Scheduling Linear Algebra Operations on Multicore Processors
– LAPACK Working Note 213

Jakub Kurzak
Department of Electrical Engineering and Computer Science, University of Tennessee

Hatem Ltaief
Department of Electrical Engineering and Computer Science, University of Tennessee

Jack Dongarra
Department of Electrical Engineering and Computer Science, University of Tennessee
Computer Science and Mathematics Division, Oak Ridge National Laboratory
School of Mathematics & School of Computer Science, University of Manchester

Rosa M. Badia
Barcelona Supercomputing Center - Centro Nacional de Supercomputación

ABSTRACT

State-of-the-art dense linear algebra software, such as the LAPACK and ScaLAPACK libraries, suffer performance losses on multicore processors due to their inability to fully exploit thread-level parallelism. At the same time the coarse-grain dataflow model gains popularity as a paradigm for programming multicore architectures. This work looks at implementing classic dense linear algebra workloads, Cholesky factorization, QR factorization and LU factorization, using dynamic data-driven execution. Two emerging approaches to implementing coarse-grain dataflow are examined, the model of nested parallelism, represented by the Cilk framework, and the model of parallelism expressed through an arbitrary Direct Acyclic Graph, represented by the SMP Superscalar framework. Performance and coding effort are analyzed and compared agains code manually parallelized at the thread level.

KEYWORDS: task graph, scheduling, multicore, linear algebra, factorization, Cholesky, LU, QR

1 Introduction & Motivation

The current trend in the semiconductor industry to double the number of execution units on a single die is commonly referred to as *the multicore discontinuity*. This term reflects the fact that existing software model is inadequate for the new architectures and existing code base will be incapable of delivering increased performance, possibly not even capable of sustaining current performance.

This problem has already been observed with state-of-the-art dense linear algebra libraries, LAPACK [1] and ScaLAPACK [2], which deliver a small fraction of peak performance on current multicore processors and multi-socket systems of multicore processors, mostly following *Symmetric Multi-Processor* (SMP) architecture.

The problem is twofold. Achieving good performance on emerging chip designs is a serious problem, calling for new algorithms and data structures. Reimplementing existing code base using a new programming paradigm is another major challenge, specifically in the area of high performance scientific computing, where the level of required skills makes the programmers a scarce resource and millions of lines of code are in question.

2 Background

In large scale scientific computing, targeting distributed memory systems, the recent push towards the PetaFlop barrier caused a renewed interest in *Partitioned Global Address Space* (PGAS) languages, such as *Co-Array Fortran* (CAF) [3], *Unified Parallel C* (UPC) [4] or Titanium [5], as well as emergence of new languages, such as Chapel (Cray) [6], Fortress (Sun) [7] and X-10 (IBM) [8], sponsored through the DARPA's *High Productivity Computing Systems* (HPCS) program.

In more mainstream, server and desktop computing, targeting mainly shared memory systems, the well known *dataflow model* is rapidly gaining popularity, where the computation is viewed as a *Direct Acyclic Graph* (DAG), with nodes representing computational tasks and edges representing data dependencies among them. The coarse-grain dataflow model is the main principle behind emerging multicore programming environments such as *Cilk/Cilk++* [9], Intel® *Threading Building Blocks* (TBB) [10, 11], Tasking in OpenMP 3.0 [12, 13, 14, 15] and *SMP Superscalar* (SMPSs) [16].

All these frameworks rely on a very small set of extensions to common imperative programming languages such as C/C++ and Fortran and involve a relatively simple compilation stage and potentially much more complex runtime system.

The following sections provide a brief overview of these frameworks, as well as an overview of a rudimentary scheduler implemented using POSIX threads, which will serve as a baseline for performance comparisons.

Since tasking facilities available in Threading Building Blocks and OpenMP 3.0 closely resemble the ones provided by Cilk, Cilk is chosen as a representative framework for all three (also due to the reason that, same as SMPSs, it is available in open-source).

2.1 Cilk

Cilk was developed at the MIT Laboratory for Computer Science starting in 1994 [9]. Cilk is an extension of the C language with a handful of keywords (*cilk, spawn, sync, inlet, abort*) aimed at providing general-purpose programming language designed for multithreaded parallel programming. When the Cilk keywords are removed from Cilk source code, the result is a valid C program, called the *serial elision* (or C elision) of the full Cilk program. The Cilk environment employs a source-to-source compiler, which compiles Cilk code to C code, a standard C compiler, and a runtime system linked with the object code to provide an executable.

The main principle of Cilk is that the programmer is responsible for exposing parallelism by identifying functions free of side effects (e.g., access to global variables causing race conditions), which can be treated as independent tasks and executed in parallel. Such functions are annotated with the *cilk* keyword and invoked with the *spawn* keyword. The *sync* keyword is used to indicate that execution of the current procedure cannot proceed until all previously spawned procedures have completed and returned their results to the parent.

Distribution of work to multiple processors is handled by the runtime system. Cilk scheduler uses the policy called *work-stealing* to schedule execution of tasks to multiple processors. At run time, each processor fetches tasks from the top of its own stack - in *First In First Out* (FIFO) order. However, when a processor runs out of tasks, it picks another processor at random and "steals" tasks from the bottom of its stack - in *Last In First Out* (LIFO) order. This way the task graph is consumed in a *depth-first* order, until a processor runs out of tasks, in which case it steals tasks from other processors in a *breadth-first* order.

Cilk also provides the mechanism of locks. The use of lock can, however, easily lead to deadlock. "Even if the user can guarantee that his program is deadlock-free, Cilk may still deadlock on the user's code because of some additional scheduling constraints imposed by Cilk's scheduler" [17]. In particular locks cannot be used to enforce parent-child dependencies between tasks.

Cilk is very well suited for expressing algorithms which easily render themselves to recursive formulation, e.g., *divide-and-conquer* algorithms. Since stack is the main structure for controlling parallelism, the

model allows for straightforward implementations on shared memory multiprocessor systems (e.g., multicore/SMP systems). The simplicity of the model provides for execution of parallel code with virtually no overhead from scheduling.

2.2 OpenMP

OpenMP was born in the '90s to bring a standard to the different directive languages defined by several vendors. The different characteristics of this approach: simplicity of the interface, use of a shared memory model, and the use of loosely-coupled directives to express the parallelism of a program, make it very well-accepted today. Due to new needs of the parallel applications, OpenMP has been recently extended to add, in its version 3.0, a tasking model that addresses new programming model aspects.

The new OpenMP directives allows the programmer to identify units of independent work (tasks), leaving the decision to how and when to execute them to the runtime system.

This gives the programmers a way of expressing patterns of concurrency that do not match the worksharing constructs defined in the OpenMP 2.5 specification. The main difference between Cilk and OpenMP 3.0 is that the latter can combine both types of parallelism, worksharing and tasks: for example, a programmer can choose to use OpenMP tasks to exploit the parallelism of an inner loop and the traditional worksharing construct to parallelize an outer loop.

2.3 Intel® Threading Building Blocks

Intel®Threading Building Blocks is a runtime-based parallel programming model for C++ code that uses threads. The main difference with other threading packages is that it enables the programmer to specify tasks instead of threads and the runtime library automatically schedules tasks onto threads in a way that makes efficient use of a multicore processor.

Another characteristic of TBB is that it focuses on the particular goal of parallelizing computationally intensive work, while this is not always true in general-purpose threading packages. TBB emphasizes data-parallel programming, enabling multiple threads to work on different parts of a collection enabling scalability to larger number of cores.

The programming model is based on template functions (parallel_for, parallel_reduce, etc.), where the user specifies the range of data to be accessed, how to partition the data, the task to be executed in each chunk.

2.4 SMPSs

SMP Superscalar (SMPSs) [16] is a parallel programming framework developed at the Barcelona Supercomputer Center (Centro Nacional de Supercomputación), part of the STAR Superscalar family, which also includes Grid Supercalar and Cell Superscalar [18, 19]. While Grid Superscalar and Cell Superscalar address parallel software development for Grid enviroments and the Cell processor respectively, SMP Superscalar is aimed at "standard" (x86 and like) multicore processors and symmetric multiprocessor systems.

The principles of SMP Superscalar are similar to the ones of Cilk. Similarly to Cilk, the programmer is responsible for identifying parallel tasks, which have to be side-effect-free (atomic) functions. Additionally, the programmer needs to specify the directionality of each parameter (input, output, inout). If the size of a parameter is missing in the C declaration (e.g., the parameter is passed by pointer), the programmer also needs to specify the size of the memory region affected by the function. Unlike Cilk, however, the programmer is not responsible for exposing the structure of the task graph. The task graph is built automatically, based on the information of task parameters and their directionality.

Similarly to Cilk, the programming environment consists of a source-to-source compiler and a supporting runtime library. The compiler translates C code with pragma annotations to standard C99 code with calls to the supporting runtime library and compiles it using the platform native compiler.

At runtime the main thread creates worker threads, as many as necessary to fully utilize the system, and starts constructing the task graph (populating its

ready list). Each worker thread maintains its own ready list and populates it while executing tasks. A thread consumes tasks from its own ready list in LIFO order. If that list is empty, the thread consumes tasks from the main ready list in FIFO order, and if that list is empty, the thread steals tasks from the ready lists of other threads in FIFO order.

The SMPSs scheduler attempts to exploit locality by scheduling dependent tasks to the same thread, such that output data is reused immediately. Also, in order to reduce dependencies, SMPSs runtime is capable of renaming data, leaving only the true dependencies, which is the same technique used by superscalar processors [20] and optimizing compilers [21].

The main difference between Cilk and SMPSs is that, while the former allows mainly for expression of nested parallelism, the latter handles computation expressed as an arbitrary DAG. Also, while Cilk requires the programmer to create the DAG by means of the spawn keyword, SMPSs creates the DAG automatically. Construction of the DAG does, however, introduce overhead, which is virtually inexistent in the Cilk environment.

2.5 Static Pipeline

The *static pipeline* scheduling presented here was originally implemented for dense matrix factorizations on the CELL processor [22, 23]. This technique is extremely simple and yet provides good locality of reference and load balance for regular computation, like dense matrix operations.

In this approach each task is uniquely identified by the $\{m, n, k\}$ triple, which determines the type of operation and the location of tiles operated upon. Each core traverses its task space by applying a simple formula to the $\{m, n, k\}$ triple, which takes into account the *id* of the core and the total number of cores in the system.

Task dependencies are tracked by a global progress table, where one element describes progress of computation for one tile of the input matrix. Each core looks up the table before executing each task to check for dependencies and stalls if dependencies are not satisfied. Each core updates the progress table after completion of each task. Access to the table does not require mutual exclusion (using, e.g., mutexes). The table is declared as *volatile*. Update is implemented by writing to an element. Dependency stall is implemented by *busy-waiting* on an element.

The use of a global progress table is a potential scalability bottleneck. It does not pose a problem, however, on small-scale multicore/SMP systems for small to medium matrix sizes. Many alternatives are possible. (Replicated progress tables were used on the CELL processor [22, 23]).

As further discussed in sections 4.3 and 5.3, this technique allows for pipelined execution of factorizations steps, which provides similar benefits to dynamic scheduling, namely, execution of the inefficient Level 2 BLAS operations in parallel with the efficient Level 3 BLAS operations.

The main disadvantage of the technique is potentially suboptimal scheduling, i.e., stalling in situations where work is available. Another obvious weakness of the static schedule is that it cannot accommodate dynamic operations, e.g., *divide-and-conquer* algorithms.

3 Related Work

Dynamic data-driven scheduling is an old concept and has been applied to dense linear operations for decades on various hardware systems. The earliest reference, that the authors are aware of, is the paper by Lord, Kowalik and Kumar [24]. A little later dynamic scheduling of LU and Cholesky factorizations were reported by Agarwal and Gustavson [25, 26] Throughout the years dynamic scheduling of dense linear algebra operations has been used in numerous vendor library implementations such as ESSL, MKL and ACML (numerous references are available on the Web). In recent years the authors of this work have been investigating these ideas within the framework *Parallel Linear Algebra for Multicore Architectures* (PLASMA) at the University of Tennessee [27, 28, 29, 30]. Noteworthy is the implementation of *sparse* Cholesky factorization by Irony et al. using Cilk [31].

Seminal work leading to the *tile QR* algorithm presented here was done by Elmroth et al. [32,

33, 34]. Gunter et al. presented an "out-of-core" (out-of-memory) implementation [35], Buttari et al. an implementation for "standard" (x86 and alike) multicore processors [27, 28], and Kurzak et al. an implementation for the CELL processor [22]. The LU algorithm used here was originally devised by Quintana-Ortí et al. for "out-of-core" (out-of-memory) execution [45].

Seminal work on performance-oriented data layouts for dense linear algebra was done by Gustavson et al. [36, 37] and Elmroth et al. [38] and was also investigated by Park et al. [39, 40].

4 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. Here the lower triangular case is considered.

The algorithm can be expressed using either the top-looking version, the left-looking version of the right-looking version, the first being the most *lazy* algorithm (depth-first exploration of the task graph) and the last being the most *aggressive* algorithm (breadth-first exploration of the task graph). The left-looking variant is used here, with the exception of Cilk implementations, which favor the most aggressive right-looking variant.

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 (Figure 1).

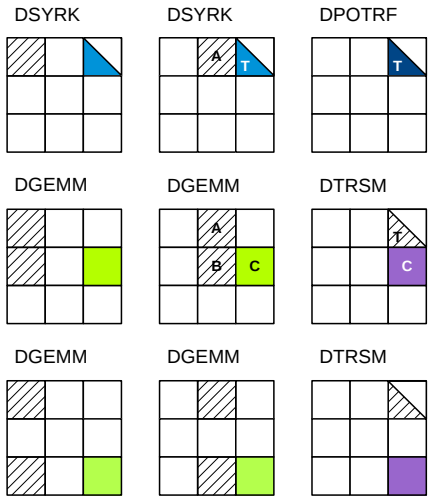

Figure 1: Tile operations in the tile Cholesky factorization.

DSYRK: The kernel applies updates to a diagonal (lower triangular) tile T of the input matrix, resulting from factorization of the tiles A to the left of it. The operation is a symmetric rank-k update.

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

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

DTRSM: The operation applies an update to an off-diagonal tile C of the input matrix, resulting from factorization of the diagonal tile above it and overrides it with the final elements of the output matrix. The operation is a triangular solve.

Figure 2 shows the pseudocode of the left-looking Cholesky factorization. Figure 3 shows the task graph of the tile Cholesky factorization of a 5×5 tiles matrix. Although the code is as simple as four loops with three levels of nesting, the task graph is far from intuitive, even for a tiny size.

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

Figure 2: Pseudocode of the tile Cholesky factorization (left-looking version).

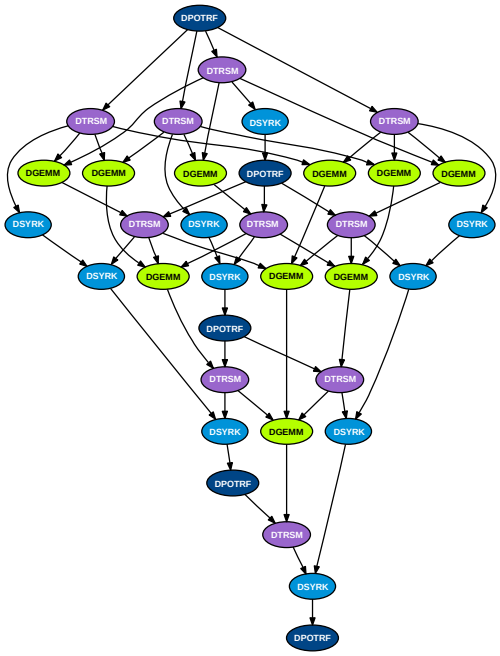

Figure 3: Task graph of the tile Cholesky factorization (5×5 tiles).

4.1 Cilk Implementation

Figure 4 presents implementation of Cholesky factorization in Cilk. The basic building blocks are the functions performing the tile operations. *dsyrk()*, *dtrsm()* and *dgemm()* are implemented by calls to a single BLAS routine. *dpotrf()* is implemented by a call to the LAPACK DPOTRF routine. The functions are declared using the *cilk* keyword and then invoked using the *spawn* keyword.

The input matrix is stored using the format referred to in literature as *Square Block* (SB) format or *Block Data Layout* (BDL). The latter name will be used here. In this arrangement, each function parameter is a pointer to a continuous block of memory, what greatly increases cache performance and virtually eliminates cache conflicts between different operations.

For implementation in Cilk the right-looking variant was chosen, where factorization of each panel is followed by an update to all the remaining submatrix. The code on Figure 4 presents a version, referred here as *Cilk 2D*, where task scheduling is not constrained by data reuse considerations (There are no provisions for reuse of data between different tasks).

Each step of the factorization involves:

- factorization of the diagonal tile - spawning of the *dpotrf()* task followed by a *sync*,
- applying triangular solves to the tiles below the diagonal tile - spawning of the *dtrsm()* tasks in parallel followed by a *sync*,
- updating the tiles to the right of the panel - spawning of the *dsyrk()* and *dgemm()* tasks in parallel followed by a *sync*.

```
cilk void dsyrk(double *A, double *T);
cilk void dpotrf(double *T);
cilk void dgemm(double *A, double *B, double *C);
cilk void dtrsm(double *T, double *C);


for (k = 0; k < TILES; k++) {

   spawn dpotrf(A[k][k]);
   sync;

   for (m = k+1; m < TILES; m++)
      spawn dtrsm(A[k][k], A[m][k]);
   sync;

   for (m = k+1; m < TILES; m++) {
      for (n = k+1; n < m; n++)
         spawn dgemm(A[k][n], A[m][n], A[m][k]);
      spawn dsyrk(A[k][n], A[k][k]);
   }
   sync;
}
```

Figure 4: Cilk implementation of the tile Cholesky factorization with 2D work assignment (right-looking version).

It is not possible to further improve parallelism by pipelining the steps of the factorization. Nevertheless, most of the work can proceed in parallel and only the *dpotrf()* task has to be executed sequentially.

Since the disregard for data reuse between tasks may adversely affect the algorithm's performance, it is necessary to consider an implementation facilitating data reuse. One possible approach is processing of the tiles of the input matrix by columns. In this case, however, work is being dispatched in relatively big batches and load imbalance in each step of the factorization will affect performance. A traditional remedy to this problem is the technique of *lookahead*, where update of step N is applied in parallel with panel factorization of step $N + 1$. Figure 5 shows such implementation, referred here as *Cilk 1D*.

First, panel 0 is factorized, followed by a *sync*. Then updates to all the remaining columns are issued in parallel. Immediately after updating the first column, next panel factorization is spawned. The code synchronizes at each step, but panels are always overlapped with updates. This approach implements one-level lookahead (lookahead of depth one). Implementing more levels of lookahead would further complicate the code.

```
void dsyrk(double *A, double *T);
void dpotrf(double *T);
void dgemm(double *A, double *B, double *C);
void dtrsm(double *T, double *C);

cilk void cholesky_panel(int k)
{
   int m;

   dpotrf(A[k][k]);

   for (m = k+1; m < TILES; m++)
      dtrsm(A[k][k], A[m][k]);
}

cilk void cholesky_update(int n, int k)
{
   int m;

   dsyrk(A[k][n], A[k][k]);

   for (m = n+1; m < TILES; m++)
      spawn dgemm(A[k][n], A[m][n], A[m][k]);

   if (n == k+1)
      spawn cholesky_panel(k+1);

}


spawn cholesky_panel(0);
sync;

for (k = 0; k < TILES; k++) {
   for (n = k+1; n < TILES; n++)
      spawn cholesky_update(n, k);
   sync;
}
```

Figure 5: Cilk implementation of the tile Cholesky factorization with 1D work assignment (right-looking version).

4.2 SMPSs Implementation

Figure 6 shows implementation using SMPSs. The functions implementing parallel tasks are designated with *#pragma ccs task* annotations defining directionality of the parameters (input, output, inout). The parallel section of the code is designated with *#pragma ccs start* and *#pragma ccs finish* annotations. Inside the parallel section the algorithm is implemented using the canonical representation of four loops with three levels of nesting, which closely

matches the pseudocode definition of Figure 2.

```
#pragma css task input(A[NB][NB]) inout(T[NB][NB])
void dsyrk(double *A, double *T);

#pragma css task inout(T[NB][NB])
void dpotrf(double *T);

#pragma css task input(A[NB][NB], B[NB][NB]) inout(C[NB][NB])
void dgemm(double *A, double *B, double *C);

#pragma css task input(T[NB][NB]) inout(B[NB][NB])
void dtrsm(double *T, double *C);


#pragma css start
for (k = 0; k < TILES; k++) {

   for (n = 0; n < k; n++)
      dsyrk(A[k][n], A[k][k]);
   dpotrf(A[k][k]);

   for (m = k+1; m < TILES; m++) {
      for (n = 0; n < k; n++)
         dgemm(A[k][n], A[m][n], A[m][k]);
      dtrsm(A[k][k], A[m][k]);
   }
}
#pragma css finish
```

Figure 6: SMPSs implementation of the tile Cholesky factorization (left-looking version).

The SMPSs runtime system schedules tasks based on dependencies and attempts to maximize data reuse by following the parent-child links in the task graph when possible.

4.3 Static Pipeline Implementation

As already mentioned in section 2.5 the *static pipeline* implementation is a hand-written code using POSIX threads and primitive synchronization mechanisms (*volatile* progress table and busy-waiting). Figure 7 shows the implementation.

The code implements the left-looking version of the factorization, where work is distributed by rows of tiles and steps of the factorization are pipelined. The first core that runs out of work in step N proceeds to factorization of the panel in step $N+1$, following cores proceed to update in step $N+1$, then to panel in step $N+2$ and so on (Figure 8).

```
void dsyrk(double *A, double *T);
void dpotrf(double *T);
void dgemm(double *A, double *B, double *C);
void dtrsm(double *T, double *C);


k = 0; m = my_core_id;
while (m >= TILES) {
   k++; m = m-TILES+k;
} n = 0;

while (k < TILES && m < TILES) {
   next_n = n; next_m = m; next_k = k;

   next_n++;
   if (next_n > next_k) {
      next_m += cores_num;
      while (next_m >= TILES && next_k < TILES) {
         next_k++; next_m = next_m-TILES+next_k;
      } next_n = 0;
   }

   if (m == k) {
      if (n == k) {
         dpotrf(A[k][k]);
         core_progress[k][k] = 1;
      }
      else {
         while(core_progress[k][n] != 1);
         dsyrk(A[k][n], A[k][k]);
      }
   }
   else {
      if (n == k) {
         while(core_progress[k][k] != 1);
         dtrsm(A[k][k], A[m][k]);
         core_progress[m][k] = 1;
      }
      else {
         while(core_progress[k][n] != 1);
         while(core_progress[m][n] != 1);
         dgemm(A[k][n], A[m][n], A[m][k]);
      }
   }
   n = next_n; m = next_m; k = next_k;
}
```

Figure 7: Static pipeline implementation of the tile Cholesky factorization (left-looking version).

The code can be viewed as a parallel implementation of Cholesky factorization with one dimensional partitioning of work and lookahead, where lookahead of varying depth is implemented by processors which run out of work.

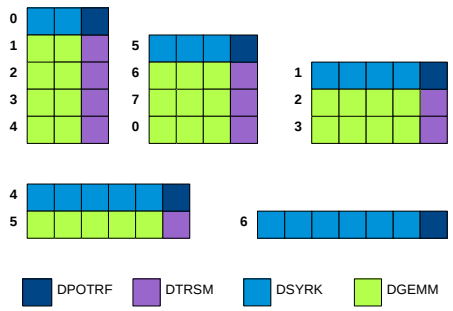

Figure 8: Work assignment in the static pipeline implementation of the tile Cholesky factorization.

5 QR Factorization

The QR factorization (or QR decomposition) offers a numerically stable way of solving underdetermined and overdetermined systems of linear equations (least squares problems) and is also the basis for the *QR algorithm* for solving the eigenvalue problem.

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

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

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 [41, 42]. 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 very 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") [35] and, more recently, to produce high performance implementation on "standard" (x86 and alike) multicore processors [27, 28] and on the CELL processor [22].

The algorithm is based on the idea of annihilating matrix elements by square tiles instead of rectangular panels (block columns). The algorithm produces 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. Whether the Q matrix is actually needed depends on the application. The tile QR algorithm relies on four basic operations implemented by four computational kernels (Figure 9).

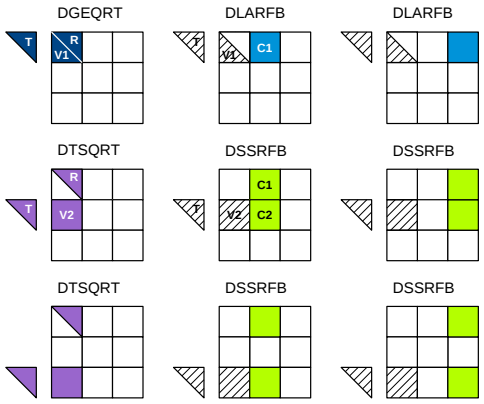

Figure 9: Tile operations in the tile QR factorization.

DGEQRT: The kernel performs the QR factorization of a diagonal tile of the input matrix 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 [41, 42]. 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 square tile of the input matrix. The T matrix is stored separately.

DLARFB: 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.

DSSRFB: 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.

Naive implementation, where the full T matrix is built, results in 25 % more floating point operations than the standard algorithm. In order to minimize this overhead, the idea of *inner-blocking* is used, where the T matrix has sparse (block-diagonal) structure (Figure 10) [32, 33, 34].

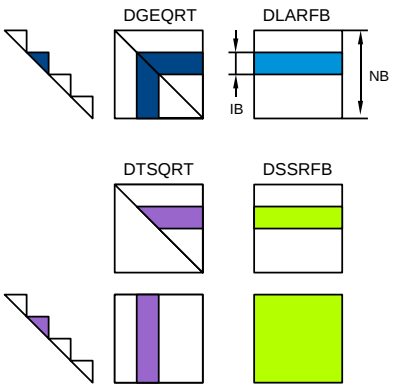

Figure 10: Inner blocking in the tile QR factorization.

Figure 11 shows the pseudocode of the tile QR factorization. Figure 12 shows the task graph of the tile QR factorization for a matrix of 5×5 tiles. Orders of magnitude larger matrices are used in practice. This example only serves the purpose of showing the complexity of the task graph, which is noticeably higher than that of Cholesky factorization.

```
FOR k = 0..TILES-1
  A[k][k], T[k][k] ← DGRQRT(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] ← DLARFB(A[k][k], T[k][k], A[k][n])
    FOR m = k+1..TILES-1
      A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
```

Figure 11: Pseudocode of the tile QR factorization.

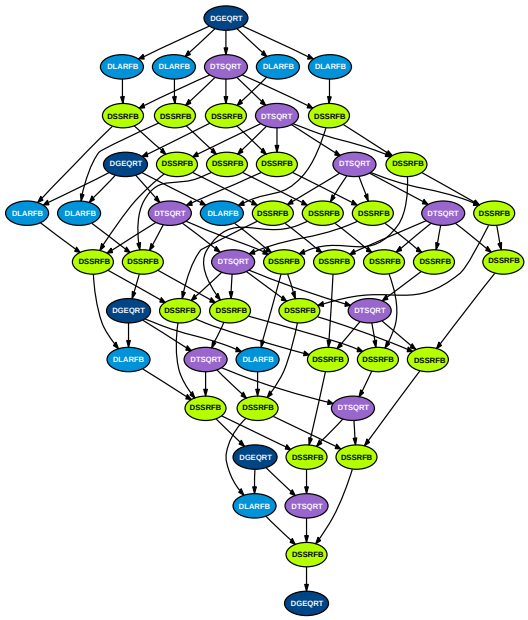

Figure 12: Task graph of the tile QR factorization (matrix of size 5×5 tiles).

5.1 Cilk Implementation

The task graph of the tile QR factorization has a much denser net of dependencies than the Cholesky factorization. Unlike for Cholesky the tasks factorizing the panel are not independent and have to be serialized and the tasks applying the update have to follow the same order. The order can be arbitrary. Here top-down order is used.

Figure 13 shows the first Cilk implementation, referred to as *Cilk 2D*, which already requires the use of lookahead to achieve performance. The basic building block are the functions performing the tile operations. Unlike for Cholesky, none of them is a simple call to BLAS or LAPACK. Due to the use of inner-blocking the kernels consist of loop nests containing a number of BLAS and LAPACK calls (currently coded in FORTRAN 77).

The factorization proceeds in the following steps:

- Initially the first diagonal tile is factorized - spawning of the *dgeqrt()* task followed by a *sync*. Then the main loop follows with the remaining steps.
- Tiles to the right of the diagonal tile are updated in parallel with factorization of the tile immediately below the diagonal tile - spawning of the *dlarfb()* tasks and the *dtsqrt()* task followed by a *sync*.
- Updates are applied to the tiles right from the panel - spawning of the *dssrfb()* tasks by rows of tiles (*sync* following each row). The last *dssrfb()* task in a row spawns the *dtsqrt()* task in the next row. The last *dssrfb()* task in the last row spawns the *dgeqrt()* task in the next step of the factorization.

Although lookahead is used and factorization of the panel is, to some extent, overlapped with applying the update, tasks are being dispatched in smaller batches, what severely limits opportunities for scheduling.

The second possibility is to process the tiles of the input matrix by columns, the same as was done for Cholesky. Actually, it is much more natural to do it in the case of QR, where work within a column has to be serialized. Load imbalance comes into picture again and lookahead is the remedy. Figure 14 shows the implementation, referred to as *Cilk 1D*.

```
cilk void dgeqrt(double *RV1, double *T);
cilk void dtsqrt(double *R, double *V2, double *T);
cilk void dlarfb(double *V1, double *T, double *C1);
void dssrfb(double *V2, double *T, double *C1, double *C2);

cilk void dssrfb_(int m, int n, int k)
{
   dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);

   if (m == TILES-1 && n == k+1 && k+1 < TILES)
      spawn dgeqrt(A[k+1][k+1], T[k+1][k+1]);

   if (n == k+1 && m+1 < TILES)
      spawn dtsqrt(A[k][k], A[m+1][k], T[m+1][k]);
}


spawn dgeqrt(A[0][0], T[0][0]);
sync;

for (k = 0; k < TILES; k++) {

   for (n = k+1; n < TILES; n++)
      spawn dlarfb(A[k][k], T[k][k], A[k][n]);

   if (k+1 < TILES)
      spawn dtsqrt(A[k][k], A[k+1][k], T[k+1][k]);
   sync;

   for (m = k+1; m < TILES; m++) {
      for (n = k+1; n < TILES; n++)
         spawn dssrfb_(m, n, k);
      sync;
   }
}
```

Figure 13: Cilk implementation of the tile QR factorization with 2D work assignment and lookahead.

The implementation follows closely the *Cilk 1D* version of Cholesky. First, panel 0 is factorized, followed by a *sync*. Then updates to all the remaining columns are issued in parallel. Immediately after updating the first column, next panel factorization is spawned. The code synchronizes at each step, but panels are always overlapped with updates. This approach implements one-level lookahead (lookahead of depth one). Implementing more levels of lookahead would further complicate the code.

```
void dgeqrt(double *RV1, double *T);
void dtsqrt(double *R, double *V2, double *T);
void dlarfb(double *V1, double *T, double *C1);
void dssrfb(double *V2, double *T, double *C1, double *C2);

cilk void qr_panel(int k)
{
   int m;

   dgeqrt(A[k][k], T[k][k]);

   for (m = k+1; m < TILES; m++)
      dtsqrt(A[k][k], A[m][k], T[m][k]);
}

cilk void qr_update(int n, int k)
{
   int m;

   dlarfb(A[k][k], T[k][k], A[k][n]);

   for (m = k+1; m < TILES; m++)
      dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);

   if (n == k+1)
      spawn qr_panel(k+1);
}


spawn qr_panel(0);
sync;

for (k = 0; k < TILES; k++) {
   for (n = k+1; n < TILES; n++)
      spawn qr_update(n, k);
   sync;
}
```

Figure 14: Cilk implementation of the tile QR factorization with 1D work assignment and lookahead.

```
#pragma css task \
   inout(RV1[NB][NB]) output(T[NB][NB])
void dgeqrt(double *RV1, double *T);

#pragma css task \
   inout(R[NB][NB], V2[NB][NB]) output(T[NB][NB])
void dtsqrt(double *R, double *V2, double *T);

#pragma css task \
   input(V1[NB][NB], T[NB][NB]) inout(C1[NB][NB])
void dlarfb(double *V1, double *T, double *C1);

#pragma css task \
   input(V2[NB][NB], T[NB][NB]) inout(C1[NB][NB], C2[NB][NB])
void dssrfb(double *V2, double *T, double *C1, double *C2);


#pragma css start
for (k = 0; k < TILES; k++) {

   dgeqrt(A[k][k], T[k][k]);

   for (m = k+1; m < TILES; m++)
      dtsqrt(A[k][k], A[m][k], T[m][k]);

   for (n = k+1; n < TILES; n++) {
      dlarfb(A[k][k], T[k][k], A[k][n]);
      for (m = k+1; m < TILES; m++)
         dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
   }
}
#pragma css finish
```

Figure 15: SMPSs implementation of the tile QR factorization.

5.2 SMPSs Implementation

Figure 15 shows implementation using SMPSs, which follows closely the one for Cholesky. The functions implementing parallel tasks are designated with *#pragma ccs task* annotations defining directionality of the parameters (input, output, inout). The parallel section of the code is designated with *#pragma ccs start* and *#pragma ccs finish* annotations. Inside the parallel section the algorithm is implemented using the canonical representation of four loops with three levels of nesting, which closely matches the pseudocode definition of Figure 11.

The SMPSs runtime system schedules tasks based on dependencies and attempts to maximize data reuse by following the parent-child links in the task graph when possible.

There is a caveat here, however. V1 is an input parameter of task *dlarfb()*. It also is an inout parameter of task *dtsqrt()*. However, *dlarfb()* only reads the lower triangular portion of the tile, while *dtsqrt()* only updates the upper triangular portion of the tile. Since in both cases the tile is passed to the functions by the pointer to the upper left corner of the tile, SMPSs sees a false dependency. As a result, the execution of the *dlarfb()* tasks in a given step will be stalled until all the *dtsqrt()* tasks complete, despite the fact that both types of tasks can be scheduled in parallel as soon as the *dgeqrt()* task completes. Figure 16 shows conceptually the change that needs to be done.

Currently SMPSs is not capable of recognizing accesses to triangular matrices. There are however multiple ways to enforce the correct behavior. The sim-

```
#pragma css task \
   inout(RV1[NB][NB]) output(T[NB][NB])
void dgeqrt(double *RV1, double *T);

#pragma css task \
   inout(R[◥], V2[NB][NB]) output(T[NB][NB])
void dtsqrt(double *R, double *V2, double *T);

#pragma css task \
   input(V1[◣], T[NB][NB]) inout(C1[NB][NB])
void dlarfb(double *V1, double *T, double *C1);

#pragma css task \
   input(V2[NB][NB], T[NB][NB]) inout(C1[NB][NB], C2[NB][NB])
void dssrfb(double *V2, double *T, double *C1, double *C2);


#pragma css start
for (k = 0; k < TILES; k++) {

   dgeqrt(A[k][k], T[k][k]);

   for (m = k+1; m < TILES; m++)
      dtsqrt(A[k][k]◥, A[m][k], T[m][k]);

   for (n = k+1; n < TILES; n++) {
      dlarfb(A[k][k]◣, T[k][k], A[k][n]);
      for (m = k+1; m < TILES; m++)
         dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
   }
}
#pragma css finish
```

Figure 16: SMPSs implementation of the tile QR factorization with improved dependency resolution for diagonal tiles.

plest method, in this case, is to drop dependency check on the V1 parameter of the *dlarfb()* function by declaring it as *volatile**. Correct dependency will be enforced between the *dgeqrt()* task and the *dlarfb()* tasks through the T parameter. This implementation is further referred to as *SMPSs**.

5.3 Static Pipeline Implementation

The *static pipeline* implementation for QR is very close to the one for Cholesky. As already mentioned in section 2.5 the *static pipeline* implementation is a hand-written code using POSIX threads and primitive synchronization mechanisms (*volatile* progress table and busy-waiting). Figure 17 shows the implementation.

The code implements the right-looking version

```
void dgeqrt(double *RV1, double *T);
void dtsqrt(double *R, double *V2, double *T);
void dlarfb(double *V1, double *T, double *C1);
void dssrfb(double *V2, double *T, double *C1, double *C2);


k = 0; n = my_core_id;
while (n >= TILES) {
   k++; n = n-TILES+k;
} m = k;

while (k < TILES && n < TILES) {
   next_n = n; next_m = m; next_k = k;

   next_m++;
   if (next_m == TILES) {
      next_n += cores_num;
      while (next_n >= TILES && next_k < TILES) {
         next_k++; next_n = next_n-TILES+next_k;
      } next_m = next_k;
   }

   if (n == k) {
      if (m == k) {
         while(progress[k][k] != k-1);
         dgeqrt(A[k][k], T[k][k]);
         progress[k][k] = k;
      }
      else{
         while(progress[m][k] != k-1);
         dtsqrt(A[k][k], A[m][k], T[m][k]);
         progress[m][k] = k;
      }
   }
   else {
      if (m == k) {
         while(progress[k][k] != k);
         while(progress[k][n] != k-1);
         dlarfb(A[k][k], T[k][k], A[k][n]);
      }
      else {
         while(progress[m][k] != k);
         while(progress[m][n] != k-1);
         dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
         progress[m][n] = k;
      }
   }
   n = next_n; m = next_m; k = next_k;
}
```

Figure 17: Static pipeline implementation of the tile QR factorization.

of the factorization, where work is distributed by columns of tiles and steps of the factorization are pipelined. The first core that runs out of work in step N proceeds to factorization of the panel in step $N+1$, following cores proceed to update in step $N+1$, then to panel in step $N+2$ and so on (Figure 18).

The code can be viewed as a parallel implementation of the tile QR factorization with one dimensional

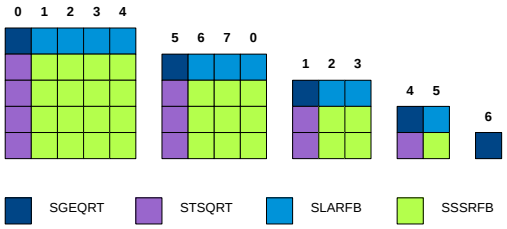

Figure 18: Work assignment in the static pipeline implementation of the tile QR factorization.

partitioning of work and lookahead, where lookahead of varying depth is implemented by processors which run out of work.

6 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 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) [43, 44]. 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 [45] and was recently rediscovered for the multicore architectures [28]. No implementation on the CELL processor has been reported so far.

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) and produces a different pivoting pattern, which is farther discussed in more detail. The tile LU algorithm relies on four basic operations implemented by four computational kernels (Figure 19).

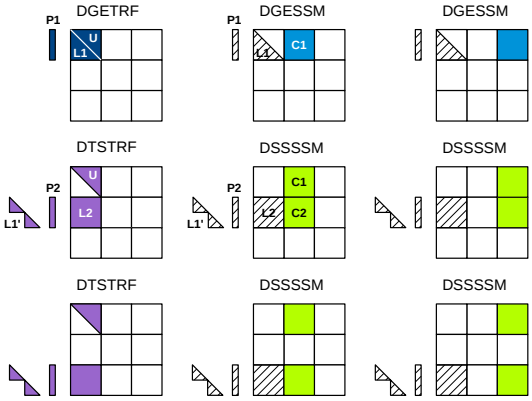

Figure 19: Tile operations in the tile LU factorization with inner blocking.

DGETRF: The kernel performs the LU factorization of a diagonal tile of the input matrix 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 build 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 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.

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 results in a less numerically stable algorithm. The numerical stability of the tile algorithm is not discussed here. As of today the authors are not aware of an exhaustive study of the topic.

As already mentioned earlier, due to pivoting, the lower triangular part of the diagonal block gets scrambled in consecutive steps of panel factorization. Each time this happens, the tiles to the right need to be updated, what introduces extra floating point operations, not accounted for in the standard formula for LU factorization. This is a similar situation to tile QR factorization, where the extra operations are caused by the accumulation of the Householder reflectors. For LU the impact is yet bigger, resulting in 50 % more operations for a naive implementation. The problem is remedied in the exact same way as for the tile QR factorization, by using the idea of inner blocking (Figure 19).

Another issue that comes into play is the concept of *LAPACK-style* pivoting versus *LINPACK-style* pivoting. In the former case, factorization of the panel is followed by row swaps both to the right of the panel and to the left of the panel. When using the factorization to solve the system, first permutations are applied to the entire right hand side vector, and then straightforward lower triangular solve is applied to perform the forward substitution. In the latter case, factorization of the panel is followed by row swaps only to the right of the panel (only to the trailing submatrix). As a result, in the forward substitution phase of solving the system, applications of pivoting and Gauss transforms are interleaved.

The tile algorithm combines LAPACK pivoting within the panel, to achieve high performance for the kernels on a cache-based system, and LINPACK pivoting between the steps of the factorization, to facilitate flexible scheduling of tile operations. The combination of the two pivoting techniques is explained in great detail by Quintana-Ortí and van de Geijn [45].

6.1 Parallel Implementation

The tile LU factorization is represented by a DAG of the exact same structure as the one for QR factorization. In other words, the tile LU factorization is identical, in terms of parallel scheduling, to the tile QR factorization. For that reason, the parallel implementations of the tile LU factorization are virtually identical to the parallel implementation of the tile QR factorization and all the facts presented in section 5 hold here. In the codes on Figures 11, 13, 14, 15, 16, the DGEQRT operation is replaced by the DGETRF operation, DLARFB operation by DGESSM operation, DTSQRT by DTSTRF and DSSRFB by DSSSSM.

7 Results and Discussion

Results were collected on a 2.4 GHz quad-socket quad-core (16 cores total) Intel Tigerton system running Linux kernel 2.6.18. Cilk and SMPSs codes where built using Cilk 5.4.6, SMPSs 2.0 and GCC 4.1.2. Static pipeline codes where built using ICC 10.1. Kernels coded in FORTRAN where compiled using IFORT 10.1. All codes where linked with MKL 10.0.1. Random input matrices where used (diagonally dominant for Cholesky factorization). Block Data Layout was used in all cases. Memory was allocated using huge TLB pages of size 2 MB.

Figure 20 shows execution traces of all the implementations of Cholesky factorization. The figure shows a small run (9×9 tiles, 1080×1080 elements) on a small number of cores (four). The goal here is

to clearly illustrate differences in scheduling by the different approaches.

The *Cilk 1D* implementation performs the worst. The 1D partitioning of work causes a disastrous load imbalance in each step of the factorization. Despite the lookahead, panel execution is very poorly overlapped with the update, in part due to the triangular shape of the updated submatrix and quickly diminishing amount of work in the update phase.

The *Cilk 2D* implementation performs much better by scheduling the *dtrsm()* operations in the panel in parallel. Also, scheduling the *dsyrk()* and *dgemm()* tasks in the update phase without constraints minimizes load imbalance. The only serial task, *dpotrf()*, does not cause disastrous performance losses.

Far better is the SMPSs implementation, where tasks are continuously scheduled without gaps until the very end of the factorization, where small stalls occur. Data reuse is clearly visible through clusters of *dsyrk()* tasks. Yet better is the *static pipeline* schedule, where no dependency stalls occur at all. and data reuse is exploited to the fullest.

Figure 21 shows execution traces of all the implementations of QR factorization. The same as for Cholesky, the figure shows a small run (7×7 tiles, 1008×1008 elements) on a small number of cores (four). Once again, the goal here is to clearly illustrate differences in scheduling by the different approaches. Traces for the tile LU factorization for a similar size problem are virtually identical to the QR traces and are not shown here. The following discussion applies equally to the tile QR and the tile LU factorization.

The situation looks a bit different for the tile QR and LU factorizations compared to the tile Cholesky factorization. The fine-grain *Cilk 2D* implementation performs poorest, which is mostly due to the dispatch of work in small batches. Although the tasks of panel factorization (*dgeqrt()*, *dtsqrt()*) are overlapped with the tasks of the update (*dlarfb()*, *dssrfb()*), synchronization after each row, and related load imbalance, contribute big number of gaps in the trace.

The *Cilk 1D* version performs better. Although the number of gaps is still significant, mostly due to 1D partitioning and related load imbalance, overall this implementation looses less time due to dependency stalls.

Interestingly the initial *SMPSs* implementation produces almost identical schedule to the *Cilk 1D* version. One difference is the better schedule at the end of the factorization. The overall performance difference is small.

The *SMPSs** implementation delivers a big jump in performance, due to dramatic improvement in the schedule. Here the *static pipeline* schedule is actually marginally worse than *SMPSs* due to a few more dependency stalls. More flexible scheduling of *SMPSs* provides for a better schedule at the end of the factorization. This advantage diminishes on larger number of cores, where the overheads of dynamic scheduling put the performance of the *SMPSs* implementation slightly behind the one of the *static pipeline* implementation.

Figure 22 shows performance for the Cholesky factorization, where Cilk implementations provide mediocre performance, SMPSs provides much better performance and *static pipeline* provides performance clearly superior to other implementations.

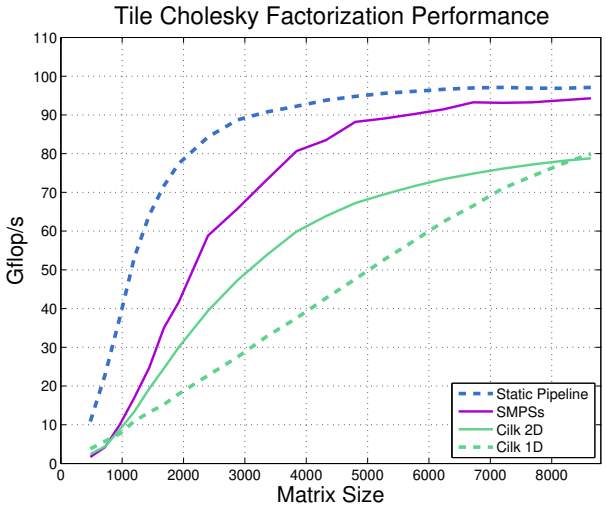

Figure 22: Performance of the tile Cholesky factorization in double precision on a 2.4 GHz quad-socket quad-core (16 cores total) Intel Tigerton system. Tile size $nb = 120$.

Figure 23 shows performance for the QR factorization. The situation is a little different here. Performance of Cilk implementations is still the poorest and the performance of the *static pipeline* is still superior. However, performance of the initial *SMPSs* implementation is only marginally better that *Cilk 1D*, while performance of the improved SMPSs* implementation is only marginally worse that *static pipeline*. The same conclusions apply to the tile LU factorization (Figure 24).

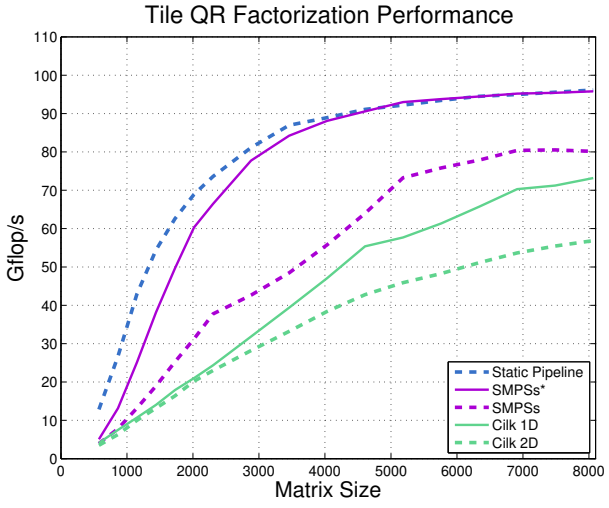

Figure 23: Performance of the tile QR factorization in double precision on a 2.4 GHz quad-socket quad-core (16 cores total) Intel Tigerton system. Tile size $nb = 144$, inner block size $IB = 48$.

Relatively better performance of SMPSs for the QR and LU factorizations versus the Cholesky factorization can be explained by the fact that the LU factorization is two times more expensive and the QR factorization is four times more expensive, in terms of floating point operations. This diminishes the impact of various overheads for smaller size problems.

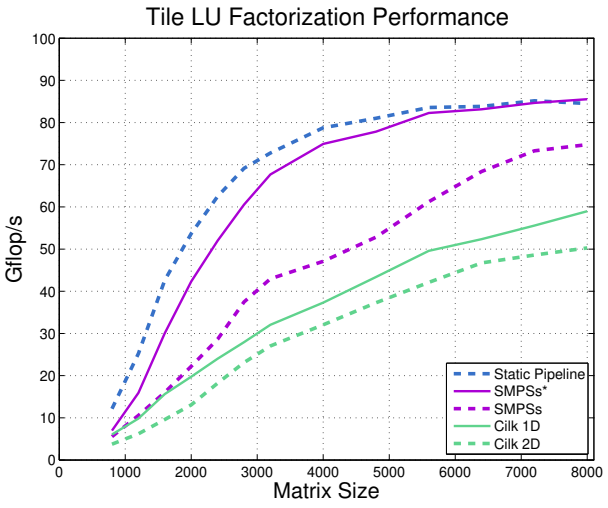

Figure 24: Performance of the tile LU factorization in double precision on a 2.4 GHz quad-socket quad-core (16 cores total) Intel Tigerton system. Tile size $nb = 200$, inner block size $IB = 40$.

8 Conclusions

In this work, suitability of emerging multicore programming frameworks was analyzed for implementing modern formulations of classic dense linear algebra algorithms, the tile Cholesky, the tile QR and the tile LU factorizations. These workloads are represented by large task graphs with compute-intensive tasks interconnected with a very dense and complex net of dependencies.

For the workloads under investigation, the conducted experiments show clear advantage of the model, where automatic parallelization is based on construction of *arbitrary* DAGs. SMPSs provides much higher level of automation than Cilk and similar frameworks, requiring only minimal programmer's intervention and basically leaving the programmer oblivious to any aspects of parallelization. At the same time it delivers superior performance through more flexible scheduling of operations.

SMPSs still looses to hand-written code for very regular compute-intensive workloads investigated

here. The gap is likely to decrease, however, with improved runtime implementations. Ultimately, it may have to be accepted as the price for automation.

9 Future Directions

Parallel programing based on the idea of representing the computation as a task graph and dynamic data-driven execution of tasks shows clear advantages for multicore processors and multi-socket shared-memory systems of such processors. One of the most interesting questions is the applicability of the model to large scale distributed-memory systems.

References

[1] 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/.

[2] 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/.

[3] Co-Array Fortran. http://www.co-array.org/.

[4] The Berkeley Unified Parallel C (UPC) project. http://upc.lbl.gov/.

[5] Titanium project home page. http://titanium.cs.berkeley.edu/.

[6] Cray, Inc. *Chapel Language Specification 0.775*. http://chapel.cs.washington.edu/spec-0.775.pdf.

[7] Sun Microsystems, Inc. *The Fortress Language Specification, Version 1.0*, 2008. http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf.

[8] V. Saraswat and N. Nystrom. *Report on the Experimental Language X10, Version 1.7*, 2008. http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Principles and Practice of Parallel Programming, Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPOPP'95*, pages 207–216, Santa Barbara, CA, July 19-21 1995. ACM. DOI: 10.1145/209936.209958.

[10] Intel Threading Building Blocks. http://www.threadingbuildingblocks.org/.

[11] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007. ISBN: 0596514808.

[12] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[13] The community of OpenMP users, researchers, tool developers and providers. http://www.compunity.org/.

[14] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A proposal for task parallelism in OpenMP. In *A Practical Programming Model for the Multi-Core Era, 3rd International Workshop on OpenMP, IWOMP 2007*, Beijing, China, June 3-7 2007. Lecture Notes in Computer Science 4935:1-12. DOI: 10.1007/978-3-540-69303-1_1.

[15] A. Duran, J. M. Perez, R. M. Ayguadé, E. amd Badia, and J. Labarta. Extending the OpenMP tasking model to allow dependent

tasks. In *OpenMP in a New Era of Parallelism, 4th International Workshop, IWOMP 2008*, West Lafayette, IN, May 12-14 2008. Lecture Notes in Computer Science 5004:111-122. DOI: 10.1007/978-3-540-79561-2_10.

[16] Barcelona Supercomputing Center. *SMP Superscalar (SMPSs) User's Manual, Version 2.0*, 2008. http://www.bsc.es/media/1002.pdf.

[17] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, 1998. http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf.

[18] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Tampa, Florida, November 11-17 2006. ACM. DOI: 10.1145/1188455.1188546.

[19] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM J. Res. & Dev.*, 51(5):593–604, 2007. DOI: 10.1147/rd.515.0593.

[20] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, 1995.

[21] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 207–218, Williamsburg, VA, January 1981. ACM. DOI: 10.1145/209936.209958.

[22] J. Kurzak and J. J. Dongarra. QR factorization for the CELL processor. *Scientific Programming*. (accepted).

[23] 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.

[24] R. E. Lord, J. S. Kowalik, and S. P. Kumar. Solving linear algebraic equations on an MIMD computer. *J. ACM*, 30(1):103–117, 1983. DOI: 10.1145/322358.322366.

[25] R. C. Agarwal and F. G. Gustavson. A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In *Proceedings of the IFIP WG 2.5 Working Conference on Aspects of Computation on Asynchronous Parallel Processors*, pages 217–221, Stanford, CA, August 22-25 1988. North-Holland Publishing Company. ISBN: 0444873104.

[26] R. C. Agarwal and F. G. Gustavson. Vector and parallel algorithms for Cholesky factorization on IBM 3090. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 225 – 233, Reno, NV, November 13-17 1989. ACM. DOI: 10.1145/76263.76287.

[27] 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.

[28] 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.

[29] J. Kurzak and J. J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. 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 4699:147-156. DOI: 10.1007/978-3-540-75755-9_18.

[30] A. Buttari, J. J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick. Multithreading for synchronization tolerance in matrix factorization. In *Scientific Discovery through Advanced Computing, SciDAC 2007*, Boston, MA, June 24-28 2007. Journal of Physics:

Conference Series 78:012028, IOP Publishing. DOI: 10.1088/1742-6596/78/1/012028.

[31] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal sparse Cholesky. *Future Gener. Comput. Syst.*, 20(3):425–440, 2004. DOI: 10.1016/j.future.2003.07.007.

[32] E. Elmroth and F. G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. & Dev.*, 44(4):605–624, 2000.

[33] E. Elmroth and F. G. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In *Applied Parallel Computing, Large Scale Scientific and Industrial Problems, 4th International Workshop, PARA'98*, Umeå, Sweden, June 14-17 1998. Lecture Notes in Computer Science 1541:120-128. DOI: 10.1007/BFb0095328.

[34] E. Elmroth and F. G. Gustavson. High-performance library software for QR factorization. In *Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, 5th International Workshop, PARA 2000*, Bergen, Norway, June 18-20 2000. Lecture Notes in Computer Science 1947:53-63. DOI: 10.1007/3-540-70734-4_9.

[35] 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.

[36] F. G. Gustavson. New generalized matrix data structures lead to a variety of high-performance algorithms. In *Proceedings of the IFIP WG 2.5 Working Conference on Software Architectures for Scientific Computing Applications*, pages 211–234, Ottawa, Canada, October 2-4 2000. Kluwer Academic Publishers. ISBN: 0792373391.

[37] 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.

[38] 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.

[39] N. Park, B. Hong, and V. K. Prasanna. Analysis of memory hierarchy performance of block data layout. In *Proceedings of the 2002 International Conference on Parallel Processing, ICPP'02*, pages 35–44, Vancouver, Canada, August 18-21 2002. IEEE Computer Society. DOI: 10.1109/ICPP.2002.1040857.

[40] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):640–654, 2003. DOI: 10.1109/TPDS.2003.1214317.

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

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

[43] 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.

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

[45] 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.

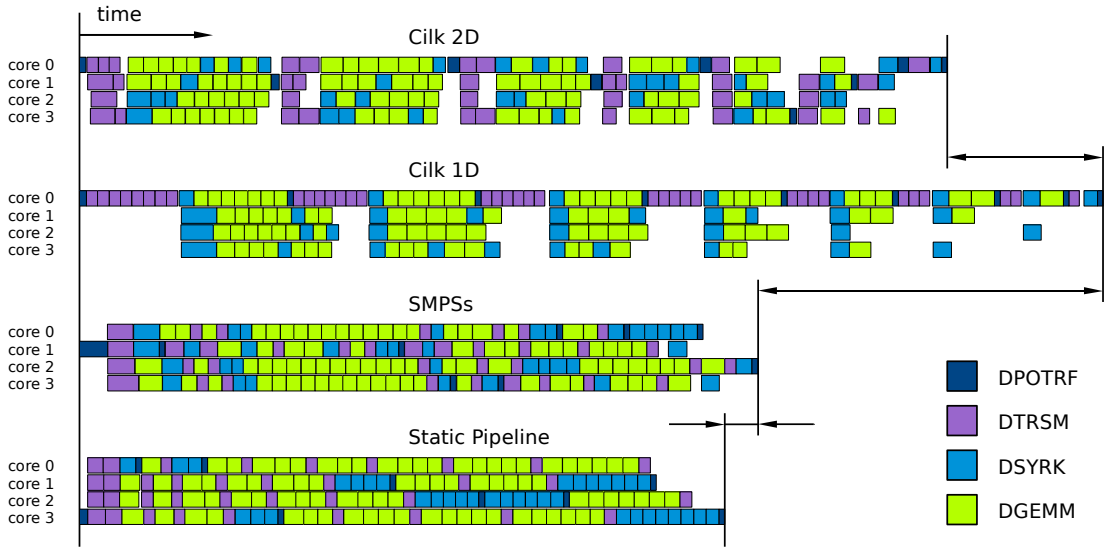

Figure 20: Execution traces of the tile Cholesky factorization in double precision on four cores of a 2.4 GHz Intel Tigerton system. Matrix size $N = 1080$, tile size $nb = 120$, total number of tasks = 140.

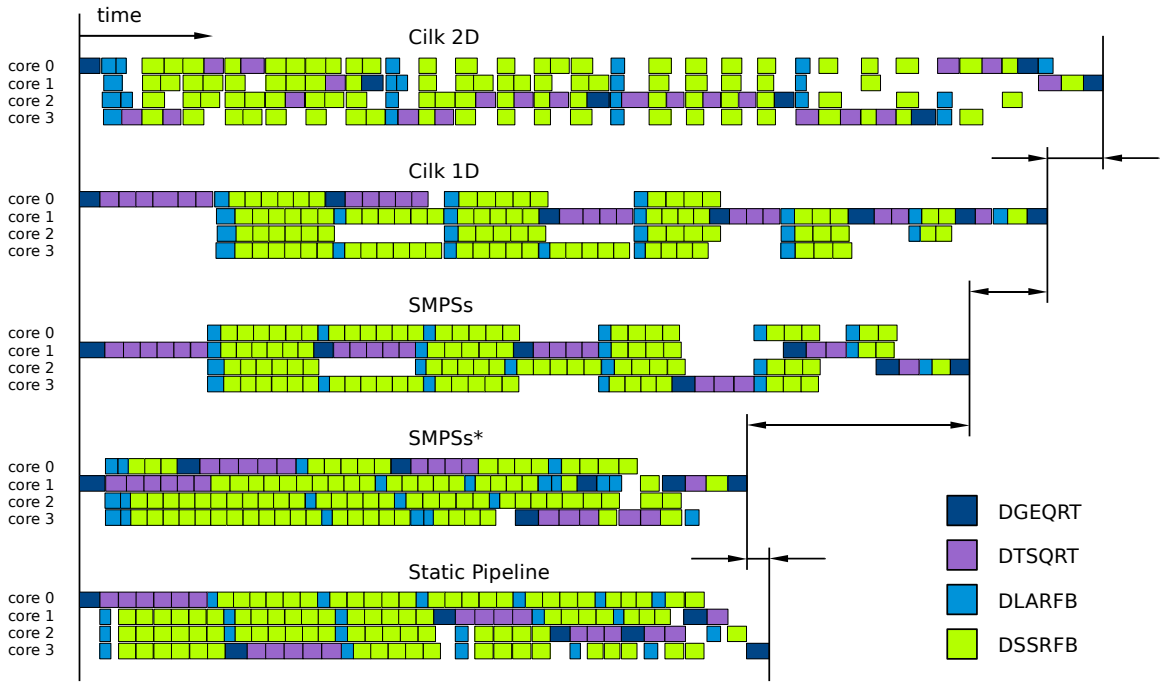

Figure 21: Execution traces of the tile QR factorization in double precision on four cores of a 2.4 GHz Intel Tigerton system. Matrix size $N = 1008$, tile size $nb = 144$, inner block size $IB = 48$, total number of tasks = 140.