Fully Dynamic Scheduler for Numerical Computing on Multicore Processors

– LAPACK Working Note 220

Jakub Kurzak
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

ABSTRACT

The dataflow model is gaining popularity as a paradigm for programming multicore processors and multi-socket systems of such processors. This work proposes a programming interface and an implementation for a dataflow-based scheduler, which dispatches tasks dynamically at runtime. The scheduler relies on data dependency analysis between tasks in a sequential representation of an algorithm, which provides productivity and facilitates rapid prototyping for developers. Also, through the application of dataflow principles, it ensures efficient scheduling with provisions for data reuse. Although designed with generality in mind, the scheduler is mainly intended for handling computational tasks in dense linear algebra in the PLASMA framework. Some customizations specific to that area are presented. Similarities and differences with other existing solutions are also discussed and preliminary performance results are given.

KEYWORDS: task graph, dataflow, multicore, dense linear algebra

Contents

1 Introduction & Motivation **2**

2 Existing Frameworks **2**
2.1 Cilk 2
2.2 SMPSs 3
2.3 PLASMA's Static Schedule 4

3 Scheduler API **4**
3.1 Creating a Task 4
3.2 Invoking a Task 4

4 Scheduler Implementation **5**
4.1 Types of Data Hazards 6
4.2 Scheduler Inner Workings 6
4.2.1 Populating Task Pool 7
4.2.2 Executing Tasks from the Pool 7

5 Performance Results **7**

6 Conclusions **8**

7 Future Directions **8**

8 Acknowledgements **8**

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 is inadequate for the new architectures, and the 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 only a small fraction of peak performance on current multicore processors and multi-socket systems of multicore processors, mostly following *Symmetric Multi-Processor* (SMP) architecture. Although the benefits of dynamic data-driven scheduling were discovered in the early 80's [3], recently they are being rediscovered as tools for exploiting the performance of multicore processors.

The problem is twofold. Achieving good performance on emerging chip designs is a serious problem, calling for new algorithms and data structures. Reimplementing an 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 programmer a scarce resource and millions of lines of code might be needed.

In mainstream server and desktop computing that targets mainly shared memory systems, the well known *dataflow model* is rapidly gaining popularity. The computation is expressed as a *Direct Acyclic Graph* (DAG), with nodes representing computational tasks and edges representing data dependencies among them.

2 Existing Frameworks

The coarse-grain dataflow model is the main principle behind emerging multicore programming environments such as *Cilk/Cilk++* [4], Intel® *Threading Building Blocks* (TBB) [5, 6], Tasking in OpenMP 3.0 [7, 8, 9, 10] and *SMP Superscalar* (SMPSs) [11]. 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 some comments on the static scheduling technique, currently utilized by the PLASMA framework and implemented "manually" using POSIX threads. 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 fact that it is available in open-source).

2.1 Cilk

Cilk was developed at the MIT Laboratory for Computer Science starting in 1994 [4]. and 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 Cilk keywords are removed from its 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, and the 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 locks can, however, easily lead to a 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" [12]. In particular locks cannot be used to enforce parent-child dependencies between tasks.

Cilk is very well suited for expressing algorithms that easily render themselves to recursive formulation, e.g., *divide-and-conquer* algorithms. Since the 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.

It has been shown that Cilk is not capable of efficiently scheduling workloads in dense linear algebra [13], which is due to the fact that Cilk does not rely on data dependency analysis and is, basically, mainly suitable for algorithms rendering themselves to recursive formulations.

2.2 SMPSs

SMP Superscalar (SMPSs) [11] 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 [14, 15]. While Grid Superscalar and Cell Superscalar address parallel software development for Grid environments 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) then 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 from 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's 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 [16] and optimizing compilers [17].

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 nonexistent in the Cilk environment.

The scheduler presented here is identical to SMPSs in philosophy. The main difference is expression of parallel code through an API instead of language extensions, which eliminates the need for a specialized compiler. Another difference is introduction of specialized hinting mechanisms, allowing for closer tweaking of dense linear algebra workloads. Also, since the scheduler presented here is developed "from scratch," it most likely shares little with SMPSs in terms of implementation.

2.3 PLASMA's Static Schedule

The first version of the PLASMA library relies on static schedules, where each core's workload is predetermined and synchronization is enforced by a global progress table. These schedules proved to deliver the highest performance in comparison to Cilk, SMPSs, and other schedulers developed at the University of Tennessee [13, 18, 19]. Also, these schedules can easily be adapted for distributed memory implementations.

The static scheduling technique has two important shortcomings. First is the difficulty of development. It requires full understanding of the data dependencies in the algorithm, which is hard to acquire even by an experienced developer. The road to a functional schedule is often paved with deadlocks and forces the developer to repeatedly trace the execution until the point of the deadlock. The second shortcoming is the inability to schedule dynamic algorithms, where the complete task graph is not know beforehand. This is the common situation for eigenvalue algorithms, which are iterative in nature.

It can be noted that static schedules were also developed for two-sided factorizations, such as reduction to block Hessenberg or band bidiagonal form [20, 21, 22]. These factorizations are significantly more complex to schedule than one-sided factorizations, such as LU, Cholesky and QR, and the effort to build the static schedules was considerably higher. Section 4 describes our current fully dynamic scheduler.

3 Scheduler API

Defining programmer interactions with the scheduler through an API is an alternative to SMPSs' compiler-based approach. Both approaches have their strengths and weaknesses. While the latter is probably more intuitive, the former is more flexible, allowing for greater control over the program and its execution, which makes it more suitable for the development of numerical libraries. It is also easier to provide APIs to different languages and platform portability is less of an issue.

Following is the description of the current state of the API, subject to extensions / improvements in the near future. Parallelization relies on two steps: transforming functions into task definitions and transforming function calls into task queueing constructs. Figure 1 shows the first step and Figure 2 shows the second one. Figure 3 shows the implementation of the tile LU algorithm.

3.1 Creating a Task

Similarly to Cilk and SMPSs, functions implementing parallel tasks have to be side-effect free, which means they cannot use global variables, etc. In order to change a regular function call to a task definition, one needs to:

- declare the function with empty argument list,
- declare the arguments as local variables, and
- get their values by using a macro *unpack_args_X*, where X is the number of arguments.

The above is a solution "borrowed" from the PLASMA library, where it is used to pass arguments between the serial, master routines and the parallel, worker routines.

3.2 Invoking a Task

The second step is changing the function call into a task invocation, which puts the task in the task pool and returns immediately, leaving the task execution for later (when dependencies are met and the scheduler decides to run the task). In order to change a function call into a task invocation, one needs to:

```
void CORE_dgetrf(
    int M, int N, int IB,
    double *A,
    int LDA, int *IPIV)
{
    ...
}
```

```
void CORE_dgetrf()
{
    int M, N IB;
    double *A;
    int LDA, *IPIV;
    unpack_args_6(M, N, IB, A, LDA, IPIV);

    ...
}
```

Figure 1: Example of transforming a function into a task definition.

```
CORE_dgetrf(
    NB     ,
    NB     ,
    IB     ,
    A(k, k),
    NB     ,
    IPIV(k, k));
```

```
Insert_Task(CORE_dgetrf,
    &NB        , sizeof(int)          , VALUE ,
    &NB        , sizeof(int)          , VALUE ,
    &IB        , sizeof(int)          , VALUE ,
    A(k, k)    , NB*NB*sizeof(double), INOUT ,
    &NB        , sizeof(int)          , VALUE ,
    IPIV(k, k), NB*sizeof(int)       , OUTPUT,
    NULL);
```

Figure 2: Example of transforming a function call into a task invocation.

- replace the function call with a call to the *Insert_Task()* function,
- pass the task name (pointer) as the first parameter, and
- follow each original parameter with its size and direction.

Array arguments have to be followed by the size of memory they occupy in bytes and one of the following directions: *INPUT*, *OUTPUT* or *INOUT*. Scalar arguments have to be passed by reference, followed by the size of their datatype and *VALUE* in the place of direction. Although scalar arguments are passed by reference, passing of scalars has the *pass by value* semantics (a copy of each scalar argument is made at the time of call to the *Insert_Task()* function).

4 Scheduler Implementation

Currently the scheduler targets small-scale, multi-socket shared memory systems based on multicore processors. The main design principle behind the scheduler is implementation of the dataflow model, where scheduling is based on data dependencies between tasks in the task graph. The second principle is constrained use of resources with strict bounds on space and time complexity.

```
for (k = 0; k < BB; k++) {
    Insert_Task(CORE_dgetrf,
        A(k, k), NB*NB*sizeof(double), INOUT,
        IPIV(k, k), NB*sizeof(double), OUTPUT,

    for (n = k+1; n < BB; n++)
        Insert_Task(CORE_dgessm,
            IPIV(k, k), NB*sizeof(double), INPUT,
            A(k, k), NB*NB*sizeof(double), NODEP,
            A(k, n), NB*NB*sizeof(double), INOUT,

    for (m = k+1; m < BB; m++) {
        Insert_Task(CORE_dtstrf,
            A(k, k), NB*NB*sizeof(double), INOUT,
            A(m, k), NB*NB*sizeof(double), INOUT,
            L(m, k), NB*IB*sizeof(double), OUTPUT,
            IPIV(m, k), NB*sizeof(double), OUTPUT,

        for (m = k+1; m < BB; m++)
            Insert_Task(CORE_dssssm,
                A(k, n), NB*NB*sizeof(double), INOUT,
                A(m, n), NB*NB*sizeof(double), INOUT,
                L(m, k), NB*IB*sizeof(double), INPUT,
                A(m, k), NB*NB*sizeof(double), INPUT,
                IPIV(m, k), NB*sizeof(double), INPUT,
    }
}
```

Figure 3: Implementation of the tile LU algorithm (scalar arguments are skipped for clarity).

The dataflow model is implemented through analysis of data hazards discussed in the following subsection. The constrained use of resources is accomplished by exploration of the task graph via a *sliding window* (Figure 4).

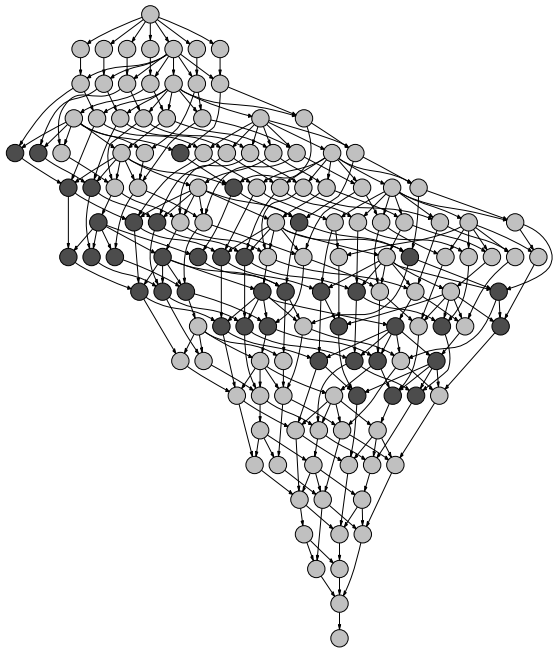

Figure 4: Task graph of a tile LU factorization of a 7×7 tiles matrix with dark nodes representing the sliding window of execution (104 total tasks, 40 tasks in the window).

Even relatively small problems in dense linear algebra (such that can be handled by a laptop or a desktop computer) can easily generate DAGs with hundreds of thousands or even millions of tasks. Generation and exploration of the entire DAG of such size would not be feasible. Instead, as execution proceeds, tasks are continuously generated and executed. However, at any given point in time only a relatively small number of tasks (on the order of one thousand) is stored in the task pool. The size of such a sliding window is a tunable parameter, allowing for trading the time and space overhead of scheduling for the quality of the schedule.

4.1 Types of Data Hazards

- **Read After Write (RAW)** hazard, often referred to as *true dependency* is the most common dependency. It defines the relation between a task writing ("creating") the data and the task reading ("consuming") the data afterwards, which has to wait until the former completes.
- **Write After Read (WAR)** hazard is caused by a situation where a task attempts to write (modify) data before a preceding task is finished reading the data. In such a case the writer has to wait until the reader completes. The dependency is not referred to as a true dependency, because it can be eliminated by renaming (making a copy) of the data. Although the dependency is unlikely to appear often in dense linear algebra, is has been encountered and had to be handled by the scheduler to ensure correctness.
- **Write After Write (WAW)** hazard is caused by a situation where a task attempts to write data before a preceding task is finished writing the data. The final result is expected to be the output of the latter task, but if the dependency is not preserved (and the former task completes after the latter one), incorrect output will result. This is an important dependency in hardware design of processor pipelines, where resource contention can be caused by a limited number of registers. The situation is, however, quite unlikely for a software scheduler, where the occurrence of the WAW hazard would mean that data is produced, which is never consumed, before it is overwritten. The same as the WAR hazard, the WAW hazard can be removed by renaming.

Although unlikely, the WAW hazard may be caused by a situation where partially overlapping regions are considered. Consider a situation where one task produces a large data set and a subsequent task replaces a small portion of that data set. Although the situation can be avoided, convenience of programming may dictate such a solution.

4.2 Scheduler Inner Workings

Figure 5 shows a simplified diagram of the scheduler implementation. The scheduler consists of a task pool, which is a static array of a fixed size and contains slots for tasks, being filled-in serially by one thread and emptied by all threads in parallel.

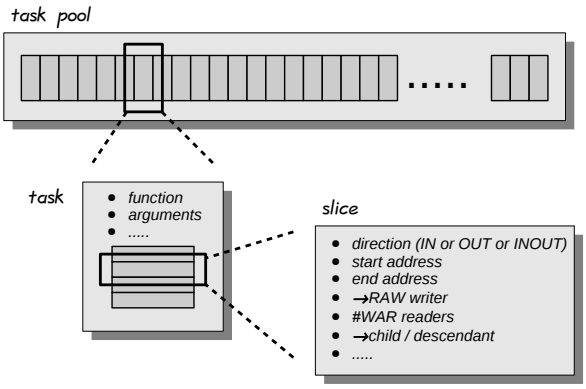

Figure 5: Simplified diagram of the scheduler implementation.

A task contains a unique task id, a function pointer and a copy of its call arguments (stored at the time of task invocation). It also contains a fixed number of *slices*. *Slice* is a code-word for a continuous region of memory being subject to dependency analysis.

Each slice is defined by its start address, end address and direction (INPUT, OUTPUT, INOUT). Associated with each slice is information necessary for dependency resolution, such as:

- RAW hazard writer id,
- RAW hazard writer pointer,
- WAR hazard reader's count, and
- WAR hazard writer's counter pointer.

Also, associated with each slice is a pointer to one descendant in the DAG, to assist with data reuse. The exact use of all this information is explained in more detail in the sections to follow.

4.2.1 Populating Task Pool

Since the user's code is expressed through a sequential definition, the process of populating the task pool is inherently sequential. The process can be replicated on multiple cores, but in principle it cannot be distributed to multiple cores for a performance advantage. For that reason, one thread takes care of the job of going through the user's code and populating the task pool. In principle, this thread can also participate in task execution (it is the case for SMPSs). However, the implementation presented here is somewhat suboptimal and one thread is solely devoted to creating tasks.

Currently, the process of inserting a task involves a scan through the entire task window in order to identify all dependencies. RAW hazards are identified by checking all INPUT and INOUT slices of the new task for overlap with all INOUT and OUTPUT slices of existing tasks. If overlap is detected, information about the writer task is stored with the reader's slice. WAR hazards are identified by checking all OUTPUT and INOUT slices of the new task for overlap with all INOUT and INPUT slices of existing tasks. If overlap is detected, the reader's slice stores the information about the writer and the writer increments its reader's counter.

4.2.2 Executing Tasks from the Pool

While one thread is inserting the tasks, a number of parallel worker threads continuously scan the task pool and pick tasks for execution. The process of fetching a task involves scanning the task pool in order to find a task without dependencies. A task is free of dependencies if all its slices are free of dependencies. An INPUT or INOUT slice if free of RAW dependency if its RAW writer does not exist in the task pool (the pointer is NULL, or points to a task with a different task id). An OUTPUT or INOUT slice is free of WAR dependency if its counter of WAR readers is zero. After executing, the reader task in the WAR hazard decrements the writer's counter of readers.

5 Performance Results

The tile LU algorithm was chosen for preliminary performance experiments. Figure 6 shows performance numbers for the scheduler versus SMPSs and the static schedule, utilized by the PLASMA library, on a 2.4 GHz quad-socket quad-core (16 cores total) Intel Tigerton system. A relatively large task granularity was chosen. The tile size of 200 was used with

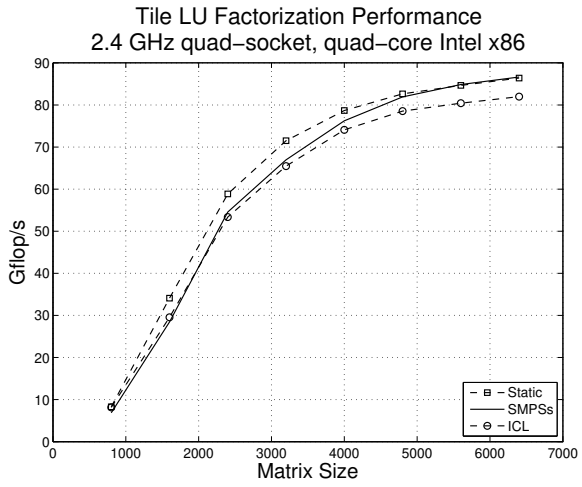

Figure 6: Performance comparison for the tile LU algorithm.

inner-blocking of 40. These are typical values for achieving good asymptotic performance. They do, however, render suboptimal performance for small problem sizes.

One can observe that, for this particular problem, SMPSs loses marginally to the static schedule for smaller problem sizes and eventually catches up for larger problems. The scheduler presented in this article performs as good as SMPSs for smaller problem sizes, but eventually drops below SMPSs' performance by a margin of roughly 6 %, which is due to one core devoted to inserting the tasks and not participating in executing them.

6 Conclusions

It has been demonstrated that with relatively small effort a data-driven scheduler can be built for efficient execution of non-trivial workloads in dense linear algebra on small-scale multicore systems. The main advantage of this scheduler is sequential representation of the algorithm, providing for very high productivity and allowing for rapid prototyping of new algorithms. Due to reliance on data dependency analysis, the scheduler has tremendous performance advantage over alternative solutions, such as Cilk, Intel TBB and OpenMP. The scheduler also has a great portability advantage by reliance on an API instead of language extensions.

Dynamic scheduling of code expressed through sequential representation has tremendous programmability advantage, but can also provide performance advantage in situations where optimal static schedules cannot be easily constructed. A common factor preventing the effectiveness of static schedules is fluctuation in task execution times caused by unpredictable behavior of the memory system, commonly consisting of multiple levels of caches with different bandwidths and latencies.

The ultimate limitation of the approach presented here is the serial nature of the process of unfolding the DAG from a serial definition of the algorithm. Due to that, the centralized implementation of the task pool is the most natural one and most likely it would be very hard to efficiently decentralize it.

7 Future Directions

Currently, the most important goal is further optimization of the scheduler in an attempt to close the gap between its performance and the performance of statically scheduled operations in the PLASMA library. One part of accomplishing this goal is the introduction of data structures for speeding up the scheduler's operations, such as hash tables for address lookups. Another part is the introduction of extensions allowing for further tweaking of the scheduler, such as the prioritization of a data path. Further down the line is the objective of extending the scheduler to small scale distributed memory systems. Using the scheduler for workloads beyond the field of dense linear algebra is also an interesting possibility.

8 Acknowledgements

The authors would like to thank Rosa Badia for help with understanding of inner workings of SMPSs and Julien Langou for comments on implementing extensions specific to problems in dense linear algebra.

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] 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.

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. LAPACK working note 213: Scheduling linear algebra operations on multicore processors. Technical Report UT-CS-09-636, Computer Science Department, University of Tennessee, 2009. http://www.netlib.org/lapack/lawnspdf/lawn213.pdf.

[14] 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.

[15] 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.

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

[17] 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.

[18] E. Agullo, B. Hadri, H. Ltaief, and J. J. Dongarra. LAPACK working note 217: Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. Technical Report UT-CS-09-640, Computer Science Department, University of Tennessee, 2009. http://www.netlib.org/lapack/lawnspdf/lawn217.pdf.

[19] F. Song, A. YarKhan, and J. J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. Technical Report UT-CS-09-638, Computer Science Department, University of Tennessee, 2009. http://www.cs.utk.edu/~library/TechReports/2009/ut-cs-09-638.pdf.

[20] H. Ltaief, J. Kurzak, and J. J. Dongarra. LAPACK working note 208: Parallel block Hessenberg reduction using algorithms-by-tiles for multicore architectures revisited. Technical Report UT-CS-09-624, Computer Science Department, University of Tennessee, 2009. http://www.netlib.org/lapack/lawnspdf/lawn208.pdf.

[21] H. Ltaief, J. Kurzak, and J. J. Dongarra. LAPACK working note 209: Parallel band two-sided matrix bidiagonalization for multicore architectures. Technical Report UT-CS-09-631, Computer Science Department, University of Tennessee, 2009. http://www.netlib.org/lapack/lawnspdf/lawn209.pdf.

[22] H. Ltaief, J. Kurzak, and J. J. Dongarra. LAPACK working note 214: Scheduling two-sided transformations using algorithms-by-tiles on multicore architectures. Technical Report UT-CS-09-637, Computer Science Department, University of Tennessee, 2009. http://www.netlib.org/lapack/lawnspdf/lawn214.pdf.