2011 IEEE International Parallel & Distributed Processing Symposium

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.

To achieve this, an MPI/threads 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. Instead of allowing the 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 portability point of view these issues are hard to adress in a generic way, and are yet orthogonal to the algorithm design computational scientists are interested on.

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. 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. 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. 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 via the data distribution. 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 being used to express parallelism and task dependencies in distributed systems, with an emphasis on grid and peer-to-peer systems [2], [3]. Recently, several projects [4], [5], [6], [7], [8], mostly in the field of Linear Algebra, have proposed to 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: [3] reads a concise representation of the DAG (in XML), and unrolls it in memory before scheduling it. [9], [6], [10] modifies the

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.281

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 [11], 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 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 [12], [13], 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. [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. 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.

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 input format used by DAGuE is called JDF. It expresses the tasks and their data dependencies. The JDF representation of a DAG is then pre-compiled as C-code by the framework 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, in the order of their definition in the JDF. This function represents the interface used by the caller.

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. As the JDF does not contain information about the data distribution, only the mapping of tasks on the data, the upper level has to provide the data distribution. The runtime will then map the tasks according to the parallel partitioning described in the JDF, find an efficient scheduling of the tasks, detect remote dependencies and automatically move data between distributed resources. Below, we present in detail the mechanisms involved in the scheduler and the communication

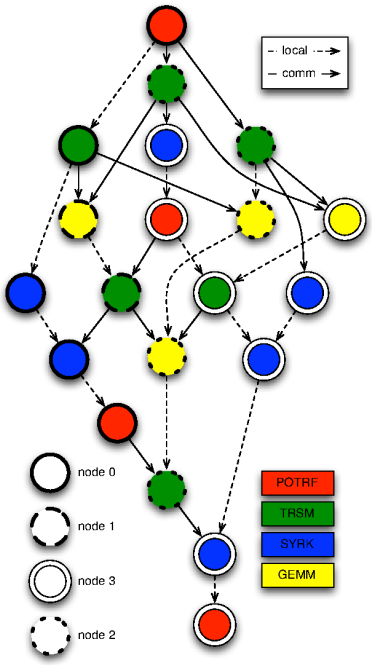


Fig. 1. DAG of Cholesky for a 4x4 tile matrix on a 2x2 grid of processors.

engine, that enable the maximum amount of parallelism with dynamic asynchronous distributed scheduling.

A. Input Format

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 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 1 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, etc...), and its memory requirements would 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 the 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

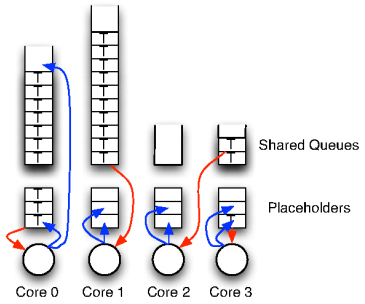


Fig. 2. Illustration of the multi-core scheduling

$O(1)$ 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

The scheduling is fully distributed; all nodes run the scheduling engine. 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.

DAGuE creates one thread per core and binds them to the core. Each thread runs its own version of the scheduler. Figure 2 represents critical structures used by the 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, 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 becomes schedulable. To improve locality and data reuse on NUMA architectures, the scheduler favors the queuing of the newly enabled tasks in the placeholder of the current thread. As these tasks will always execute on the same thread, this approach maximizes cache and memory locality. A large placeholder prevents tasks from migrating between threads, while a zero size placeholder would lead to excessive job stealing and thus poor cache and NUMA locality. While this can be subject to some tuning, the current version of the engine uses a placeholder with size one. If the placeholder is full, the tasks are put at the beginning or at the end of the thread waiting-queue, based on their priorities as specified in the JDF. When a thread looks for a task to execute, the first task in the placeholder is chosen; if no task is found, the thread tries to pop from the beginning of its own queue. If this fails, the thread tries to pop from the end of other threads' waiting-queues, ordered by architectural proximity. The DAGuE environment uses the HWLOC library [16] to discover the NUMA architecture of the machine at runtime and discover architectural proximity.

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. The epilogue ends with a call to the Asynchronous Communication Engine to trigger the movement of the output data to the requesting nodes.

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. 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.

C. Asynchronous Communication Engine

In DAGuE, communications are implicitly inferred from the data dependencies between tasks, according to 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 compute threads and issue 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 figure 4, 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 arrival 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

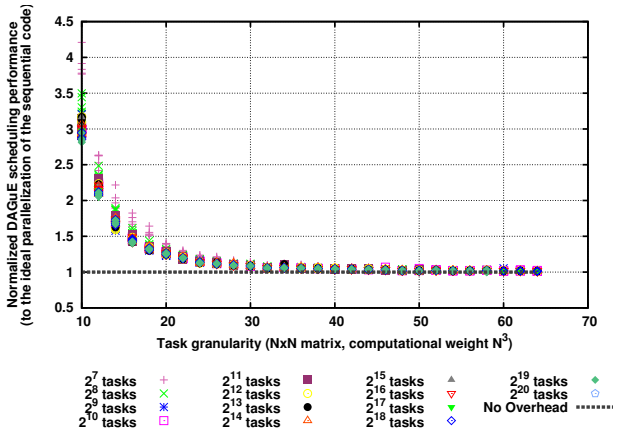


Fig. 3. Ratio between the time taken by the DAGuE engine to schedule Nb matrix-matrix multiply of size $N \times N$ and the time taken by the similar sequential code divided by the number of cores (ideal parallelization)

the 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.

IV. Performance Evaluation

A. Experimental conditions

The Griffon cluster is one of the clusters of the Grid'5000 experimental grid [17]. 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 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, 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.

B. Micro benchmarking

1) 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.

Figure 3 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.

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 $44\mu s$ on the dancer platform), DAGuE reaches the ideal parallelization performance projection.

```
1  PING(k)
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)
7   k = 0 .. NT-1  // Execution space
8   I <- T PING(k)                     [ATYPE]
9      -> T PING(k+1)                  [ATYPE]
10
```

Fig. 4. JDF representation of the ping pong

2) 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 node to another, a certain number of times. The JDF representation of this ping pong is presented in Figure 4. 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 Gbs ethernet, then with Myricom 10Gbs, and finally with Infiniband 20 Gb/s. From this time t we compute 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 5 we compare these measurements with the NetPIPE [18] benchmark using the same MPI library.

Figure 5 demonstrates a high overhead on latency for DAGuE, independent of the networks: 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. Figure 5 also show that for medium-size messages (64KB), the difference between NetPIPE and DAGuE is small 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 efficiency range.

C. Application Benchmarking

Cholesky Factorization: The Cholesky factorization (or Cholesky decomposition) is mainly used for the numerical solution of linear equations $Ax = b$.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 version of the Cholesky factorization. As described in [19], a single step of the algorithm is implemented by a sequence of calls to the LAPACK and BLAS routines: SYRK, POTRF, GEMM, TRSM. The tile Cholesky algorithm is identical to the block Cholesky algorithm implemented in LAPACK, except for

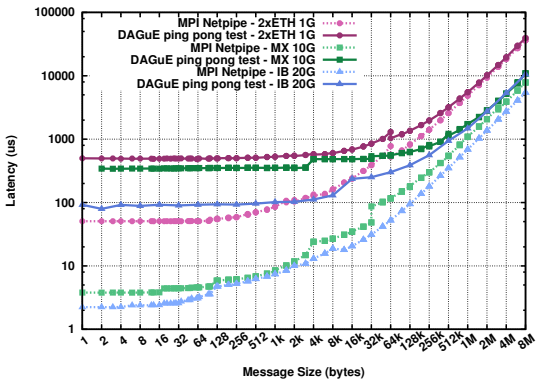


Fig. 5. Round-Trip benchmark – comparison of DAGuE and NetPIPE on Ethernet, Myricom and Infiniband networks.

```
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. 6. Pseudocode of the tile Cholesky factorization (right-looking version).

processing the matrix by tiles. Figure 6 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 a tile, or a block, in tiled, or blocked algorithms, respectively. A $N \times N$ matrix is divided in $NT \times NT$ tiles 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 on the computing resources. According to [20] and to our experiments, the best performance is achieved when using a process grid that is square or closest to square with $P \leq Q$, as it balances the communications and computations across the nodes. 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 performance 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* (matrix-matrix product) performance of the platform. GEMM peak is measured as the best performance obtained by a single core computing a 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

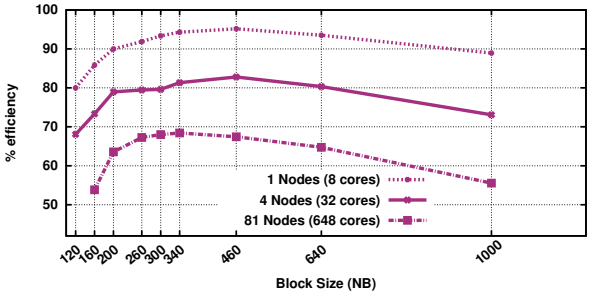


Fig. 7. Performance (relative to the theoretical peak) of the DAGuE Cholesky Factorization as function of the Tile Size (Griffon platform).

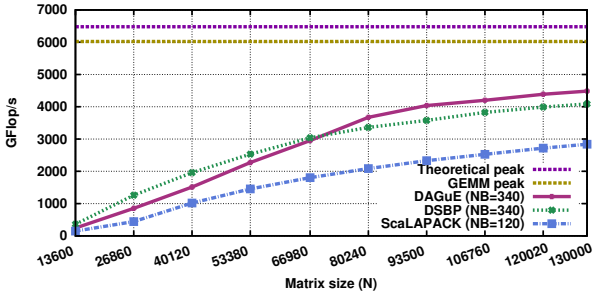


Fig. 8. Problem Scaling of the Cholesky Factorization, on 81 nodes (Griffon platform).

as the practical peak performance of the platform. 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 [20] is the reference implementation for distributed parallel machines. 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 Figure 7, 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.

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 Figure 5, 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 with computation, and as a consequence, block size tuning mostly depends on the BLAS kernels.

2) Problem Scaling: Figure 8 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) varying the problem size (from $13,600 \times 13,600$ to $130,000 \times 130,000$). We took the best block value for each implementation; block sizes were tuned as demonstrated in Figure 7 for DAGuE.

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 either implementation.

Figure 8 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 only decreases slightly when replacing shared memory computation by MPI distributed messaging, outlining the nearly perfect overlap achieved by the communication engine.

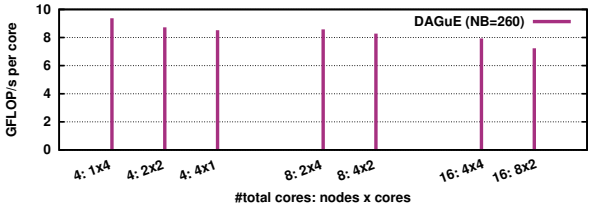


Fig. 9. Performance comparison at fixed total number of cores between distributed and shared memory performance with N=18200 (Dancer platform, 2xGEthernet).

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

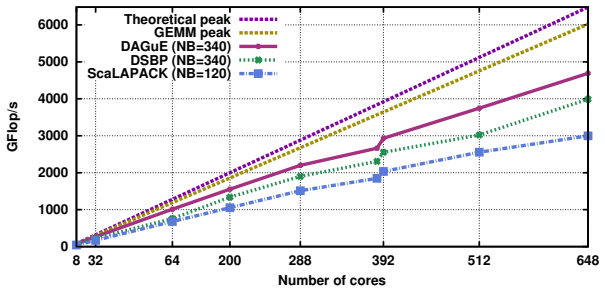


Fig. 10. Weak Scalability of the Cholesky Factorization, starting from N=13,600 for 8 cores (Griffon platform).

4) Weak Scalability: Figure 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.

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 6×8 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 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 Figure 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 (340×340). For Figure 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 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.

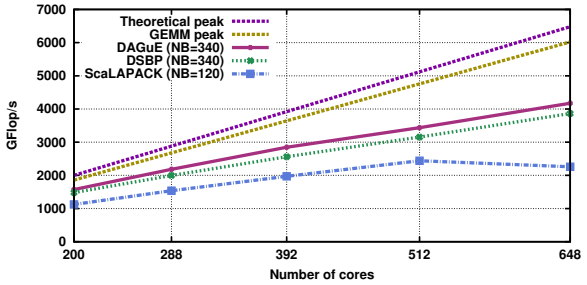
Figure 11(b) illustrates that the DAGuE and DSBP approaches are best fitted for clusters with many cores. We were able to run on a larger matrix because even at 2 cores per node, the whole memory of the 81 nodes is available. As shown in [15], DSBP data representation enables it to outperform ScaLAPACK. 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 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.

6) Generality of DAGuE: Because the existence of DSBP gives a comparison point against a similar tiled factorization algorithm, but using a static scheduling, in this paper we mostly focused our results on the Cholesky factorization. However, we have also used the DAGuE framework to implement two other well known Dense Linear Algebra factorization algorithms: the tiled version of QR [21] and the tiled version of LU [19]. Moreover, a working prototype for the Sparse Linear Algebra GMRES kernel is underway. However, due to lack of space there results are not presented in this paper.

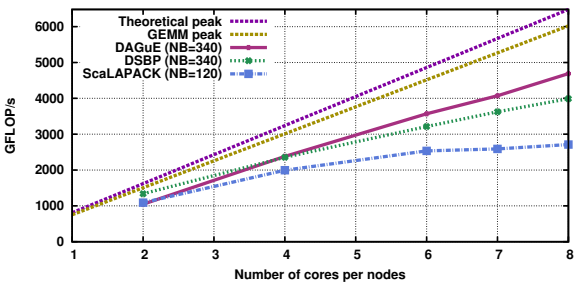
V. Conclusion

With the emergence of massively multicore architectures, the classical approach based on MPI SPMD 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 different 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 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



(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)

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.

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, vol. 4699. Springer, 2006, pp. 1–10.

[5] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and 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.

[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.

[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] 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/

[17] 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.

[18] 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.

[19] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, 2009.

[20] 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.

[21] 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.