As early as 1966, the Bernstein conditions, which formalize the requirements for two tasks to be executed in parallel, have been isolated [1]; Direct Acyclic Graphs are a convenient abstraction of these conditions, with a large variety of applications [2]. More specifically, in a distributed context, the dataflow execution model is iconic of DAG based approaches [3], which have mostly been applied, in the last two decades, to grid and peer-to-peer systems [4,5].

Recently, several projects [6–10], mostly in the field of linear algebra, have proposed to revive the general use of DAGs, as an approach to tackle the challenges of harnessing the power of multi-core and hybrid platforms. We distinguish three approaches to building and managing the DAG during execution: [5] reads a concise representation of the DAG (in XML), and unrolls it in memory before scheduling it. Perez et al. [11], Agullo et al. [8], Song et al. [12], and Augonnet et al. [13] modifies the sequential code with pragmas, to isolate tasks that will be run as an atomic entity, and runs 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 minimize the impact of unrolling the complete DAG. Using structures such as a Parameterized Task Graph (*PTG*) proposed in [14], the memory used for DAG representation is linear in the number of task types and totally independent of the total number of tasks.

However, only a few of these recent 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 between nodes. In the context of linear algebra, three projects are prominent: in [15,16], the authors propose a centralized approach to schedule computational tasks on clusters of SMPs using a PTG representation and RPC calls based on the PM2 project. Husbands and Yelick [17] proposes an implementation of a tiled algorithm based on dynamic scheduling for the LU factorization on top of UPC. Gustavson et al. [18] 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.

In comparison, 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. The algebraic representation of the dependencies enables out-of-order execution of the tasks, only limited by the actual data dependencies, and not by the original control flow of the input language, hence extracting more potential parallelism. At best of our knowledge no such approach has been taken in the past. Moreover, as shown in the rest of this paper, DAGuE is a general tool not dedicated to a single application or platform.

3. The Direct Acyclic Graph environment

We consider the following three challenges as the most important when considering DAG scheduling on emerging high performance architectures, consisting of many multi-cores NUMA computing nodes, clustered in a large distributed memory machine. Data distribution must be left to the programmer, to enable control over the communication volume and to ease composition with other software layers; inside a node, scheduling must be completely dynamic and asynchronous, to

maximize throughput and load balance; communication between nodes should be asynchronous and reactive, to overlap as much as possible computation and communication and to avoid costly distributed synchronizations.

To achieve these goals, DAGuE consists of a runtime engine and a set of tools to build, analyze, and pre-compile a compact representation of a DAG. The DAGuE library includes the runtime environment which is formed by the assembly of the generic runtime code, and the program-specific codes generated by the compile phases. The runtime library encompasses a distributed multi-level dynamic scheduler, an asynchronous communication engine and a data dependencies engine, all deeply integrated into the generated code of the considered routine. A typical DAGuE application is a regular MPI application, that links with a DAGuE optimized computational routines generated with the DAGuE toolchain. The end-user programmer does not necessarily have to use DAGuE, as the routines can be embedded in interoperable libraries.

The representation of Fig. 1 describes the conceptual representation of the unrolling of the DAG, as seen by the runtime. Although the size of the fully unrolled DAG is a function of the global parameter space (SIZE, etc...), for small problem sizes, the DAG can be entirely unrolled in memory, and can be handled by well established management techniques. However, considering larger scale machines envisioned in the future, Gustafson's law calls for much larger problem sizes, for which the memory requirements would explode with the combination of all possible tasks, far exceeding the available memory. The traditional workaround for this issue, as seen in StarPU, PLASMA, Cilk, etc. is to restrict the exploration of the DAG to a bounded window. This interferes with the exploitation of potential parallelism, as the filling of the window follows the ordering dictated by the sequential code flow, executed on all compute nodes, a serious limitation on million way parallel machines envisioned in the future. This is not a concern for DAGuE, as the engine never unfolds the DAG. Instead, it uses a symbolic interpretation, similar to the algebraic representations depicted in Fig. 3, to explore successors, generate and schedule ready-tasks. Hence, at any given time, the DAG is never entirely unrolled in memory, only tasks ready to be executed on the local resources exists in the system. Furthermore, each compute node computes only successors for tasks that are locally executed (unlike the code flow approach that conceptually unrolls the entire DAG on all nodes).

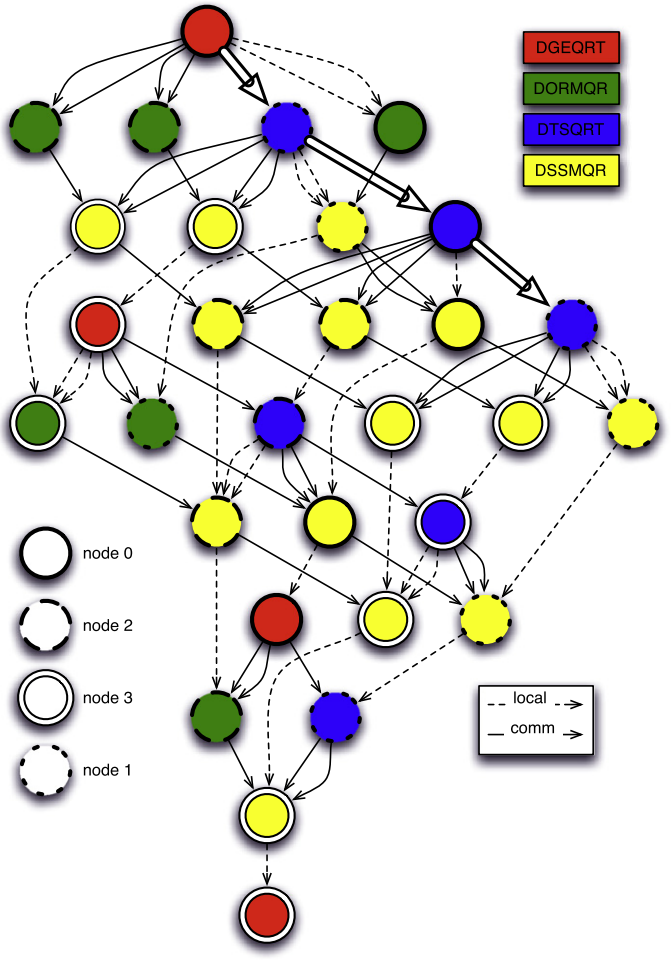


Fig. 1. DAG of QR for a 4×4 tiles matrix on a 2×2 grid of processors.

The IN and OUT dependencies, are accessible between any pair of tasks that have a dependent relation, in the successor or predecessor direction. If task A modifies a data d_A and passes it as data d_B to task B as read-only, task A can compute that task B is part of its successors; task B can compute that task A is part of its predecessors; and both tasks know what data is accessed how by the other task. This is sufficient knowledge 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 $O(d)$ operations what local tasks are enabled by this completion, d being the output degree of this task in the DAG. This computation is independent of the total dimension of the underlying DAG, the problem size, the number of task or total number of edges. Its memory requirements is also bounded by the number of tasks generated. To accelerate the scheduling operations, the IN dependencies are compiled into a prologue function, and the OUT dependencies into an epilogue function, inserted around the computational body of the task.

3.1. Input format

The input format of the DAGuE framework can be either sequential pseudo-code, consisting of simple loop nests (see Fig. 2), or directly the intermediate Job Data Flow representation (JDF). In the case of the sequential pseudo-code, it is translated in the JDF format, using a provided tool (H2J) based on the integer programming framework Omega-Test [19] to extract data dependencies. The JDF format, is a textual representation of the algorithm, each task being divided into two parts: the dataflow representation and the code body. The code body is a simple translation of the kernel call in C, and can be customized by the programmer. The DAG dependencies are represented separately by algebraic conditions on inputs and outputs of each tasks, as illustrated by the JDF code generated from the aforementioned pseudocode for the GEQRT kernel, presented in Fig. 3. In this example, if the parameter $k = 0$, the value of the V data is read from the local A matrix, otherwise, it comes from the C2 output of the DSSMQR kernel. Inputs and outputs can be typed (U depends only on the upper triangle of the data, L on the lower triangle, and T on the entire data). In order to fine tune the algorithm, the programmer can alter an existing JDF, or create it from scratch for simple cases. Adding a task priority function to the tasks, which computes a priority hint based on the task's parameters, is a typical example of JDF alterations for tuning purposes.

The JDF representation is then pre-compiled into a C-code and becomes available as an object file. The pre-compiler generates a user visible C function, based on the JDF name, taking all global variables as arguments. This function represents the interface used by the caller of the routine; it includes, in addition to the code provided by the user in the body, an epilogue and a prologue. These two entities are composed by a set of stubs necessary to the DAGuE runtime to retrieve the local or remote data required by the task (in the prologue); to execute the task's body and to infer and trigger all local or remote successors of the task (in the epilogue). The C-code generated at this step remains problem-size independent. It is a symbolic representation of the DAG, that maps very closely the textual representation of the JDF. After he set the parameters values using the main entry point provided by the DAGuE translator, the user gets an object that can iterate on the underlying DAG, to explore it if necessary. Each task discovered by the process that walks the DAG can be used to compute the tasks that are successors. Since the link between tasks are symbolic, there is no need to access the rest of the DAG to compute the successors.

The DAGuE engine is responsible for moving data from one processor to another when necessary; tasks are enabled only when all incoming data is locally available. Dependencies of the JDF may be marked with a modifier. This modifier is a type qualifier. It tells the communication engine how to transfer data from a memory location to another. As both the in and out dependencies can be typed, complex memory layout can be transferred. This is useful to save 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.

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

The scheduling is fully distributed; all nodes run the scheduling engine. Each process can parse the concise JDF representation independently to find information about any tasks in a memory constrained space. Each computing thread runs for

```
FOR k = 0 .. SIZE-1
  A[k][k], T[k][k] <- DGEQRT(A[k][k])
  FOR m = k+1 .. SIZE-1
    A[k][k], A[m][k], T[m][k] <-
          DTSQRT(A[k][k], A[m][k], T[m][k])
  FOR n = k+1 .. SIZE-1
    A[k][n] <- DORMQR(A[k][k], T[k][k], A[k][n])
    FOR m = k+1 .. SIZE-1
      A[k][n], A[m][n] <-
            DSSMQR(A[m][k], T[m][k], A[k][n], A[m][n])
```

Fig. 2. Pseudocode of the tile QR factorization (right-looking loop nesting order version).

```
1  DGEQRT(k) executes on A(k, k)
2   k = 0..SIZE-1
3
4   V <- (k==0)       ? A(0,0) : C2 DSSMQR(k-1,k,k)
5     -> (k==SIZE-1) ? A(k,k) : R DTSQRT(k,k+1)    [U]
6     -> (k!=SIZE-1) ? V DORMQR(k, k+1..SIZE-1)    [L]
7     -> A(k,k)                                    [L]
8   T -> T DORMQR(k, k+1..SIZE-1)                  [T]
9     -> T(k,k)                                    [T]
```

Fig. 3. Generated JDF representation of the GEQRT dependencies. Arrows indicate the direction of the flow of data, brackets indicate the type of the data.

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. Many advanced strategies for work stealing have been proposed in the past, in this work we implemented one that focuses solely on improving cache locality, as it has been demonstrated to be a major criterion for performant multithreaded execution [20]. Considering all of the many other approaches and selecting the most appropriate is out of the scope of this paper; the performance section will demonstrate that our approach is sufficient to demonstrate the superiority of the contribution of this work, which is the distributed decentralized algebraic DAG scheduling approach.

DAGuE creates one thread per core and binds them to the core. Each thread alternates between executing sequential computational kernels, and executing the generated epilogues and prologues, which are effectively calls to the scheduler. Each thread maintains a local (bounded) list of ready tasks. When a thread completes a task, it executes the epilogue derived from the JDF, determining which tasks may have been enabled. Iterating on the outgoing dependencies of the task, the thread atomically marks which incoming dependencies of the targets are enabled. A task with all IN dependencies enabled can be scheduled. To improve locality and data reuse on NUMA architectures, the scheduler pushes tasks in the local thread-specific queue, if it is not full. If this queue is full, it will push the task to a global shared dequeue that is unbounded. When requesting a task, the scheduler will try to pull a task from the local queue, selecting according to the user-defined priority function for each task. If the local queue is empty, it will steal a task from one of the other threads queues. Bounded queues are organized following the NUMA hierarchy, to favor the cache reuse effect (hence, a thread bound on core c will try to pull from the thread bound on a close core, if its local list is empty, trying each of the other threads one after the other, from closest to farthest). If no work can be found locally, the first task in the global system queue is pulled. As the experiments will show, this approach provides very efficient cache and memory locality. A large local queue size prevents tasks from migrating between threads, while a small local queue would lead to excessive job stealing and thus poor cache and NUMA locality. While this can be subject to further tuning, the queue size of 48 used currently in the engine performs well. The DAGuE environment uses the HWLOC library [21] to discover the NUMA architecture of the machine at runtime and discover architectural proximity. The JDF language and its internal representation at runtime, are specifically optimized to handle DAGs that enable simultaneously a large number of dependencies, called ranges. In Fig. 3, such a range is found on line 6, where the DGEQRT (k) kernel will enable $SIZE - 1 - k - 1$ DORMQR (k, p), $k + 1 \leqslant p \leqslant SIZE - 1$. All the DORMQR tasks use the data V modified by DGEQRT (k), and it is efficient for the cache reuse to schedule these tasks on the same core. Should other cores endure starvation, task stealing evens the workload. The internal dynamic structures are designed for memory efficiency and can support millions of activations with very small overheads, as we demonstrate in Section 5.

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 will require. Again, this evaluation is decentralized, every node evaluates the conditions pertaining only to tasks of interest to determine the nodes hosting successor tasks. The epilogue ends with a call to the asynchronous communication engine to trigger the movement of the output data to the necessary nodes.

3.3. Asynchronous communication engine

In DAGuE, communications are implicitly inferred from the data dependencies between tasks, according to the parallel partitioning. The JDF does not contain the parallel partitioning itself, but the mapping between tasks and data. The upper level has to provide the data distribution, and the programmer the task to data mapping (which can be an arbitrary function as long as it evaluates identically on every rank). Typical mappings (1D, 2D block cyclic) are provided in the framework. The runtime will then map the tasks on nodes according to the (static) parallel partitioning, use job-stealing techniques to favor an efficient scheduling of the local tasks, detect remote dependencies and accordingly issues all the necessary commands to move data between distributed resources without requiring explicit communications to appear in the input language. Although job stealing is a key concept for scheduling within nodes, tasks are immutably pinned to a specific node, according to the evaluation of the data partitioning predicate, in order for the programmer to keep control of the memory distribution load balance and amount of communications of the algorithm. However, the remote scheduling of tasks is still dynamic in the sense that the order in which communication requests are satisfied, hence promoting the exploration of a particular path in the DAG over another, is considered by the communication engine according to tasks priorities and potential network contentions dynamically. The communication engine has to also exhibit a strong level of asynchrony in the progression of network transfers, to effectively achieve communication/computation overlap and asynchronous progress of tasks on

different nodes. As a consequence, in DAGuE, communications are handled by a separate thread, which takes commands from all the compute threads and issues the corresponding network operations. Upon completion of a task, the dependency resolution is executed. Local tasks activations are handled locally, while for remote dependencies an *activate* message is sent to the process that verifies the predicate. From the compute thread's 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 Fig. 6, 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. Leveraging on the ranges expressed in the JDF representation, the communication topology can be adapted to limit the outgoing degree of one-to-all dependencies and establish proper collective communication techniques, such as pipelining or spanning three approaches.

Upon the arrival of an activate message, the destination process schedules the reception of the relevant output data from the parent task by evaluating itself the dependencies of the parent tasks. 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 task, inside the communication thread, with a specific restricted mask to satisfy only the dependencies related to this particular transfer. Remote dependencies resolutions are data specific, not task specific, in order to maximize asynchrony. Tasks enabled during this process are added to the queue of the first compute thread, as there are no cache constraints involved.

In the current version, the communications are performed using MPI. To increase asynchrony, data messages are non-blocking, point-to-point operations allowing tasks to concurrently release remote dependencies, while keeping the maximum number of concurrent messages limited. The collaboration between the MPI processes is realized using *control messages*, short messages containing only the information about completed tasks. The MPI process pre-posts persistent receives to handle the control messages for the maximum number of concurrent tasks completion. Unlike the data messages, 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. Due to the rendezvous protocol described in the previous paragraph, the data payloads are never unexpected, thus reducing memory consumption from the network engine and ensuring flow control.

3.4. GPU and accelerators support

From a memory hierarchy point of view, a GPU behaves as a remote resource, the data has to be explicitly moved into and retrieved from the GPU. However, from an execution point of view, the accelerator is reliant on the CPU, which is in charge of orchestrating the submission of tasks to be executed. Due to architectural choices, some tasks can be executed more efficiently on the GPU, while others are more efficient on the cores. In order to increase the overall utilization of all hardware resources, the workload can be balanced between the cores and the GPUs. The runtime DAGuE scheduler has been extended to handle this heterogeneity, and decides at runtime which tasks to run on which resources, using the appropriate kernel and moving the data back and forth between the accelerator and host memory.

The tasks representation has been extended to allow application developers to specify multiple versions of the computation kernels, implementing a codelet approach. If a task has multiple representations (BODY in the JDF format), the scheduler selects between all the versions the one that matches the hardware it considers for executing the tasks. Minimal interaction from the programmer is required, limited to providing the appropriate GPU equivalents of the regular kernels (oftentimes as simple as replacing a BLAS call by a CuBLAS call).

When the scheduler is invoked via the stubs in the epilogue of a task, it can switch, if an accelerator is idling and if the task at hand can execute on it, to a different scheduling approach and take a GPU in charge. From this point on, it will orchestrate the submission of tasks and the retrieval of results computed by completed tasks on the GPU. This thread maintains this role until all available tasks for the GPU have been submitted. The scheduler, when in GPU mode, multiplexes the different operations asynchronously using multiple streams to enable overlapping of I/O and computation, and to increase the throughput of the GPU. It moves data from the CPU to the GPU, places kernel execution orders, and moves data back from the GPU to the CPU, as well as detects tasks completions and triggers the corresponding actions in the task scheduling system. On hardware supporting concurrent executions (such as NVIDIA Fermi), several tasks will be submitted to the GPU simultaneously, in order to achieve a more effective utilization of the GPU warps/threads. To minimize GPU to host traffic, intermediate results are not retrieved by the host (except until a task that is executed on a CPU needs it) and tasks that depend on already loaded data are preferably executed on that same GPU.

4. Applications and use cases

A distinct feature of our engine is its ability to translate a sequential nested loop code into a concise, symbolic format which it can interpret and then execute in a distributed environment. In this paper, we illustrate the type of algorithms that

can be improved with DAGuE, by presenting the features of the three most useful dense linear algebra algorithms ported to DAGuE, namely: Cholesky, LU and QR factorizations. These three algorithm are among the most useful for production applications, and are often used to compute the numerical solution of linear equations $Ax = b$, to compute eigenvalues, to solve overdetermined systems, or are critical building blocks for sparse solvers like GMRES. These algorithms are specific enough that their parallelization from a sequential code is not practically impossible, but are still challenging enough to demonstrate the wide range of critical applications that can benefit from the DAGuE approach.

4.1. Tile algorithms

We decided to consider the tile version of the linear algebra algorithms. Tile algorithms are based on the idea of processing the matrix by square sub-matrices, referred to as tiles [22], of relatively small size, which makes the operations efficient in terms of cache and TLB use. More importantly in the context of DAGuE, tile algorithms provide more task parallelism than traditional approaches for linear algebra operations, and thus is well suited for DAG scheduling strategies. Tile algorithms have a long history of research in the domain of linear algebra [23], and their use for multicore shared memory architectures led to significant performance improvements [24].

4.2. Matrix factorizations

The QR factorization (or QR decomposition) offers a numerically stable way of solving full rank underdetermined, overdetermined, and regular square linear systems of equations. The QR factorization of an $m \times n$ real matrix A has the form $A = QR$, where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix.

A detailed tile QR algorithm description can be found in [25]. Fig. 2 shows the pseudocode of the Tile QR factorization. It relies on four basic operations implemented by four computational kernels for which reference implementations are freely available as part of either the BLAS, LAPACK or PLASMA [8].

- DGEQRT: The kernel performs the QR factorization of a diagonal tile and produces an upper triangular matrix R and a unit lower triangular matrix V containing the Householder reflectors. The kernel also produces the upper triangular matrix T as defined by the compact WY technique for accumulating Householder reflectors [26]. The R factor overrides the upper triangular portion of the input and the reflectors override the lower triangular portion of the input. The T matrix is stored separately.
- DTSQRT: The kernel performs the QR factorization of a matrix built by coupling the R factor, produced by DGEQRT or a previous call to DTSQRT, with a tile below the diagonal tile. The kernel produces an updated R factor, a square matrix V containing the Householder reflectors and the matrix T resulting from accumulating the reflectors V. The new R factor overrides the old R factor. The block of reflectors overrides the corresponding tile of the input matrix. The T matrix is stored separately.
- DORMQR: The kernel applies the reflectors calculated by DGEQRT to a tile to the right of the diagonal tile, using the reflectors V along with the matrix T.
- DSSMQR: The kernel applies the reflectors calculated by DTSQRT to the tile two tiles to the right of the tiles factorized by DTSQRT, using the reflectors V and the matrix T produced by DTSQRT.

The Cholesky 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. It relies also on four computational kernels: DPOTRF, DSYRK, DTRSM and DGEMM. A detailed description of this algorithm can be found in [24]. The LU factorization with partial row pivoting of an $m \times n$ real matrix A has the form $A = PLU$, where L is an $m \times n$ real unit lower triangular matrix, U is an $n \times n$ real upper triangular matrix and P is a permutation matrix. It relies on four kernels: DGETRF, DTSTRF, DGESSM and DSSSSM. A detailed description of this algorithm can be found in [24,27].

4.3. Expressivity and generality

To avoid the pitfalls associated with general automatic parallelization, the DAGuE approach has to trade some generality to gain in performance. In most programming paradigms, the control flow enforces an ordering on the operations, limiting the possible parallelism, but enabling maximum expressivity at the user level. DAG approaches try to set free from the parallelism limitation, but are usually restricted by the size of the window of visible tasks in the dependency graph, that has to be small to spare the memory consumption. In DAGuE, the graph is not unrolled, instead, algebraic conditions are evaluated to unfold the direct successors and predecessors of a particular task at hand. This is the key to one of the major feature in the DAGuE approach: the ability to execute the kernels in any arbitrary order allowed by the data flow, but without artificial limitations imposed by the control flow. Conversely, this introduces a specific condition on the type of algorithms that can be expressed with DAGuE: any algorithm that depends on the value of the data to drive the construction of the DAG is difficult, nearly impossible to express. Should the list of successor of a task depend on the value of the computed data (or a random value, as is the case for Monte-Carlo simulations), the other processors have no means of computing locally,

without an interaction with the predecessor processor holding that data, that they are part of that successor list, and remote dependencies cannot be satisfied.

While this limitation is strong, we believe that many algorithms can be expressed (even outside the field of dense linear algebra). All algorithms that can be expressed as imperfectly nested loops with affine bounds, increment functions and array subscripts, regardless of their purpose, can benefit from the DAGuE approach. Beyond that, some algorithms might be too complex to be extracted automatically from the sequential code, especially if loop boundaries are not affine, but an advanced programmer can write them directly in the intermediate JDF format. Example of such algorithms are most sorting algorithms, FFT, etc. Moreover, many sparse algorithms feature a graph that is data dependent, but that graph is globally known, as it is directly inferred from the dense blocks structure of the input sparse matrix, and can be evaluated on every node independently with consistent results. As a consequence, we believe that most sparse algorithms can also be expressed with DAGuE; and ongoing work are attempting at porting the sparse GMRES solver.

5. Performance evaluation

5.1. Experimental conditions

Platforms. The Griffon cluster is one of the clusters of the Grid'5000 experimental grid [28]. It is a 648 core machine composed of 81 dual socket Intel Xeon L5420 quad core processors at 2.5 GHz with 16 GB of memory, interconnected by a 20Gbs Infiniband network. Linux 2.6.24 (Debian Sid) is deployed.

The Dancer cluster is a 8 quad core node cluster, based on a Intel Q9400 2.5 Ghz processor, each node with 4 GB of memory. All nodes are connected using a dual Gigabit Ethernet, and Myrinet 10G. Linux 2.6.31.2 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 Math Library MKL-10.1.0.015.

Competing Benchmarks. We compare the performances of the DAGuE based factorizations with three other implementations. ScaLAPACK [29] 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 [18] 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]. HPL is considered the state of the art implementation of an LU factorization. It is used as the prominent metric in the evaluation of the performance level of the most powerful machines in the world issued by the Top 500 [30]. The algorithm and programming paradigm are very similar to the ScaLAPACK version of the LU factorization, but extremely tuned.

Tuning. Parallel factorizations are controlled by several parameters: N defines the size of the input matrix ($N \times N$ doubles), while *NB* defines the size of a tile, or a block, in tiled, or blocked algorithms, respectively. An $N \times N$ matrix is divided in $NT \times NT$ tiles where $NT \times NB = N$. When *NB* does not divide *N*, the incomplete tiles are padded with zeroes. No computation happens on the padding but complete tiles are transferred over the network nonetheless. *P* and *Q*, control the process grid used to map the block cyclic distribution of the matrices. According to [31] and to our experiments, a close to square process grid, with $P \leqslant Q$, minimize the communications while balancing computations. 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. HPL has been tuned experimentally on the Griffon platform (*WR10R2R1* is the best performing combination).

In many figures, we provide the theoretical performance of the platform, computed from the frequency, the depth of the pipeline and the number of cores. The practical peak (*GEMM peak*) is defined as the hypothetical performance of all cores reaching the best performance measured by a single core computing a matrix–matrix multiplication. All benchmarks are using double precision operations, except for GPU benchmarks which are in single precision.

5.2. Scheduling performance

The first results evaluate the overhead of the scheduling engine 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.

Fig. 4 plots the ratio between the time taken by the sequential program with ideal scaling (hence *time*/*p*, 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.

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

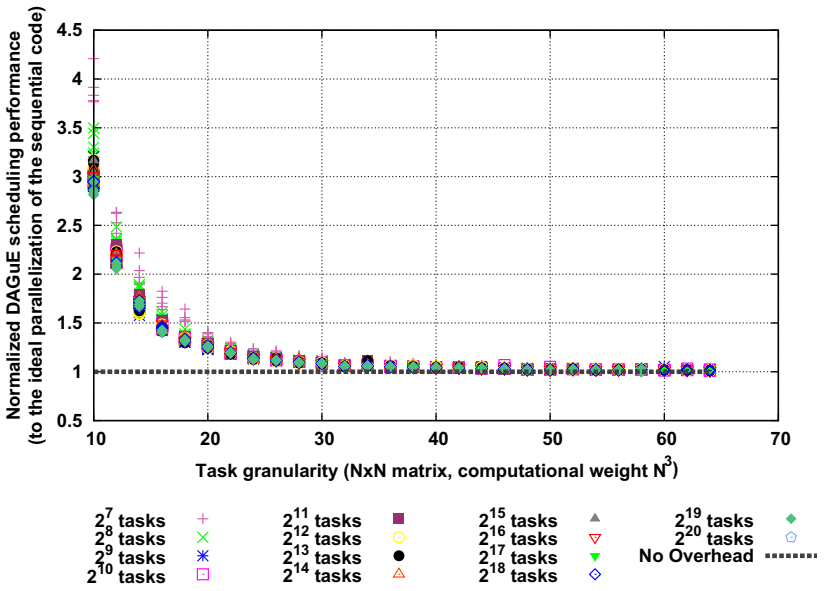


Fig. 4. Ratio between the time taken by the DAGuE engine to schedule *Nb* matrix–matrix multiply of size $N \times N$ and the time taken by the similar sequential code divided by the number of cores (ideal parallelization).

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 computation time. For very small tasks (in the order of microseconds), the overheads due to dynamic scheduling can exceed the ideal execution time by a factor of three, 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 μs on the dancer platform), DAGuE reaches the ideal parallelization performance projection.

5.3. Communication performance

The second benchmark aims at evaluating the communication performance of the DAGuE engine. We have designed a simple ping pong benchmark where a message of variable size is sent from one task to another, a certain number of times. By carefully placing the tasks on different nodes (according to the distribution of data A), we can measure the inherent cost of communications. The JDF representation of this ping pong is presented in Fig. 6. $A(0)$ is the original data it is used in `PING (k)` as a starting point for T if $k = 0$, otherwise T is the output of the previous $(k-1)$ `PONG` task. Similarly, when the expected number of data movements have been reached ($k = NT$) the content of the data is stored back in $A(0)$, otherwise it is given as input for the next `PONG (k+1)` task. Each dataflow, marked with ← or → in the text, is tagged with the expected type `ATYPE`, defined to the desired size by the main program. In the current implementation on top of MPI, the `ATYPE` is an MPI datatype.

We measure the total time t taken to execute this JDF on two machines, interconnected with two 1 Gb/s ethernet, then with Myricom 10 Gb/s, and finally with Infiniband 20 Gb/s. From this time t we compute the average latency of the DAGuE engine. In Fig. 5 we compare these measurements with the NetPIPE [32] benchmark using the same MPI library. For all network types, a high overhead on small message latency is observed for DAGuE: from a factor of 10 on the double-1G Ethernet network to a factor of 90 on the MX-10G network. 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, adding four 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 in the order of magnitude of a matrix tile. Fig. 5 also show that for medium-size messages (64 KB), the difference between NetPIPE and DAGuE is small for the Ethernet

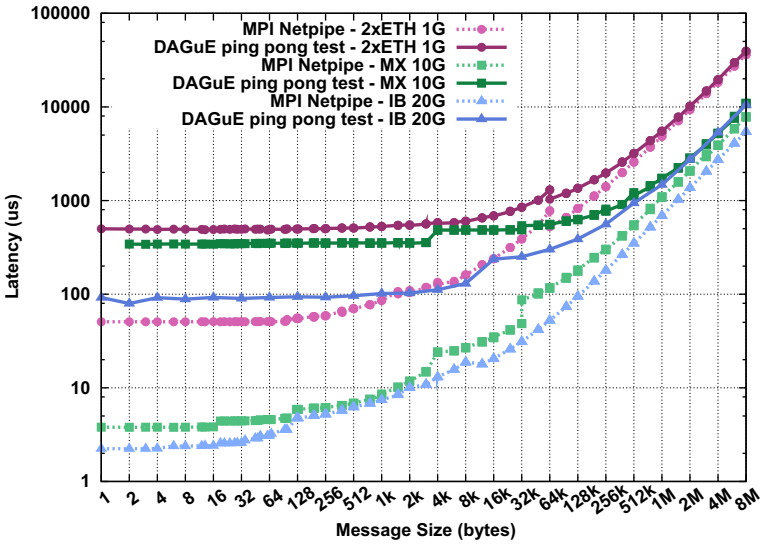


Fig. 5. Round-Trip benchmark – comparison of DAGuE and NetPIPE on Ethernet, Myricom and Infiniband networks.

```
1  PING(k) executes on A(0)
2  k = 0 .. NT      // Execution space
3   T <- (k == 0) ? A(0) : I PONG(k-1) [ATYPE]
4     -> (k == NT) ? A(0) : I PONG(k)  [ATYPE]
5
6  PONG(k) executes on A(1)
7   k = 0 .. NT-1   // Execution space
8   I <- T PING(k)                     [ATYPE]
9      -> T PING(k+1)                  [ATYPE]
```

Fig. 6. JDF representation of the ping pong.

network, and it becomes small at 512 KB for high-speed networks. For the tested applications, the tile size resulting from tuning varies from 200 × 200 (320 KB) to 350 × 350 (≈1 MB), which is in the high efficiency range. Fig. 7 asserts this claim by presenting the performance per core, for a fixed total number of cores, when varying the repartition between distributed memory and shared memory accesses, for the Cholesky factorization test case. Even using the inefficient Ethernet network, the performance per core only decreases slightly when replacing shared memory computation by MPI distributed messaging, outlining the nearly perfect overlap achieved by the communication engine.

5.4. Impact of task granularity

In Fig. 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 tile 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. To compare all runs in a normalized way, the figure represents the efficiency as a percentage of the theoretical peak.

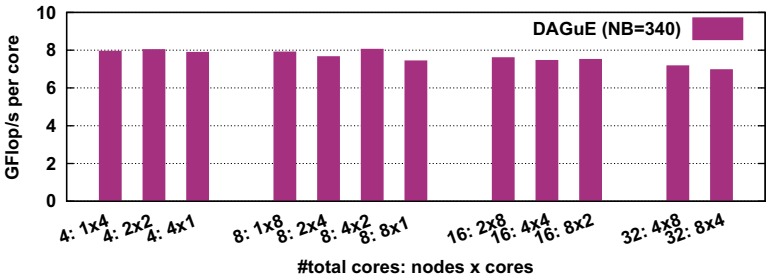


Fig. 7. Performance comparison at fixed total number of cores between distributed and shared memory performance with N = 18200 (Dancer platform, 2xGEthernet).

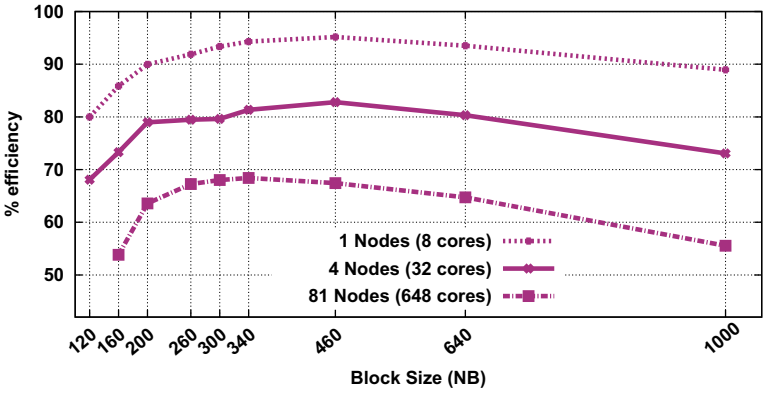


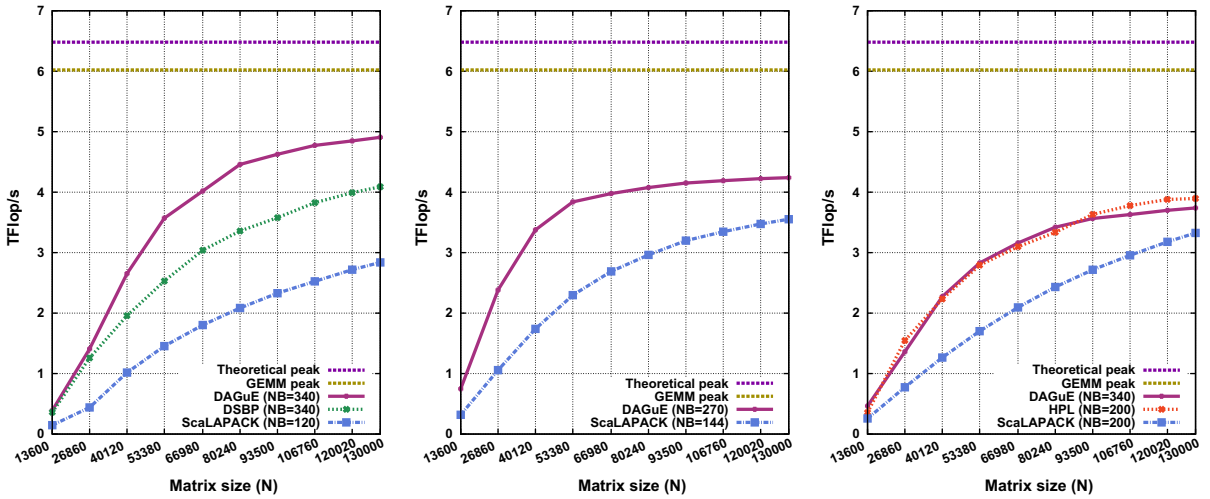
Fig. 8. Performance (relative to the theoretical peak) of the DAGuE Cholesky factorization as function of the tile size (Griffon platform).

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 reuse 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, network efficiency and available parallelism. As seen in Fig. 5, starting at 1 MB, 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 with computation, and as a consequence, block size tuning mostly depends on the BLAS kernels.

5.5. Problem scaling

Fig. 9 presents the problem scaling of the DAGuE factorizations compared to state of the art comparable algorithms. We ran the different 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 size value for each of the reference implementations; tile sizes were tuned as demonstrated in Fig. 8 for the DAGuE implementation; block size, process grid and other parameters are tuned by experimental exploration for HPL, DSBP and ScaLAPACK. 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 tile size, this also means that the number of tiles in the matrix increases with the square of the matrix 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



(a) Cholesky factorization. (b) QR factorization. (c) LU factorization.

Fig. 9. Problem scaling of the Cholesky, QR and LU factorizations, on 648 cores (Griffon platform).

sufficient to reach the plateau with neither of the implementations, except for the DAGuE QR. On the entire range of benchmarks and problem sizes, DAGuE outperforms the non-multi-core aware ScaLAPACK. This advantage is more pronounced for smaller problem sizes, although the lower amount of parallelism due to the smaller number of tiles makes it more difficult to harness the performance of the machine.

Fig. 9(a) compares the performance of the DAGuE Cholesky factorization with DSBP and ScaLAPACK. Though DSBP features a data format specifically tailored for the Cholesky factorization (exploiting the symmetry of the matrix), DAGuE benefits from its dynamic scheduling, and better extraction of parallelism from the loop nest, to better utilize the computing resources and the network. Consequently, DAGuE is able to reach up to 70% of the theoretical peak (75% of GEMM-peak).

When compared with the highly tuned HPL version of the LU algorithm (Fig. 9(c)), DAGuE exhibits an overall similar level of performance. For the largest data-sets, HPL exhibits slightly better performance than DAGuE. Considering that the DAGuE LU is not optimized and is the result of a semi-automatic translation from a sequential code, being able to be competitive against HPL is a significant achievement. One of the major optimizations in HPL is the lookahead, the capacity to compute in advance the next synchronous and expensive panel operation, while the GEMM updates are still ongoing. The DAGuE scheduler mimics this optimization automatically, without the intricacies of the HPL source code.

5.6. Weak scalability

Fig. 10 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. In most benchmarks, Cholesky factorization is preferred, as the DSBP software gives a reference point against another tiled algorithm approach, hence outlining the sole benefit from the superior scheduling of DAGuE.

Clearly, all benchmarks scale almost perfectly, attaining 49% of the GEMM peak for ScaLAPACK, 66% for DSBP, and up to 78% for DAGuE. All runs in the figure are done with a square process grid, 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 $\times$ 8 for the DAGuE engine, and 16 $\times$ 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.

5.7. Strong Scalability

Fig. 11 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 Fig. 11(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 $\times$ 340). For Fig. 11(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 performances (120,020 $\times$ 120,020).

The figure shows that, for a fixed matrix size, the performance of both tiled factorizations (DAGuE and DSBP) scales almost linearly. Because the same matrix is distributed on an increasing number of nodes, the ratio between computations

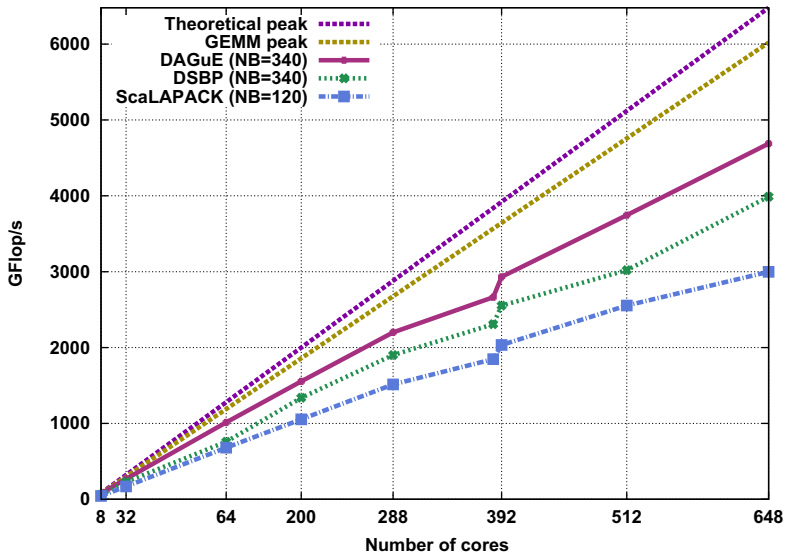
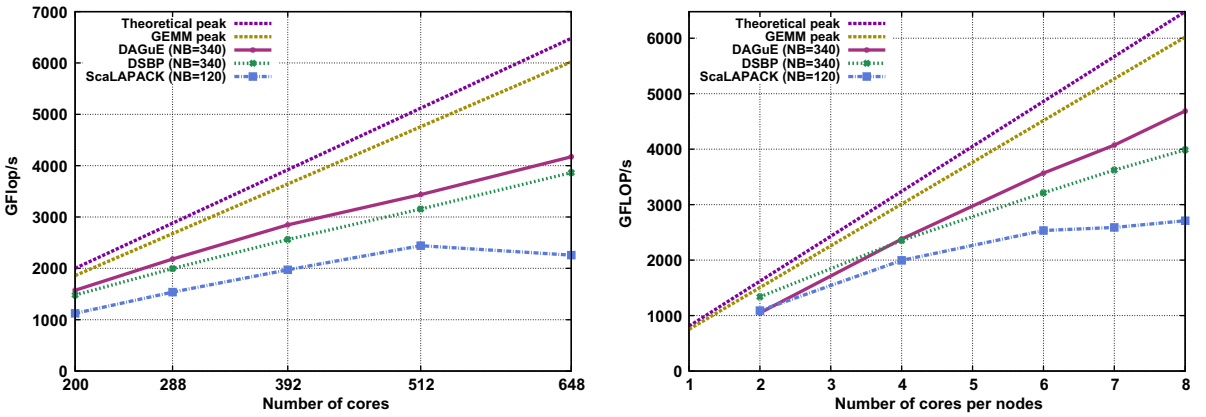


Fig. 10. Weak scalability of the Cholesky factorization, starting from $N = 13{,}600$ for 8 cores (Griffon platform).



(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. 11. Strong scalability of the Cholesky factorization (Griffon platform).

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.

Fig. 11(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 [18], 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 problem specific 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.

5.8. Performance portability

Not only DAGuE can express a large variety of linear algebra algorithms, but, as we show in this section, it also enables porting applications performance from a hardware platform to another without changing the main algorithm description. Fig. 12 presents an execution of the Cholesky algorithm on a heterogeneous cluster encompassing two different types of GPU accelerators. The code of the Cholesky factorization itself is left unchanged; only the GEMM kernels are GPU accelerated.

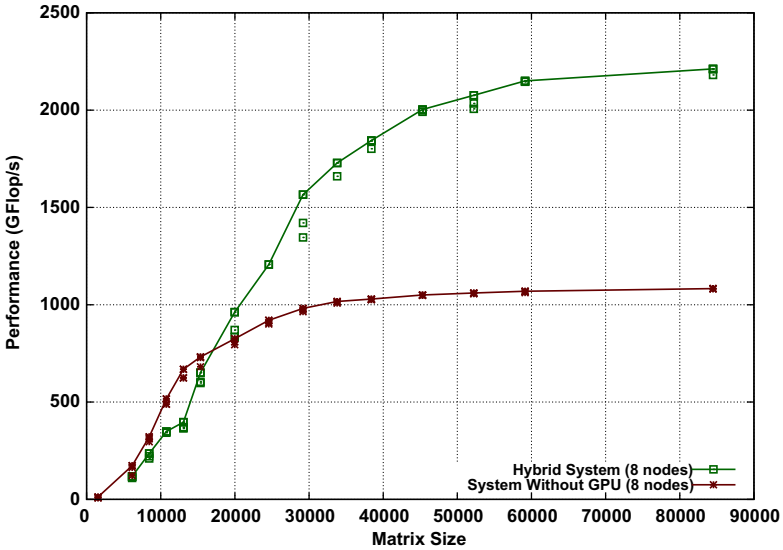


Fig. 12. Problem scaling of the Cholesky factorization on the 8 nodes GPU accelerated dancer platform (4 Tesla C1060 nodes, 4 Fermi C2050 nodes).

The programmer only has to provide the appropriate kernels, and the DAGuE runtime takes care of moving data to the GPU boards, cache coherency, and scheduling on both the CPU cores and the GPU accelerators computing units. As the performance illustrates, DAGuE is able to extract a significant part of the peak performance of this heterogeneous distributed platform. As expected, inter-node load balance is not flexible enough to support efficiently nodes with significantly divergent computing power: the overall execution runs at the pace of a Tesla C1060 homogeneous system. DAGuE does not target heterogeneous ad hoc desktop grids, but supercomputers, whose architecture is envisioned [33] to be an homogeneous cluster of nodes with internal heterogeneity. Indeed, intra-node heterogeneity is very well handled, as the total performance represents 72% of an homogeneous system with 8 Tesla nodes; for comparisons, the efficiency of the CPU only system is 83%, the efficiency of the HPL algorithm on Tianhe-1A, the most powerful GPU accelerated machine in the TOP500, is only 55%.

6. Conclusion

With the emergence of massively many-core parallel architectures, the classical approach based on pure MPI programming model is becoming 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 slower rates than the increase of operations per second performed by the CPU. Specifically, 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 striking 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 specifically targeting clusters of multi-cores, with a hybrid programming model mixing implicit 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, emphasizing a very good efficiency starting from task granularity of a few microseconds. The dense linear algebra one-sided factorizations, QR, LU and Cholesky, have been implemented, using an automated tool to extract the dataflow representation from a sequential algorithm, and demonstrate the performance of the system on a realistic workload. The performance of these algorithms have been compared to the classical approach for distributed systems programming, represented by the ScaLAPACK routines and the very optimizes HPL, 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, *DAGuE* 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 most finely chiseled one-sided factorizations available today.

References

[1] A.J. Bernstein, Analysis of programs for parallel processing, IEEE Transactions on Electronic Computers EC-15 (5) (1966) 757–763.
[2] E.G. Coffman, Jr., P.J. Denning, Operating Systems Theory, Prentice Hall Professional Technical Reference, 1973.
[3] J.A. Sharp (Ed.), Data Flow Computing: Theory and Practice, Ablex Publishing Corp, 1992.
[4] J. Yu, R. Buyya, A taxonomy of workflow management systems for grid computing, Tech. rep., Journal of Grid Computing, 2005.
[5] O. Delannoy, N. Emad, S. Petiton, Workflow global computing with YML, in: 7th IEEE/ACM International Conference on Grid Computing, 2006.
[6] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, S. Tomov, The impact of multicore on math software, in: Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA, Lecture Notes in Computer Science, vol. 4699, Springer, 2006, pp. 1–10.
[7] E. Chan, F.G. Van Zee, P. Bientinesi, E.S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn, Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks, in: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ACM, 2008, pp. 123–132.
[8] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, Journal of Physics: Conference Series 180.
[9] R. Dolbeau, S. Bihan, F. Bodin, HMPP: A hybrid multi-core parallel programming environment, in: Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007), 2007.
[10] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198.
[11] J. Perez, R. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: IEEE International Conference on Cluster Computing, 2008, pp. 142–151.
[12] F. Song, A. YarKhan, 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, ACM, New York, NY, USA, 2009, pp. 1–11.
[13] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, in: Euro-Par 2009 Euro-par'09 Proceedings, LNCS, Delft Pays-Bas, 2009.
[14] M. Cosnard, E. Jeannot, Automatic parallelization techniques based on compact DAG extraction and symbolic scheduling, Parallel Processing Letters 11 (2001) 151–168.
[15] M. Cosnard, E. Jeannot, T. Yang, Compact DAG representation and its symbolic scheduling, Journal of Parallel and Distributed Computing 64 (8) (2004) 921–935.

[16] E. Jeannot, Automatic multithreaded parallel program generation for message passing multiprocessors using parameterized task graphs, in: International Conference 'Parallel Computing 2001' (ParCo2001), 2001.
[17] P. Husbands, K.A. Yelick, Multi-threading and one-sided communication in parallel lu factorization, in: B. Verastegui (Ed.), Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10–16, 2007, Reno, Nevada, USA, ACM Press, 2007.
[18] F.G. Gustavson, L. Karlsson, B. Kågström, Distributed SBP cholesky factorization algorithms with near-optimal scheduling, ACM Transactions on Mathematical Software 36 (2) (2009) 1–25.
[19] 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.
[20] U.A. Acar, G.E. Blelloch, R.D. Blumofe, The data locality of work stealing., in: SPAA'00, 2000, pp. 1–12.
[21] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications, in: IEEE (Ed.), PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Pisa Italy, 2010.
[22] F.G. Gustavson, J.A. Gunnels, J.C. Sexton, Minimal data copy for dense linear algebra factorization, Applied Parallel Computing, State of the Art in Scientific Computing, 8th International Workshop, PARA 2006, vol. 4699, LNCS, Umeå, Sweden, 2006, pp. 540–549.
[23] G.W. Stewart, Matrix algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.
[24] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Computation 35 (1) (2009) 38–53.
[25] A. Buttari, J. Langou, J. Kurzak, J.J. Dongarra, Parallel tiled QR factorization for multicore architectures, Concurrency Computation: Practice and Experience 20 (13) (2008) 1573–1590.
[26] R. Schreiber, C. van Loan, A storage-efficient WY representation for products of householder transformations, J. Sci. Stat. Comput. 10 (1991) 53–57.
[27] E.S. Quintana-Ortí, R.A. van de Geijn, Updating an LU factorization with pivoting, ACM Transactions on Mathematical Software 35 (2) (2008) 11.
[28] 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, I. Touche, Grid'5000: a large scale and highly reconfigurable experimental grid testbed, IJHPCA 20 (4) (2006) 481–494.
[29] 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, 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.
[30] J.J. Dongarra, P. Luszczek, A. Petitet, The LINPACK benchmark: past, present and future, Concurrency and Computation: Practice and Experience 15 (9) (2003) 803–820.
[31] J. Choi, J. Demmel, I.S. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D.W. Walker, R.C. Whaley, ScaLAPACK: a portable linear algebra library for distributed memory computers – design issues and performance, in: J. Dongarra, K. Madsen, J. Wasniewski (Eds.), Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, Second International Workshop, PARA '95, Lyngby, Denmark, August 21–24, 1995, Proceedings, Lecture Notes in Computer Science, vol. 1041, Springer, 1995, pp. 95–106.
[32] Q.O. Snell, A.R. Mikler, J.L. Gustafson, Netpipe: A network protocol independent performance evaluator, in: IASTED International Conference on Intelligent Information Management and Systems, 1996.
[33] J. Dongarra, P. Beckman, et al., The international exascale software project roadmap, Tech. rep., IESP, 2011, http://www.exascale.org/iesp.