2011 IEEE International Parallel & Distributed Processing Symposium

Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA

George Bosilca*, Aurelien Bouteiller*, Anthony Danalis*†, Mathieu Faverge*, Azzam Haidar*,
Thomas Herault*‡, Jakub Kurzak*, Julien Langou§¶, Pierre Lemarinier*, Hatem Ltaief*,
Piotr Luszczek*, Asim YarKhan* and Jack Dongarra*†
*University of Tennessee Innovative Computing Laboratory
†Oak Ridge National Laboratory
‡University Paris-XI
§University of Colorado Denver
¶Research was supported by the National Science Foundation grant no. NSF CCF-811520

***Abstract*—We present a method for developing dense linear algebra algorithms that seamlessly scales to thousands of cores. It can be done with our project called DPLASMA (Distributed PLASMA) that uses a novel generic distributed Direct Acyclic Graph Engine (DAGuE). The engine has been designed for high performance computing and thus it enables scaling of tile algorithms, originating in PLASMA, on large distributed memory systems. The underlying DAGuE framework has many appealing features when considering distributed-memory platforms with heterogeneous multicore nodes: DAG representation that is independent of the problem-size, automatic extraction of the communication from the dependencies, overlapping of communication and computation, task prioritization, and architecture-aware scheduling and management of tasks. The originality of this engine lies in its capacity to translate a sequential code with nested-loops into a concise and synthetic format which can then be interpreted and executed in a distributed environment. We present three common dense linear algebra algorithms from PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures), namely: Cholesky, LU, and QR factorizations, to investigate their data driven expression and execution in a distributed system. We demonstrate through experimental results on the Cray XT5 Kraken system that our DAG-based approach has the potential to achieve sizable fraction of peak performance which is characteristic of the state-of-the-art distributed numerical software on current and emerging architectures.**

***Keywords*-Numerical linear systems, scalable parallel algorithms, scheduling and task partitioning**

I. Introduction and Motivation

Among the various factors that drive the momentous changes occurring in the design of microprocessors and high end systems, three stand out as especially notable: 1) the number of transistors per chip will continue the current trend, i.e. double roughly every 18 months, while the speed of processor clocks will cease to increase; 2) we are getting closer to the physical limit for the number and bandwidth of pins on the CPUs and 3) there will be a strong drift toward hybrid/heterogeneous systems for petascale (and larger) systems. While the first two involve fundamental physical limitations that the state-of-art research today is unlikely to prevail over in the near term, the third is an obvious consequence of the first two, combined with the economic necessity of using many thousands of CPUs to scale up to petascale and larger systems.

More transistors and slower clocks means multicore designs and more parallelism required. The fundamental laws of traditional processor design – increasing transistor density, speeding up clock rate, lowering voltage – have now been blocked by a set of physical barriers: excess heat produced, too much power consumed, too much energy leaked, useful signal overcome by noise. Multicore designs are a natural response to this situation. By putting multiple processor cores on a single die, architects can overcome the previous limitations, and continue to increase the number of gates per chip without increasing the power densities. However, despite obvious similarities, multicore processors are not equivalent to multiple-CPUs or to SMPs. Multiple cores on the same chip can share various caches (including TLB – Translation Look-aside Buffer) and they compete for memory bandwidth. Extracting performance from such configurations of resources means that programmers must exploit increased thread-level parallelism (TLP) and efficient mechanisms for inter-processor communication and synchronization to manage resources effectively. The complexity of parallel processing will no longer be hidden in hardware by a combination of increased instruction level parallelism (ILP) and pipeline techniques, as it was with superscalar designs. It will have to be addressed at an upper level, in software, either directly in the context of the applications or in the programming environment. As portability remains a requirement, clearly the programming environment has to drastically change.

A thicker memory wall means that communication efficiency will be even more essential. The pins that connect the processor to main memory have become a strangle point, with both the rate of pin growth and the bandwidth per pin slowing down, if not flattening out. Thus the processor to memory performance gap, which is already approaching

1530-2075 2011

DOI 10.1109/IPDPS.2011.299

a thousand cycles, is expected to grow, by 50% per year according to some estimates. At the same time, the number of cores on a single chip is expected to continue to double every 18 months, and since limitations on space will keep the cache resources from growing as quickly, cache per core ratio will continue to go down. Problems with memory bandwidth and latency, and cache fragmentation will, therefore, tend to become more severe, and that means that communication costs will present an especially notable problem. To quantify the growing cost of communication, we can note that time per flop, network bandwidth (between parallel processors), and network latency are all improving, but at significantly different rates: 59%/year, 26%/year and 15%/year, respectively. Therefore, it is expected to see a shift in algorithms' properties, from computation-bound, i.e. running close to peak today, toward communication-bound in the near future. The same holds for communication between levels of the memory hierarchy: memory bandwidth is improving 23%/year, and memory latency only 5.5%/year. Many familiar and widely used algorithms and libraries will become obsolete, especially dense linear algebra algorithms which try to fully exploit all these architecture parameters. They will need to be reengineered and rewritten in order to fully exploit the power of the new architectures.

In this context, the PLASMA project [1] has developed several new algorithms for dense linear algebra on shared memory system based on tile algorithms (see section II). In this paper, we present DPLASMA, a follow up project related to PLASMA, that operates in the distributed-memory environment. DPLASMA introduces a novel approach to schedule dynamically dense linear algebra algorithms on distributed systems. It, too, is based on tile algorithms, and takes advantage of DAGuE [2], a new generic distributed Direct Acyclic Graph Engine for high performance computing. This engine supports a DAG representation independent of problem-size, overlaps communications with computation, prioritizes tasks, schedules in an architecture-aware manner and manages micro-tasks on distributed architectures featuring heterogeneous many-core nodes. The originality of this engine resides in its capability of translating a sequential nested-loop code into a concise and synthetic format which it can interpret and then execute in a distributed environment. We consider three common dense linear algebra algorithms, namely: Cholesky, LU and QR factorizations, to investigate through the DAGuE framework their data driven expression and execution in a distributed system. We demonstrate through performance results at scale that our DAG-based approach has the potential to bridge the gap between the peak and the achieved performance that is characteristic in the state-of-the-art distributed numerical software on current and emerging architectures. However, the most essential contribution, in our view, is the ease with which new algorithmic variants may be developed and how they can be simply launched on a massively parallel architecture without much consideration to the underlying hardware structure. It is due to the flexibility of the underlying DAG scheduling engine and straightforward expression of parallel data distributions.

The remainder of the paper is organized as follows. Section II describes the related work. Section III details the background information on translating a domain-specific algorithm into to a generic DAG of tasks. Section IV presents the DAGuE framework. Finally, Section V gives the experimental results and Section VI provides the conclusion and future work.

II. Related Work

This paper reflects the convergence of algorithmic and implementation advancements in the area of dense linear algebra in the recent years. This section presents the solutions that laid the foundation for this work, which include: the development of the class of *tile algorithms*, the application of performance-oriented matrix layout and the use of dynamic scheduling mechanisms based on representing the computation as a *Directed Acyclic Graph* (DAG) [3].

A. Tile Algorithms

The tile algorithms are based on the idea of processing the matrix by square submatrices, referred to as tiles, of relatively small size. This makes the operation efficient in terms of cache and TLB use. The Cholesky factorization lends itself readily to tile formulation, however the same is not true for the LU and QR factorizations. The tile algorithms for them are constructed by factorizing the diagonal tile first and then incrementally updating the factorization using the entries below the diagonal tile. This is a very well known concept, that dates back to the work by Gauss, and is clearly explained in the classic book by Golub and Van Loan [4] and Stewart [5]. These algorithms were subsequently rediscovered as very efficient methods for implementing linear algebra operations on multicore processors [6], [7], [8], [9], [10].

It is crucial to note that the technique of processing the matrix by square tiles yields satisfactory performance only when accompanied by data organization based on square tiles. This fact was initially observed by Gustavson [11], [12] and recently investigated in depth by Gustavson, Gunnels and Sexton [13]. The layout is referred to as *square block* format by Gustavson et al. and as *tile layout* in this work. The paper by Elmroth, Gustavson, Jonsson and Kågström [14] provides a systematic treatment of the subject.

Finally, the well established computational model that uses DAGs as its representation together with the dynamic task scheduling have gradually made their way into academic dense linear algebra packages. The model is currently used in shared memory codes, such as PLASMA (University of Tennessee, University of California Berkeley, University of Colorado at Denver) [1] and FLAME (Formal Linear

Algebra Methods Environment; University of Texas Austin) [15].

B. Parameterized Task Graphs

One challenge in scaling to large scale many-core systems is how to represent extremely large DAGs of tasks in a compact fashion, incorporating the dependency analysis and structure within the compact representation. Cosnard and Loi have proposed the *Parameterized Task Graph* [16] as a way to automatically generate and represent the task graphs implicitly in an annotated sequential program. The data flow within the sequential program is automatically analyzed to produce a set of tasks and communication rules. The resulting compact DAG representation is conceptually similar to the representation described in this paper. Using the parameterized task graph representation various static and dynamic scheduling techniques were explored by Cosnard et al. [17], [18].

C. Task BLAS for distributed linear algebra algorithms

The *Task-based BLAS (TBLAS)* project [19], [20] is an alternative approach to task scheduling for linear algebra algorithms in a distributed memory environment. The TBLAS layer provides a distributed and scalable tile based substrate for projects like ScaLAPACK [21]. Linear algebra routines are written in a way that uses calls to the TBLAS layer, and a dynamic runtime environment handles the execution in an environment consisting of a set of distributed memory, multi-core computational nodes.

The ScaLAPACK style linear algebra routines make a sequence of calls to the TBLAS layer. The TBLAS layer restructure the calls as a sequence of tile-based tasks, which are then submitted to the dynamic runtime environment. The runtime accepts additional task parameters (data items are marked as input, output or input and output) upon insertion of tasks into the system and this information is later used to infer the dependences between various tasks. The tasks can then be viewed as comprising a DAG with the data dependences forming the edges. The runtime system uses its knowledge of the data layout (e.g., block cyclic) in order to determine where the data items are stored in a distributed memory environment and decide which tasks will be executed on the local node and which tasks will be executed remotely. The portion of the DAG relevant to the local tasks are retained at each node. Any task whose dependences are satisfied can be executed by the cores on the local node. As tasks execute, additional dependences become satisfied and the computation can progress. Data items that are required by a remote task are forwarded to that remote node by the runtime.

This approach to task scheduling scales relatively well, and has performance that is often comparable to that of ScaLAPACK. However, there is an inherent bottleneck in the DAG generation technique. Each node must execute the entire ScaLAPACK level computation and generate all the tasks in the DAG, even though only the portions of the DAG relevant to that node are retained. This is one of the most fundamental design differences between TBLAS and DAGuE.

```
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])
```

Figure 1. Pseudocode of the tile Cholesky factorization (right-looking version).

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

Figure 2. Pseudocode of the tile QR factorization.

III. Background on Dependence Analysis

We will apply the DAGuE framework to three of the most fundamental one-sided factorizations of numerical linear algebra: Cholesky, LU, and QR factorizations. Figure 1 shows the pseudocode of the Cholesky factorization (the right-looking variant). Figure 2 shows the pseudocode of the tile QR factorization. Figure 3 shows the pseudocode of the tile LU factorization. Each of the figures shows the tile formulation of the respective algorithm: a single tile of the

```
FOR k = 0..TILES-1
  A[k][k], T[k][k] ← DGETRF(A[k][k])
  FOR m = k+1..TILES-1
    A[k][k], A[m][k], T[m][k] ← DTSTRF(A[k][k], A[m][k], T[m][k])
  FOR n = k+1..TILES-1
    A[k][n] ← DGESSM(A[k][k], T[k][k], A[k][n])
    FOR m = k+1..TILES-1
      A[k][n], A[m][n] ← DSSSSM(A[m][k], T[m][k], A[k][n], A[m][n])
```

Figure 3. Pseudocode of the tile LU factorization.

matrix is denoted by double-index notation A[i][j].

The DAGuE framework is generic by design and requires from a specific algorithm to be represented as a DAG of dependences. This may be readily achieved for the three linear algebra factorizations by recasting the linear algebra meaning of the computational kernels into dependence scheduling nomenclature [22] commonly used in the compiler community. To start with a simple example, in Figure 1, the first (and only) invocation of the DPOTRF computational kernel has a form:

```
A[k][k] <- DPOTRF(A[k][k])
```

From the compiler stand point, this operation reads from A[k][k] (input dependence) and writes to A[k][k] (output dependence). To simplify the dependence analysis we could rewrite the operation as:

```
A[k][k] <- A[k][k] + 1
```

The loss of semantics (the new form is not equivalent to the original) may easily be compensated by preserving a reference to the original code. It is trivial for most of mainstream compiler frameworks to analyze the modified form of the statement: it is both input and output dependence – INOUT for short (following the notation borrowed from Fortran 90's function parameter annotation). It is also possible to have input-only dependences:

```
A[m][k] <- DTRSM(A[k][k], A[m][k])
```

A[k][k] carries input dependence and the whole statement may be rewritten in simpler (but dependence preserving) form:

```
A[m][k] <- A[k][k] + A[m][k]
```

For output-only dependences:

```
A[k][k], T[k][k] <- DGEQRT(A[k][k])
```

T[k][k] carry output dependence. And the equivalent form could be:

```
T[k][k] <- A[k][k] + 1
A[k][k] <- A[k][k] + 1
```

Finally, it is also possible to have SCRATCH designation for temporary storage that doesn't carry any dependence but is necessary for proper functioning of the algorithm (this is again borrowed from Fortran 2008's SCRATCH designation). The SCRATCH parameters allow for dynamic allocation of memory of size not known before runtime (i.e. at compile time). In addition, the allocated memory is automatically deallocated upon exiting the lexical scope where such allocation occurred.

By rewriting the original statement we can simplify the original code and have it accessible for loop-carried dependence analysis. An alternative approach is to use the dependence designation introduced above (IN, OUT, INOUT, and SCRATCH) inserted into the original code and have the rewriting and dependence analysis done automatically. This is in fact the approach taken by the DAGuE framework as it separates the semantics of the domain specific code from its DAG representation required for efficient scheduling. For example, the DPOTRF function is designated to accept a single argument (a matrix tile) that carries input and output dependence:

```
DPOTRF(A : INOUT)
```

And this is the only change required from the end user in the implementation of DPOTRF() which otherwise should just be a standard sequential function: an LAPACK subroutine in this case.

IV. The DAGuE Framework

This section introduces the DAGuE framework [2], a new runtime environment which schedules tasks dynamically in a distributed environment. The tile QR factorization is used as a test case to explain how the overall execution is performed in parallel.

A. Description

The originality of this framework for distributed environment resides in the fact that its starting point is a sequential nested-loop user-application, similar to the pseudocode from Fig. 1-3. The framework then translates it in DAGuE's internal representation called JDF (Job Description Format), which is a concise parameterized representation of the sequential program's DAG. This intermediate representation is eventually used as input to trigger the parallel execution by the DAGuE engine. It includes the input and output dependences for each task, decorated with additional information about the behavior of the task.

For an $NT \times NT$ tile matrix, there are $\mathcal{O}(NT^3)$ tasks. The memory requirement to store the full DAG quickly increases with NT. In order to have a scalable approach however, DAGuE uses symbolic interpretation to schedule tasks without unrolling the JDF in memory at any given time, and thus spares computation cycles to walk the DAG, and memory to keep a global representation. So, basically this synthetic representation allows the internal dependence management mechanism to efficiently compute the flow of data between tasks without having to unroll the whole DAG, and to discover on the fly the communications required to satisfy these dependences. Indeed, the knowledge of the IN and OUT dependences, accessible anywhere in the DAG execution, is sufficient to implement a fully distributed scheduling engine for the underlying DAG. At the same time, the concept of looking variants (i.e., right-looking, left-looking, top-looking) in the context of LAPACK and ScaLAPACK becomes irrelevant with this representation: instead of hard-coding a particular variant of tasks ordering, the execution is now data-driven and dynamically scheduled.

The issue of which "looking" variant to choose is avoided because the execution of a task is scheduled when the data is available. On the other hand, it is still possible to insist on a particular traversal order of the DAG which would yield a particular "looking" variant. This kind of extension to DAGuE is supported mostly for educational purpose.

Such representation is expected to be internal to the DAGuE framework though, and not a programming language at user disposal. The framework, as described here, does not automate the computation of the data and task distribution. The user is thus required to manually add such information in the JDF. The process of such automation is beyond the scope of this writing as we are trying to compare against the established practices of distributed linear algebra software which assumes fixed data distribution.

From a technical point of view, the main goal of the distributed scheduling engine is to select a local task for which all the IN dependences are satisfied, i.e. the data is available locally, select one of the local cores where to run the task and execute the body of the task when it is scheduled. Once executed, the scheduling engine releases all the OUT dependences of this task, thus making more tasks available to be scheduled, locally or remotely. It is noteworthy to mention that the scheduling mechanism is architecture aware, taking into account not only the physical layout of the cores, but also the way different cache levels and memory nodes are shared between the cores. This allows to determine the best local core, i.e. the one that minimizes the number of cache misses and data movements over the memory bus.

The DAGuE engine is obviously responsible of moving data from one node to another when necessary. These data movements are necessary to release dependences of remote tasks. The framework language introduces a type qualifier called *modifier*, expressed as MPI datatypes in the current version. It tells the communication engine what is the shape of the data to be transferred from a remote location to another. By default, the communication engine uses a default data type for the tiles (the user defines it to fit the tile size of the program). But the framework has also the capability to transfer data of any shape. Indeed, sometimes, only a particular area of the default data type must be conveyed. Again, at this stage, the user has still to manually specify how the transfers must be done using these modifiers. Moreover, the data tracking engine is capable to understand if the different modifiers overlap, and behaves appropriately when tracking the data dependences. One should note that the DAGuE engine allows modifier settings on both input and output dependences, so that one can change the shape of the data on the fly during the communication.

B. A Test Case: QR Factorization

A realistic example of the DAGuE's internal representation for the QR factorization is given in Fig. 4. As stated in the previous section, this example has been obtained starting from the sequential pseudocode shown in Fig. 2 using the DAGuE's translation tools. The logic to determine the task distribution scheme has been hard-coded and could be eventually provided by auto-tuning techniques. The tile QR consists of four kernel operations: DGEQRT, DSSMQR, DORMQR, and DTSQRT. For each operation, we define a function (lines 1 to 13 for DGEQRT) that consists of 1) a definition space (DGEQRT is parametrized by k, the step of the factorization, that takes values between 0 and $NT-1$); 2) how task distribution maps the data distribution (DGEQRT(k) runs on the process that holds the tile A(k, k)); 3) a set of data flows (lines 7 to 13 for DGEQRT(k)); and 4) a body that holds the effective C-code that will eventually be executed by the scheduling engine (the body has been excluded from the picture. It is a simple C code to call the DGEQRT routine of LAPACK on the variables V and T which are instanciated to the corresponding memory locations of the process by the DAGuE framework before the execution of the body).

Dependencies apply on data that are necessary for the execution of the task, or that are produced by the task. For example, the task DGEQRT uses one data V as input, and produces two data, a modified version of the input V, and T a data locally produced by the task. Input data, such as V, are indicated using the left arrow. They can come either from input matrix (local to the task, or located on a remote process), or from the output data of another task (executed either locally, or remotely). For example, the V of DGEQRT(k) comes either from the original matrix located in tile A(0, 0) if k==0, or from the output data C2 of task DSSMQR(k-1, k, k) otherwise. Output dependences, marked with a right arrow, work in the same manner. In particular, DGEQRT produces V which can be sent to DTSQRT and DORMQR depending on the values of k. These dependences are marked with a modifier (line 8 and 9) at their end: [U] and [L] for DTSQRT and DORMQR, respectively. This tells the DAGuE engine that the functions DTSQRT and DORMQR only require the strict lower part of V and only the upper part of V as inputs, respectively. The whole tile could have been transferred instead, but this would engender two main drawbacks: (1) communicating more data than required and (2) add extra dependences into the DAG which will eventually serialize the DORMQR and DTSQRT calls. This works in the same manner for output dependences. For example, in line 10, only the lower part of V is written and stored on the memory in the lower part of the tile pointed by A(k, k). Also, a data that is sent to memory is final, meaning that no other task will modify its contents until the end of the DAG execution. However, this does not prevent other tasks from using it as a read-only input.

Fig. 5 depicts the complete unrolled DAG of a 4x4 tiles QR, as resulting from the execution of the previously described DAG on a 2x2 processor grid. The color represents

```
1  DGEQRT(k)  (high_priority)
2   // Execution space
3   k = 0..NT-1
4   // Data Distribution
5   : A(k, k)
6   // Data flows
7   V <- (k==0) ? A(0,0) : C2 DSSMQR(k-1,k,k)
8     -> (k==NT-1) ? A(k,k) : R DTSQRT(k,k+1)  [U]
9     -> (k!=NT-1) ? V1 DORMQR(k, k+1..NT-1)   [L]
10    -> A(k,k)                                [L]
11  T -> T DORMQR(k, k+1..NT-1)                [T]
12    -> T(k,k)                                [T]
13
14 DTSQRT(k,m)  (high_priority)
15  // Execution space
16  k = 0..NT-2
17  m = k+1..NT-1
18  // Data Distribution
19  : A(m, k)
20  // Data flows
21  V2 <- (k==0) ? A(m,0) : C2 DSSMQR(k-1,k,m)
22     -> V2 DSSMQR(k,k+1..NT-1,m)
23     -> A(m,k)
24  R  <- (m==k+1) ? V DGEQRT(k) :
25                 R DTSQRT(k,m-1)      [U]
26     -> (m==NT-1) ? A(k, k) :
27                  R DTSQRT(k,m+1)     [U]
28  T  -> T DSSMQR(k,k+1..NT-1,m)       [T]
29     -> T(m,k)                        [T]
```

```
36 DORMQR(k,n)  (high_priority)
37  // Execution space
38  k = 0..NT-2
39  n = k+1..NT-1
40  // Data Distribution
41  : A(k, n)
42  // Data flows
43  T  <- T DGEQRT(k)                          [T]
44  V1 <- V DGEQRT(k)                          [L]
45  C1 <- (k==0) ? A(k,n) : C2 DSSMQR(k-1,n,k)
46     -> C1 DSSMQR(k,n,k+1)
47
48 DSSMQR(k,n,m)
49  // Execution space
50  k = 0    .. NT-2
51  n = k+1  .. NT-1
52  m = k+1  .. NT-1
53  // Data Distribution
54  : A(m, n)
55  // Data flows
56  V2 <- V2 DTSQRT(k,m)
57  T  <- T DTSQRT(k,m)                                [T]
58  C2 <- (k==0) ? A(m,n) : C2 DSSMQR(k-1,n,m)
59     -> (n==k+1 & m==k+1) ? V DGEQRT(k+1)
60     -> (n==k+1 & k<m-1) ? V2 DTSQRT(k+1,m)
61     -> (k<n-1 & m==k+1) ? C1 DORMQR(k+1,n)
62     -> (k<n-1 & k<m-1) ? C2 DSSMQR(k+1,n,m)
63  C1 <- (m==k+1) ? C1 DORMQR(k,n) :
64                 C1 DSSMQR(k,n,m-1)
65     -> (m==NT-1) ? A (k,n) : C1 DSSMQR(k,n,m+1)
```

Figure 4. Concise representation of tile QR factorization

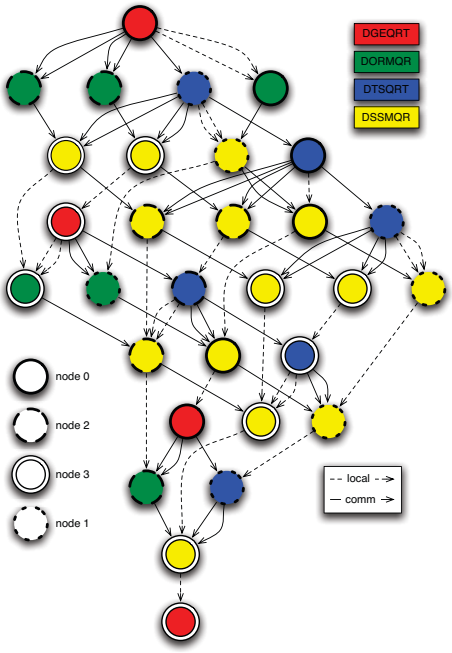

Figure 5. DAG of QR for a 4x4 tile matrix.

the task to be executed (DGEQRT, DORMQR, DTSQRT and DSSMQR), while the border of the circles represents the node where the tasks has been executed. The edges between the tasks represents the data flowing from one tasks to another. A solid edge indicate that the data is coming from a remote resource, while a dashed edge indicate a local output of another task.

C. DPLASMA and the DAGuE framework

DPLASMA is an extension of the PLASMA idea using the DAGuE framework. It implements a subset of PLASMA's tile algorithms for some of the linear algebra operations of LAPACK inside the DAGuE system. It provides an implementation of these algorithms for a distributed-memory system with multicore nodes. Four operations have been implemented in DPLASMA today: the Cholesky, QR and LU factorizations, as well as the distributed matrix matrix multiply (GEMM). Although DPLASMA is implemented on top of DAGuE, it is useable in any MPI scientific program. Thus, in this context, DPLASMA provides a replacement for ScaLAPACK, as PLASMA replaces LAPACK for shared-memory multicore systems.

V. Performance Results

In this section we show the cost of using DPLASMA as opposed to an optimized vendor library. This is to ascertain the feasibility of our approach for performance-conscious codes. In particular, the penalty associated with decoupling the optimized sequential kernels from the parallelization and scheduling components. We focus on the three common algorithms we have described extensively earlier: the tile Cholesky, tile QR, and tile LU.

A. Hardware Description

The Kraken system is a Cray XT5 with 8256 compute nodes interconnected with SeaStar that features a 3D torus topology. Each compute node has two six-core AMD

Opterons (clocked at 2.6 GHz) for a total of 99072 cores. All nodes have 16 Gbytes of memory: 4/3 Gbytes of memory per core. Cray Linux Environment (CLE) 2.2 is the OS on each node. The Kraken system is located at the National Institute for Computational Sciences (NICS) at Oak Ridge National Laboratory.

B. Software Description

For comparative studies, we used Cray LibSci – the best set of parallel routines available on the system from the hardware vendor. From the numerical analysis stand point, LibSci offers functional equivalents of the aforementioned DPLASMA routines. Cray LibSci shares input data layout and API with ScaLAPACK. However, to increase the achievable fraction of peak performance, the library was tuned by tightly integrating the computational portion and the communication layer of the library. Additional optimizations incorporated into Cray LibSci take advantage of the Cray XT5 interconnect characteristics and the peculiarities of the CLE, and that includes the job scheduler's strict affinity policies, kernel aggregation of hardware interrupt handlers, and a very long process scheduler's tick (it is almost an order of magnitude longer than the setting in most Linux distributions). Neither DPLASMA nor DAGuE take advantage of this kind of optimizations in the version of the code presented here.

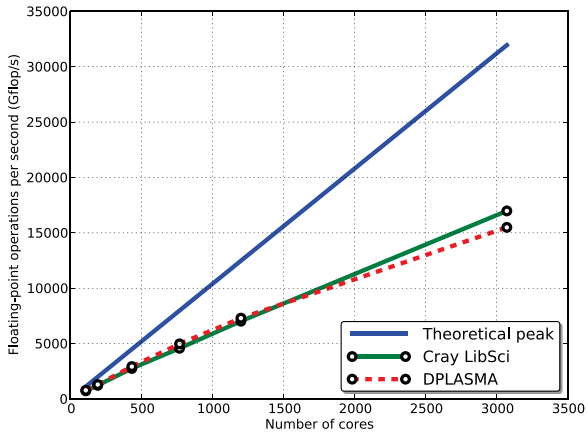

Figure 6. Performance comparison of Cholesky factorization codes on Cray XT5 in weak scaling scenario.

C. Performance Tuning

Maximizing the performance and minimizing the execution time of scientific applications is a daily challenge for the HPC community. The tile QR and LU factorizations depend strongly on tunable execution parameters, namely the outer and the inner blocking sizes (NB and IB), which trade utilization of different system resources. The tile Cholesky depends only on the outer blocking size.

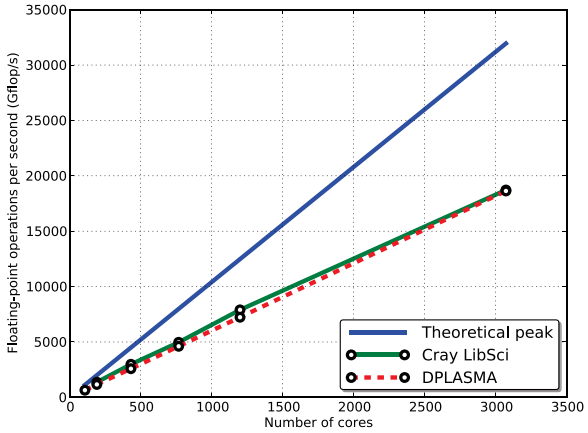

Figure 7. Performance comparison of LU factorization codes on Cray XT5 in weak scaling scenario.

The outer block size (NB) trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size (IB) trades off memory load with extra-flops due to redundant calculations. Hand-tuning by active probing has been performed to determine the optimal NB and IB for each factorization. $NB = 1800$ has been selected for all three factorizations and $IB = 225$ for LU and QR factorizations.

Moreover, in a parallel distributed framework, the efficient parallelization of the tile QR and LU factorization algorithms greatly relies on the data distribution.

There are several indicators of a "good" data distribution and it is actually a challenge to optimize all of these cost functions at once. A good distribution has to unlock tasks on remote nodes as quickly as possible (concurrency); it has to enable a good load balance of the algorithm; and it definitely has to minimize communication. ScaLAPACK popularized *elementwise* 2D block cyclic data distribution as its data layout. The distribution currently used in DPLASMA is *tilewise* 2D block cyclic. As we have raised the level of abstraction from scalars to tiles when going from LAPACK to PLASMA, we found it useful to raise the level for the data distribution from scalars to tiles when going from ScaLAPACK to DPLASMA. In ScaLAPACK, each process contains an rSIZE×cSIZE block of scalars and this pattern is repeated in a 2D block cyclic fashion. In DPLASMA, each process possesses an rtileSIZE x ctileSIZE block of tiles (of size NB x NB). This block of tiles enable multiple cores within a nodes to work concurrently on the various tiles of the block (as opposed to the elementwise distribution) while enabling good load balancing, low communication and great concurrency among nodes (similarly to elementwise distribution). We found it best for the *tilewise* 2D block cyclic distribution to be strongly rectangular for QR and

LU (with more tile rows than tile columns) and more square for Cholesky. These facts on tilewise distribution for DPLASMA matches previous results obtained for element-wise distribution for ScaLAPACK [23].

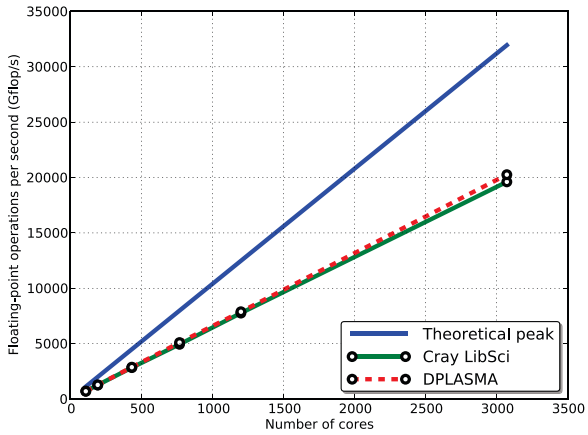

Figure 8. Performance comparison of QR factorization codes on Cray XT5 in weak scaling scenario.

D. Comparative Study

We present our results in Figure 6, Figure 7, and Figure 8. The results indicate that the performance obtained by DAGuE and DPLASMA is nearly identical to Cray's LibSci in a weak scaling context and remains at a constant fraction of the peak performance at the scale of multiple thousands of cores.

Even though DPLASMA already matches the performance of Cray's LibSci there are a number of improvements we consider.

First, for QR and LU, we have tested DPLASMA with square tile 2D block cyclic distribution. The configuration is depicted in Figure 5. Starting from the top we see that DGEQRT is executed on node 0, followed by DTSQRT on node 2, followed by another DTSQRT on node 0, and another DTSRQT on node 2. We are doing three inter-node communications. A better algorithm would be to have DGEQRT on node 0, followed by DTSQRT on node 0, followed by DTSQRT on node 2, followed by DTSQRT on node 2. The total number of inter-node communication is now reduced to 1 (instead of 3). Such an execution pattern can be achieved by adjusting the data distribution or by changing the DAG traversal order of the algorithm. In the global sense, the number of messages sent is $\mathcal{O}((N/NB)*(N/NB))$ and the volume of messages sent is $\mathcal{O}((N/NB)*(N/NB)*NB)$, both of which are asymptotically larger than the lower bound afforded by ScaLAPACK's 2D block-cyclic distribution. We consider this to be the cost that comes with the flexibility of the automatically scheduled DAG.

Second, we need to improve the broadcast operations in DAGuE. Currently the broadcast is implemented with the root sending the data to each of the recipients. This is suitable for some operations but certainly not for Cholesky, LU or QR. A better way to do the broadcast in this context is with a ring broadcast [24]. We believe that DAGuE needs to be able to support broadcast topology provided by the user and thus better adapt to a given algorithm. The BLACS (Basic Linear Algebra Communication Subroutines) [25] has this capacity. Also DAGuE is not yet able to group messages, this would be useful for example in the tile LU or QR factorizations where two arrays need to be broadcast at once in the same broadcast configuration. As an illustration, in Figure 5, the three pairs of arrows going from the first DGEQRT to the three DORMQR below, represent the broadcast of the same data (T and V) from the same root to the same nodes. These two arrays can obviously been concatenated to work around the network latency.

VI. Summary and Future Work

This paper introduced a method of developing distributed implementation of linear algebra kernels using DPLASMA. The three particular examples used throughout were QR, LU, and Cholesky factorizations. DPLASMA is based on tile algorithms and DAGs of tasks that are scheduled dynamically. It is implemented on top of the DAGuE engine, which is capable of extracting tasks and their data dependencies from the sequential user applications based on nested loops. DPLASMA expresses such algorithms with an intermediate concise and synthetic format (JDF). The engine then schedules the generated tasks across a distributed system without the need to unroll the entire DAG. Compared to established software provided by vendor, DPLASMA's linear algebra routines are equivalent in functionality, accuracy, and performance. Based on these results we conclude that DAGuE is a promising framework that provides a scalable and dynamic scheduling for parallel distributed machines. All the existing distributed-memory codes rely on static scheduling of the operations. This is inadequate in cases when multiple scheduling variants are required to achieve performance on vastly different hardware platforms. Parallel distributed dynamic schedulers offer a viable solution. However, the current dynamic parallel distributed schedulers are intrinsically limited in scalability. However, we have shown that our dynamic scheduler scales over 3000 cores and achieves, or even surpasses, performance levels available only from highly tuned parallel libraries that were developed by the hardware vendor. And unlike any existing distributed-memory implementation, our approach is much more flexible from the user perspective as it requires only a sequentiential loop nest as input. Multiple variants of the same factorization may be easily provided by the user and our system is able to generate a parallel distributed implementation automatically and will schedule the execution

accordingly thus creating a rapid development capability without sacrificing performance.

REFERENCES

[1] University of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Archtectures, Version 2.2*, November 2009.

[2] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. In *Proceedings of the 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, Anchorage, AL, USA, May, 20 2011. to appear.

[3] John A. Sharp, editor. *Data flow computing: theory and practice*. Ablex Publishing Corp, 1992.

[4] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996. http://www.amazon.com/exec/obidos/ASIN/0801854148/ISBN: 0801854148.

[5] G. W. Stewart. *Matrix algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[6] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008. http://dx.doi.org/10.1002/cpe.1301DOI: 10.1002/cpe.1301.

[7] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parellel Comput. Syst. Appl.*, 35:38–53, 2009. http://dx.doi.org/10.1016/j.parco.2008.10.002DOI: 10.1016/j.parco.2008.10.002.

[8] E. S. Quintana-Ortí and R. A. van de Geijn. Updating an LU factorization with pivoting. *ACM Trans. Math. Softw.*, 35(2):11, 2008. http://doi.acm.org/10.1145/1377612.1377615DOI: 10.1145/1377612.1377615.

[9] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equation on the CELL processor using Cholesky factorization. *Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008. http://dx.doi.org/10.1109/TPDS.2007.70813DOI: TPDS.2007.70813.

[10] J. Kurzak and J. J. Dongarra. QR factorization for the CELL processor. *Scientific Programming*, 17(1-2):31–42, 2009. http://dx.doi.org/10.3233/SPR-2009-0268DOI: 10.3233/SPR-2009-0268.

[11] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. & Dev.*, 41(6):737–756, 1997. http://dx.doi.org/10.1147/rd.416.0737DOI: 10.1147/rd.416.0737.

[12] F. G. Gustavson. New generalized matrix data structures lead to a variety of high-performance algorithms. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 211–234, Ottawa, Canada, October 2-4 2000. Kluwer Academic Publishers. http://www.amazon.com/exec/obidos/ASIN/0792373391ISBN: 0792373391.

[13] F. G. Gustavson, J. A. Gunnels, and J. C. Sexton. Minimal data copy for dense linear algebra factorization. In *Applied Parallel Computing, State of the Art in Scientific Computing, 8th International Workshop, PARA 2006*, Umeå, Sweden, June 18-21 2006. Lecture Notes in Computer Science 4699:540-549. http://dx.doi.org/10.1007/978-3-540-75755-9_66DOI: 10.1007/978-3-540-75755-9_66.

[14] E. Elmroth, F. G. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004. http://dx.doi.org/10.1137/S0036144503428693DOI: 10.1137/S0036144503428693.

[15] The FLAME project. http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage, April 2010.

[16] M. Cosnard and M. Loi. Automatic task graph generation techniques. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*, page 113, Washington, DC, USA, 1995. IEEE Computer Society.

[17] Michel Cosnard and Emmanuel Jeannot. Compact dag representation and its dynamic scheduling. *J. Parallel Distrib. Comput.*, 58(3):487–514, 1999.

[18] Michel Cosnard, Emmanuel Jeannot, and Tao Yang. Compact dag representation and its symbolic scheduling. *J. Parallel Distrib. Comput.*, 64(8):921–935, 2004.

[19] Fengguang Song, Asim YarKhan, and Jack 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*, pages 1–11, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1654059.1654079DOI: 10.1145/1654059.1654079.

[20] Fengguang Song. *Static and dynamic scheduling for effective use of multicore systems*. PhD thesis, University of Tennessee, 2009.

[21] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997. http://www.netlib.org/scalapack/slug/.

[22] Utpal Banerjee. *Dependence Analysis (Loop Transformation for Restructuring Compilers)*. Springer, 1996.

[23] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11-12):1723–1743, November-December 2003.

[24] Fred G. Gustavson, Lars Karlsson, and Bo Kågström. Distributed SBP Cholesky factorization algorithms with near-optimal scheduling. *ACM Trans. Math. Softw.*, 36(2):1–25, 2009.

[25] J. Dongarra, R. van de Geijn, and C. Whaley. *A Users' Guide to the BLACS*. University of Tennessee, Knoxville, TN, 1993. available in postscript from netlib/scalapack.