Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: 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 DPLASMA, a new project related to PLASMA, that operates in the distributed memory regime. It uses a new generic distributed Direct Acyclic Graph engine for high performance computing (DAGuE). Our work also takes advantage of some of the features of DAGuE, such as DAG representation that is independent of problem-size, overlapping of communication and computation, task prioritization, architecture-aware scheduling and management of micro-tasks on distributed architectures that feature heterogeneous many-core nodes. The originality of this engine is that it is capable of translating a sequential nested-loop code into a concise and synthetic format which it can be interpret and then execute in a distributed environment. We consider three common dense linear algebra algorithms, namely: Cholesky, LU and QR factorizations, to investigate their data driven expression and execution in a distributed system. We demonstrate from our preliminary results 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 softwares on current and emerging architectures.**

***Index Terms*—Linear systems, 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 on the 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 modus operandi of traditional processor design, increase the transistor density, speed up the clock rate, raise the voltage has now been blocked by a stubborn set of physical barriers: excess heat produced, too much power consumed, too much voltage leaked. 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 on the 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!) and they certainly share the memory bus. 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.

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 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 exponentially 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[1] project 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 new 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 is based on these tile algorithms, using 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 from preliminary our results 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.

The remainder of the paper is organized as follows. Section II describes the related work, Section III recalls the three one-sided factorization based on tile algorithms. 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 Denver Colorado) [1] and FLAME (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 and collaborators [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. Curing this problem is one of our motivation for creating the DAGuE framework.

III. Background

All the kernels mentioned below have freely available reference implementations as part of either the BLAS [22], [23], LAPACK [24] or PLASMA [1] sequential kernels. Optimized implementations are available on a given machine for the BLAS and LAPACK. Note: PLASMA sequential kernel names do not follow previous papers. They follow current PLASMA code terminology.

A. Cholesky Factorization

The Cholesky factorization (or Cholesky decomposition) is mainly used for the numerical solution of linear equations $Ax = b$, where A is symmetric and positive definite. Such systems arise often in physics applications, where A is positive definite due to the nature of the modeled physical phenomenon. This happens frequently in numerical solutions of partial differential equations.

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. In LAPACK the double precision algorithm is implemented by the DPOTRF routine. A single step of the algorithm is implemented by a sequence of calls to the LAPACK and BLAS routines: DSYRK, DPOTF2, DGEMM, DTRSM. Due to the symmetry, the matrix can be factorized either as upper triangular matrix or as lower triangular matrix.

The tile Cholesky algorithm is identical to the block Cholesky algorithm implemented in LAPACK, except for processing the matrix by tiles. Otherwise, the exact same operations are applied. The algorithm relies on four basic operations implemented by four computational kernels:

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

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

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

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

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

Figure 1 shows the pseudocode of the Cholesky factorization (the right-looking variant).

B. QR Factorization

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. The traditional algorithm for QR factorization applies a series of elementary Householder matrices of the general form

$$H = I - \tau v v^T,$$

where v is a column reflector and τ is a scaling factor. In the block form of the algorithm a product of nb elementary Householder matrices is represented in the form

$$H_1 H_2 \ldots H_{nb} = I - VTV^T,$$

where V is an $N \times nb$ real matrix those columns are the individual vectors v, and T is an $nb \times nb$ real upper triangular matrix [25], [26]. In LAPACK the double precision algorithm is implemented by the DGEQRF routine.

Here a derivative of the block algorithm is used called the *tile QR* factorization. The ideas behind the tile QR factorization are well known. The tile QR factorization was initially developed to produce a high-performance "out-of-memory" implementation (typically referred to as "out-of-core") [27] and, more recently, to produce high performance implementation on "standard" (x86 and alike) multicore processors [6], [7], [28] and on the CELL processor [10]. Further more, Demmel et al. [29] proved that the tile QR factorization was *communication optimal* in the sequential case and the parallel case (using a binary tree).

The algorithm is based on the idea of annihilating matrix elements by square tiles instead of rectangular panels (block columns). The algorithm produces "essentially" the same R factor as the classic algorithm, e.g., the implementation in the LAPACK library. (Elements may differ in sign.) However, a different set of Householder reflectors is produced and a different procedure is required to build the Q matrix. The tile QR algorithm relies on four basic operations implemented by four computational kernels:

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 [25], [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 two tiles to the right of the tiles factorized by DTSQRT, using the reflectors V and the matrix T produced by DTSQRT.

Figure 2 shows the pseudocode of the tile QR factorization.

C. LU Factorization

The LU factorization (or LU decomposition) 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

```
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])
```

Fig. 2. Pseudocode of the tile QR factorization.

```
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])
```

Fig. 3. Pseudocode of the tile LU factorization.

a permutation matrix. In the block formulation of the algorithm, factorization of *nb* columns (the panel) is followed by the update of the remaining part of the matrix (the trailing submatrix) [30], [31]. In LAPACK the double precision algorithm is implemented by the DGETRF routine. A single step of the algorithm is implemented by a sequence of calls to the following LAPACK and BLAS routines: DGETF2, DLASWP, DTRSM, DGEMM, where DGETF2 implements the panel factorization and the other routines implement the update.

Here a derivative of the block algorithm is used called the *tile LU* factorization. Similarly to the tile QR algorithm, the tile LU factorization originated as an "out-of-memory" ("out-of-core") algorithm [8] and was recently rediscovered for the multicore architectures [7], [28].

Again, the main idea here is the one of annihilating matrix elements by square tiles instead of rectangular panels. The algorithm produces different U and L factors than the block algorithm (e.g., the one implemented in the LAPACK library). In particular we note that the L matrix is not lower unit triangular anymore. Another difference is that the algorithm does not use partial pivoting but a different pivoting strategy. The tile LU algorithm relies on four basic operations implemented by four computational kernels:

DGETRF: The kernel performs the LU factorization of a diagonal tile and produces an upper triangular matrix U, a unit lower triangular matrix L and a vector of pivot indexes P. The U and L factors override the input and the pivot vector is stored separately.

DTSTRF: The kernel performs the LU factorization of a matrix built by coupling the U factor, produced by DGETRF or a previous call to DTSTRF, with a tile below the diagonal tile. The kernel produces an updated U factor and a square matrix L containing the coefficients corresponding to the off-diagonal tile. The new U factor overrides the old U factor. The new L factor overrides the corresponding off-diagonal tile. New pivot vector P is created and stored separately. Due to pivoting, the lower triangular part of the diagonal tile is scrambled and also needs to be stored separately as L'.

DGESSM: The kernel applies the transformations produced by the DGETRF kernel to a tile to the right of the diagonal tile, using the L factor and the pivot vector P.

DSSSSM: The kernel applies the transformations produced by the DTSTRF kernel to the tiles to the right of the tiles factorized by DTSTRF, using the L' factor and the pivot vector P.

Figure 3 shows the pseudocode of the tile LU factorization.

One topic that requires further explanation is the issue of pivoting. Since in the tile algorithm only two tiles of the panel are factorized at a time, pivoting only takes place within two tiles at a time, a scheme which could be described as *block-pairwise pivoting*. Clearly, such pivoting is not equivalent to the "standard" *partial row pivoting* in the block algorithm (e.g., LAPACK). A different pivoting pattern is produced, and also, since pivoting is limited in scope, the procedure could potentially result in a less numerically stable algorithm. More details on the numerical stability of the tile LU algorithm can be found in [7].

IV. The DAGuE Framework

This section introduces the DAGuE framework [2], a new runtime environment system which efficiently schedules dynamically tasks 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, 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 dependencies for each task, decorated with additional information about the behavior of the task.

For an NTxNT 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, DAGuE however 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 dependencies 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 dependencies. Indeed, the knowledge of the IN and OUT dependencies, accessible from any task to any task, ascendant or descendant, 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) present in LAPACK and ScaLAPACK becomes obsolete with this representation as the execution is now data-driven and dynamically scheduled.

Such representation is expected to be internal to the DAGuE framework though, and not a programming language at user disposal. The framework is still in an early stage of development and it does not attempt to compute automatically the data and task distribution. The user is thus required to manually add such information in the JDF.

From a technical point of view, the main goal of the scheduling engine is to select a task for which all the IN dependencies are satisfied, i.e. the data is available locally, select a core where to run the task and execute the body of the task when it is scheduled. Once executed, release all the OUT dependencies of this task, thus making more tasks available to be scheduled. 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 target 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. 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 any shapes of data. 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 appropriately behave when tracking the data dependencies. One should note that the DAGuE engine allows modifier settings on both, input and output dependencies, 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) a task distribution in the process space (DGEQRT(k) runs on the process that verifies the two predicates of lines 5 and 6); 3) a set of data dependencies (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).

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 (and the optional IN keyword). 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 dependencies, marked with an right arrow (and the optional OUT keyword), 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 dependencies are marked with a modifier (line

```
1  DGEQRT(k)  (high_priority)
2    // Execution space
3    k = 0..NT-1
4    // Parallel partitioning
5    : (k / rtileSIZE) % GRIDrows == rowRANK
6    : (k / ctileSIZE) % GRIDcols == colRANK
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   // Parallel partitioning
19   : (m / rtileSIZE) % GRIDrows == rowRANK
20   : (k / ctileSIZE) % GRIDcols == colRANK
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   // Parallel partitioning
41   : (k / rtileSIZE) % GRIDrows == rowRANK
42   : (n / ctileSIZE) % GRIDcols == colRANK
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   // Parallel partitioning
54   : (m / rtileSIZE) % GRIDrows == rowRANK
55   : (n / ctileSIZE) % GRIDcols == colRANK
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)
```

Fig. 4. Concise representation of tile QR factorization

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 dependencies into the DAG which will eventually serialize the DORMQR and DTSQRT calls. This works in the same manner for output dependencies. 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 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.

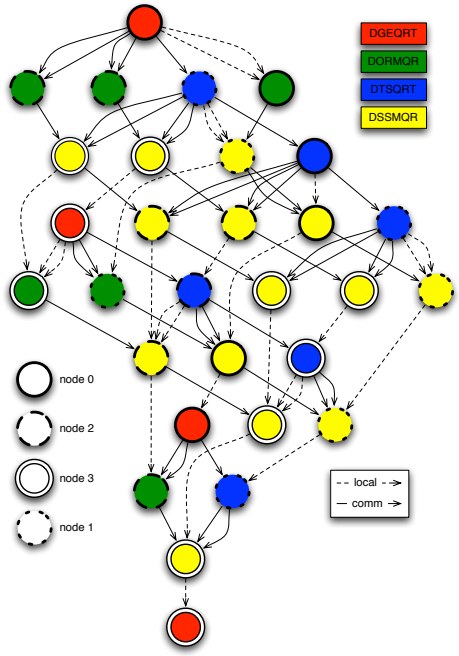

Fig. 5. DAG of QR for a 4x4 tile matrix.

V. Performance Results

This section shows some preliminary results of the tile Cholesky, tile QR and tile LU with the DAGuE engine on today's distributed systems.

A. Hardware Descriptions

The Kraken system is a Cray XT5 with 8256 compute nodes interconnected with SeaStar, a 3D torus. 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.

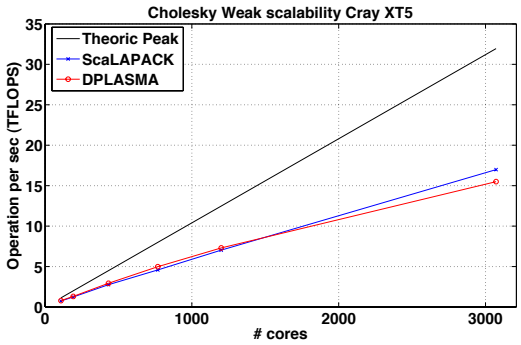

Fig. 6. Cholesky Weak Scalability.

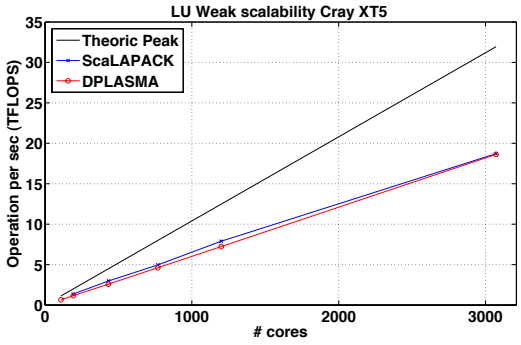

Fig. 7. LU Weak Scalability.

B. 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 factorization strongly depend on tunable execution parameters trading off utilization of different system resources, the outer and the inner blocking sizes (NB and IB). The tile Cholesky depends only on the outer blocking size.

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 of 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 and data transfer. ScaLAPACK uses *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 scalar to tiles when going from LAPACK to PLASMA, we found it useful to raise the level for the data distribution from scalar to tiles when going from ScaLAPACK to DPLASMA. In ScaLAPACK, each process contains an rSIZE x 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 elementwise distribution for ScaLAPACK [32].

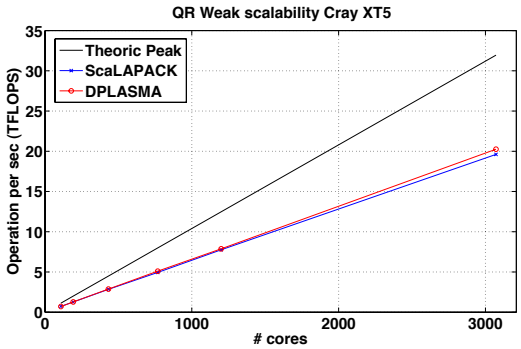

Fig. 8. QR Weak Scalability.

C. Comparisons

We present our results in Figure 6, Figure 7, and Figure 8. As we can see the performances obtained by DAGuE and DPLASMA are comparable to the one of ScaLAPACK in a weak scaling context.

We want to stress out that the conditions in which DAGuE and DPLASMA are tested are not optimal. There are currently a few points that can be improved.

First of, 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 algorithmic is amenable by playing on the data distribution or by changing the way the algorithm moves along the data. Overall, the number of messages sent is $\mathcal{O}((N/NB)*(N/NB))$ and the volume of messages sent is $\mathcal{O}((N/NB)*(N/NB)*NB)$ which is asymptotically more than ScaLAPACK.

Secondly, 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 fine for some operations but not for Cholesky, LU or QR (e.g.). A better way to do the broadcast in this context is with a ring broadcast as done in [33]. DAGuE needs to be able to support broadcast topology provided by the user that are adapted to a given algorithm. The BLACS (communication layer) 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 broadcasted 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 it represent the broadcast of the same data (T and V) from the same root to the same nodes. These two arrays can obviously been concantenated to gain on the latency term.

VI. Summary and Future Works

This paper introduces DPLASMA, a new distributed implementation of three linear algebra kernels (QR, LU, and LLT), based on Tile algorithms and Direct Acyclic Graphs of tasks scheduling for high performance linear algebra computing. It is based on the DAGuE engine, which is capable of extracting tasks and their data dependencies from the sequential nested-loop user application and expressing them using an intermediate concise and synthetic format (JDF). The engine then schedules the generated tasks across a distributed system without having to unroll the whole DAG. DAGuE is still at early stage of development and the results shown on this paper are very encouraging.

Our current results show that DAGuE is a promising framework that provides a scalable and dynamic scheduling for parallel distributed machines. For parallel distributed code development, one often has to rely on static scheduling of the operations. This lends to an important complexity of coding. Parallel distributed dynamic schedulers offer a solution however current dynamic parallel distributed schedulers are intrinsically limited in scalability. We have proven that our dynamic scheduler DAGuE scales up to 5,000 cores and its scalability from 1 to 5,000 cores has not been hindered the least (as expected by our design).

REFERENCES

[1] University of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Archtectures, Version 2.0*, 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. Technical Report ICL-UT-10-01, Innovative Computing Laboratory, University of Tennessee, 2010. Submitted at SC'10.

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

[4] G. H. Golub and C. F. C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996. 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. DOI: 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. DOI: 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. DOI: 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. DOI: TPDS.2007.70813.

[10] J. Kurzak and J. J. Dongarra. QR factorization for the CELL processor. *Scientific Programming*, 17(1-2):31–42, 2009. DOI: 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. DOI: 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. ISBN: 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. DOI: 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. DOI: 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. DOI: 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] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[23] J. J. Dongarra, J. Du Croz, I. S. Duff, , and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[24] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. http://www.netlib.org/lapack/lug/.

[25] C. Bischof and C. van Loan. The WY representation for products of Householder matrices. *J. Sci. Stat. Comput.*, 8:2–13, 1987.

[26] R. Schreiber and C. van Loan. A storage-efficient WY representation for products of Householder transformations. *J. Sci. Stat. Comput.*, 10:53–57, 1991.

[27] B. C. Gunter and R. A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, 2005. DOI: 10.1145/1055531.1055534.

[28] E. Chan, E. S. Quintana-Orti, G. Gregorio Quintana-Orti, and R. van de Geijn. Supermatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07*, pages 116–125, June 2007.

[29] J. Demmel, L. Grigori, M. F. Hoemmen, , and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report arXiv:0808.2664v1, 2008.

[30] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998. ISBN: 0898714281.

[31] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713897.

[32] 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.

[33] 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.