List of Figures

List of Tables

Contents

Chapter 1

Multithreading in the PLASMA Library

Jakub Kurzak
University of Tennessee, Knoxville

Piotr Luszczek
University of Tennessee, Knoxville

Asim YarKhan
University of Tennessee, Knoxville

Mathieu Faverge
University of Tennessee, Knoxville

Julien Langou
University of Colorado, Denver

Henricus Bouwmeester
University of Colorado, Denver

Jack Dongarra
University of Tennessee, Knoxville
Oak Ridge National Laboratory
University of Manchester

1.1 Introduction

Parallel Linear Algebra Software for Multicore Architectures (PLASMA) is a numerical software library for solving problems in dense linear algebra on systems of multicore processors and multi-socket systems of multicore processors [2]. PLASMA offers routines for solving a wide range of problems in dense linear algebra, such as: non-symmetric, symmetric and symmetric positive definite systems of linear equations, least square problems, singular value problems and eigenvalue problems (currently only symmetric eigenvalue problems). PLASMA solves these problems in real and complex arithmetic and in single and double precision. PLASMA is designed to give high efficiency on homogeneous multicore processors and multi-socket systems of multicore processors. As of today, the majority of such systems are on-chip symmetric multiprocessors with classic *super-scalar* processors as their building blocks (x86 and alike) augmented with short-vector SIMD extensions (SSE and alike). PLASMA has been designed to supercede LAPACK [6], principally by restructuring the software to achieve much greater efficiency on modern computers based on multicore processors.

The interesting part of PLASMA from the mutithreading perspective is the variety of scheduling mechanism utilized by PLASMA. In the next subsection the main desing principles of PLASMA are introduced. Section 1.2 discusses the different multithreading mechanisms employed by PLASMA. Section 1.3 introduces PLASMA's most powerfull mechanism of dynamic runtime task scheduling with the QUARK scheduler and briefly iterates through QUARK's extensions essential to the implementation of a production-quality numerical software. Section 1.4 highlights the advantages of dynamic scheduling for parallel task composition by showing how PLASMA implements explicit matrix inversion using the Cholesky factorization. Section 1.5 covers the concept of task aggregation when large clusters of tasks are offloaded to a GPU accelerator. Finally, section 1.6 discusses the very important case of using QUARK for exploiting nested parallelism.

1.1.1 PLASMA Desing Principles

The main motivation behind the PLASMA project are performance shortcomings of LAPACK [6] and ScaLAPACK [11] on shared memory systems, specifically systems consisting of multiple sockets of multicore processors. The three crucial elements that allow PLASMA to achieve performance greatly exceeding that of LAPACK and ScaLAPACK are: the implementation of *tile algorithms*, the application of *tile data layout* and the use of *dynamic scheduling*. Although some performance benefits can be delivered by each one of these techniques on its own, it is only the combination of all of them that delivers maximum performance and highest hardware utilization.

Tile algorithms are based on the idea of processing the matrix by square tiles of relatively small size, such that a tile fits entirely in one of the cache levels associated with one core. This way a tile can be loaded to the cache and processed completely before being evicted back to the main memory. Of the three types of cache misses, *compulsory*, *capacity* and *conflict*, the use of tile algorithms minimizes the number of capacity misses, since each operation loads the amount of data that does not "overflow" the cache.

Tile layout is based on the idea of storing the matrix by square tiles of relatively small size, such that each tile occupies a continuous memory region. This way a tile can be loaded to the cache memory efficiently and the risk of evicting it from the cache memory before it is completely processed is minimized. Of the three types of cache misses, *compulsory*, *capacity* and *conflict*, the use of tile layout minimizes the number of conflict misses, since a continuous region of memory will completely fill out a set-associative cache memory before an eviction can happen. Also, from the standpoint of multithreaded execution, the probability of *false sharing* is minimized. It can only affect the cache lines containing the beginning and the ending of a tile.

Dynamic scheduling is the idea of assigning work to cores based on the availability of data for processing at any given point in time and is also referred to as *data-driven* scheduling. The concept is related closely to the idea of expressing computation through a task graph, often referred to as the DAG (*Direct Acyclic Graph*), and the flexibility of exploring the DAG at runtime. Thus, to a large extent, dynamic scheduling is synonymous with *runtime scheduling*. An important concept here is the one of the *critical path*, which defines the upper bound on the achievable parallelism and needs to be pursued at the maximum speed. This is in direct opposition to the *fork-and-join* or *data-parallel* programming models, where artificial synchronization points expose serial sections of the code, where multiple cores are idle, while sequential processing takes place.

1.1.2 PLASMA Software Stack

Starting from the PLASMA, Version 2.2, released in July 2010, the library is built on top of standard software components, all of which are either available as open source or are standard OS facilities. Some of them can be replaced by packages provided by hardware vendors for efficiency reasons. Figure 1.1 presents the current structure of PLASMA's software stack. Following is a brief bottom-up description of individual components.

Basic Linear Algebra Subprograms (BLAS) [10] is a, *de facto* standard, set of basic linear algebra operations, such as vector and matrix multiplication. CBLAS is the C language interface to BLAS [14]. Most commercial and academic implementations of BLAS also provide CBLAS. *Linear Algebra PACKage* (LAPACK) [6] is a software library for numerical linear algebra, a direct predecessor of PLASMA, providing routines for solving linear systems of equations, linear least square problems, eigenvalue problems and singular

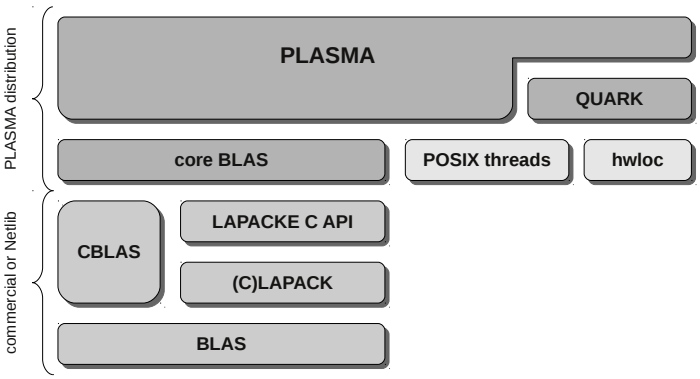

FIGURE 1.1: The software stack of PLASMA, Version 2.3.

value problems. CLAPACK [15] is a version of LAPACK available from Netlib, created by automatically translating FORTRAN LAPACK to C with the help of the F2C [16] utility. It provides the same, FORTRAN, calling convention as the "original" LAPACK. LAPACKE C API **??** is a proper C language interface to LAPACK (or CLAPACK).

"core BLAS" is a set of serial kernels, the building blocks for PLASMA's parallel algorithms. PLASMA scheduling mechanisms coordinate the execution of these kernels in parallel on multiple cores. PLASMA relies on POSIX threads for access to the systems multithreading capabilities and on the hwloc **??** library for the control of thread affinity. PLASMA employs *static scheduling*, where threads have their work statically assigned and coordinate execution through progress tables, but can also rely on the QUARK [25] scheduler for dynamic (runtime) scheduling of work to threads.

1.1.3 PLASMA Scheduling

By now, multicore processors are ubiquitous in both low-end consumer electornics and high-end servers and supercomputer installations. This led to the emergence of a myriad of multithreading frameworks, both academic and commercial, embracing the idea of task scheduling: Cilk [12], OpenMP (tasking features) [22], Intel Threading Building Blocks [24], just to name a few prominent examples. One expecially important category are multithreading systems based on dataflow principles, which represent the computation as a *Direct Acyctlic Graph* (DAG) and schedule tasks at runtime through resolution of data hazards: *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). PLASMA's scheduler QUARK is an example of such a system. Two other, very similar, academic projects are also available: StarSs [9] from Barcelona Supercomputer Center and StarPU [8] fron IN-

RIA Bordeaux. While all three systems have their strength and weaknesses, QUARK has vital extensions for use in a numerical library.

1.2 Multithreading in PLASMA

PLASMA has to be intialized with the call to the `PLASMA_Init` function, before any calls to its computational routines can be made. When the user's (master) thread calls the `PLASMA_Init` function and specifies N as the number of cores to use, PLASMA launches $N-1$ additional (worker) threads and puts them in a waiting state. In the master thread, control returns to the user. When the user calls PLASMA's computational routine, the worker threads are woken up and all the threads, including the user's master thread, enter the routine. Upon completion, worker threads return back to the waiting state, while the master thread returns from the call. A call to the `PLASMA_Finalize` function terminates the worker threads.

PLASMA is *thread-safe*. Multiple PLASMA instances, referred to as *contexts* can be active at the same time without conflicts, with the restriction that one user thread can be associated with one context only, i.e., one thread can only call `PLASMA_Init` once, before it calls `PLASMA_Finalize`. (Currently contexts are managed implicitly. Context handle is not provided to the user.) The typical usage scenario is to have a serial code (with a single thread) and launch one instance of PLASMA to use all the cores in the system. However, it is also possible for the user to spawn, e.g., four threads, each of which creates a four-thread instance of PLASMA, in order to exploit 16-way parallelism. In such case, each PLASMA instance synchronizes its four threads, while the user has to manage synchronization among the four PLASMA instances. PLASMA provides mechanisms for managing *thread affinity*, i.e. controlling the placement of threads on the physical cores.

PLASMA's statically scheduled routines follow the *Single Program Multiple Data* (SPMD) programming paradigm. Each thread knows its thread ID (`PLASMA_RANK`) and the total number of threads within the context (`PLASMA_SIZE`) and follows a specific execution path based on that information. Synchronization is implemented though shared progress tables and busy waiting. If a thread cannot progress, it yields the core to the OS (`sched_yield`). PLASMA's dynamically scheduled routines follow the *superscalar* programming paradigm, where the code is written sequentially and parallelized at runtime through the analysis of the dataflow between different tasks. When a static routine is called, all the threads simply enter the SPMD code of the routine. When a dynamic routine is called, the master thread enters the superscalar routine and begins queueing tasks using QUARK's task queueing calls. At the same time all the worker threads enter QUARK's *worker loop*, where they keep executing the queued tasks until notified by the master.

The master also participates in the execution of tasks, unless overwhelmed with the job of queueing.

PLASMA's computational routines are implemented using either static or dynamic scheduling or both. PLASMA provides a switch, which can be changed at runtime, to decide if static or dynamic scheduling is preferred, for routines with both implementations. If a sequence of user's calls invokes a mixed set of routines with static implementations only and dynamic implementation only, PLASMA switches the scheduling mode on the flight. Statically scheduled rouitnes are separated with barriers and each switch from static scheduling to dynamic scheduling, and vice versa, invokes a barrier. However, any continuous sequence of dynamically scheduled routines is free of barriers.

1.3 Dynamic Scheduling with QUARK

The QUARK scheduler targets multicore, multi-socket shared memory systems. The main design principle behind QUARK 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 bounds on space and time complexity. The dataflow model is implemented through analysis of data hazards, discussed in the following paragraphs. The constrained use of resources is accomplished by exploration of the task graph by a *sliding 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. 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 the sliding window is a tunable parameter, allowing for trading the time and space overhead for scheduling flexibility.

Parallelization using QUARK relies on two steps: transforming function calls to task definitions and replacing function calls with task queueing constructs. Figure 1.2 shows how PLASMA defines QUARK *dgemm* (matrix multiply) task using a call to CBLAS. Similarly to Cilk and SMPSs, functions implementing parallel tasks must be side-effect free (cannot use global variables, etc.) The task definition takes QUARK handle as its only parameter, declares all arguments as local variables, fetches them from QUARK using the `quark_unpack_args_` construct and executes the work (which in this example is just calling the `cblas_dgemm` function).

In order to change a function call to a task invocation, one needs to: replace the function call with a call to `QUARK_Insert_Task()`, pass the task

```
void CORE_dgemm_quark(Quark *quark) {
    int transA, transB;
    int m, n, k;
    double alpha, beta;
    double *A, *B, *C;
    int lda, ldb, ldc;

    quark_unpack_args_13(quark, transA, transB, m, n, k,
                         alpha, A, lda, B, ldb, beta, C, ldc);
    cblas_dgemm( CblasColMajor,
                (CBLAS_TRANSPOSE)transA, (CBLAS_TRANSPOSE)transB,
                 m, n, k, alpha, A, lda, B, ldb, beta, C, ldc);
}
```

FIGURE 1.2: QUARK dgemm (matrix multiply) task definition in PLASMA.

name (pointer) as the first parameter, preceede each parameter with its size and follow with direction. Array (pointer) arguments are preceeded with the size of memory they occupy, in bytes, and followed with one of the directions: `INPUT`, `OUTPUT` or `INOUT`. Scalar arguments are passed by a reference (pointer), preceeded with the size of their datatype, and followed by `VALUE` in place of the 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 `Insert_Task()`.)

```
QUARK_Insert_Task(quark, CORE_dgemm_quark, task_flags,
    sizeof(PLASMA_enum),    &transA,    VALUE,
    sizeof(PLASMA_enum),    &transB,    VALUE,
    sizeof(int),            &m,         VALUE,
    sizeof(int),            &n,         VALUE,
    sizeof(int),            &k,         VALUE,
    sizeof(double),         &alpha,     VALUE,
    sizeof(double)*nb*nb,    A,                 INPUT,
    sizeof(int),            &lda,       VALUE,
    sizeof(double)*nb*nb,    B,                 INPUT,
    sizeof(int),            &ldb,       VALUE,
    sizeof(double),         &beta,      VALUE,
    sizeof(double)*nb*nb,    C,                 INOUT,
    sizeof(int),            &ldc,       VALUE,
    0);
```

FIGURE 1.3: QUARK dgemm (matrix multiply) task invocation (queueing) in PLASMA.

QUARK schedules tasks at runtime through the resolution of data hazards (dependencies): *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). The *Read After Write* (RaW) hazard, often referred to as the *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. In that case the latter task has to wait until the former task completes.

The *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 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 has to be handled by the scheduler to ensure correctness.

The *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 means that some data is produced and overwritten before it is consumed. The same as the WaR hazard, the WaW hazard can be removed by renaming.

1.4 Parallel Composition

One vital feature of QUARK, or any other superscalar scheduler, is the *parallel composition*, i.e., the ability to construct larger task graphs from a set of smaller task graphs. The benefit is exposing more parallelism in the combined task graph then each of the components posseses alone. PLASMA's routine for computing an inverse of a symmetric positive definite matrix is a great example [1].

The appropriate direct method to compute the solution of a symmetric positive definite system of linear equations consists of computing the Cholesky factorization of that matrix and then solving the underlying triangular systems. It is not recommended to use the inverse of a matrix in this case. However, some applications need to explicitly form the inverse of the matrix. A canonical example is the computation of the variance-covariance matrix in statistics. Higham [19, p.260,§3] lists more such applications.

The matrix inversion presented here follows closely the one in LAPACK and ScaLAPACK. The inversion is performed in-place, i.e., the data structure initially containing matrix A is gradually overwritten with the result and eventually A^{-1} replaces A. (No extra storage is used.) The algorithm involves three steps: computing the Cholesky factorization ($A = LL^T$), inverting the L factor (computing L^{-1}) and, finally, computing the inverse matrix $A^{-1} = {L^{-1}}^T L^{-1}$. In LAPACK the three steps are performed by the functions: POTRF, TRTRI and LAUUM. In PLASMA the steps are performed by functions which process

```
void plasma_pdpotrf_quark(...) {
    for (k = 0; k < M; k++) {
        QUARK_CORE_dpotrf(...);
        for (m = k+1; m < M; m++) {
            QUARK_CORE_dtrsm(...);
        }
        for (m = k+1; m < M; m++) {
            QUARK_CORE_dsyrk(...);
            for (n = k+1; n < m; n++) {
                QUARK_CORE_dgemm(...);
            }
        }
    }
}
void plasma_pdtrtri_quark(...) {
    for (n = 0; n < N; n++) {
        for (m = n+1; m < M; m++) {
            QUARK_CORE_dtrsm(...);
        }
        for (m = n+1; m < M; m++) {
            for (k = 0; k < n; k++) {
                QUARK_CORE_dgemm(...);
            }
        }
        for (m = 0; m < n; m++) {
            QUARK_CORE_dtrsm(...
        }
        QUARK_CORE_dtrtri(...)
    }
}
void plasma_pdlauum_quark(...) {
    for (m = 0; m < M; m++) {
        for(n = 0; n < m; n++) {
            QUARK_CORE_dsyrk(...);
            for(k = n+1; k < m; k++) {
                QUARK_CORE_dgemm(...);
            }
        }
        for (n = 0; n < m; n++) {
            QUARK_CORE_dtrmm(...);
        }
        QUARK_CORE_dlauum(...);
    }
}
```

FIGURE 1.4: PLASMA functions, POTRF, TRTRI, LAUUM, computing the inverse of a symmetric positive definite matrix using QUARK for superscalar parallelization.

the matrix by tiles define the work in terms of tile operations (tile kernels), and use QUARK to queue, schedule and execute the kernels. Figure 1.4 shows an excerpt of PLASMA implementation of the three functions. Each one includes four loops, the first one with three levels of nesting, the other two with two levels of nesting. All work in expressed through elementary BLAS operations encapsulated in `core_blas` kernels: POTRF, TRSM, SYRK, GEMM, TRTRI, TRMM, LAUUM.

The scheduler addresses three important problems in parallel software development: complexity, productivity and performance. A superscalar scheduler

eliminates the complexity of writing parallel software, by automatically parallelizing algorithms defined sequentially and guaranteeing parallel correctness of sequentially expressed algorithms. For some workloads in dense linear algebra manual parallelization is relatively straightforward. A good example here is the Cholesky factorization (when considered alone) and its statically scheduled implementation in PLASMA [20]. For other operations it becomes nontrivial. Designing a static schedule for the combined three operations of the matrix inversion would be much harder. Finally, for some operations it becomes prohibitively complex. A good example here are PLASMA routines for band reductions through bulge chasing [21].

Another benefit of the scheduler is productivity. Thanks to the fact that sequential correctness guarantees parallel correctness, the scheduler facilitates very rapid development of numerical software, where manual design of parallel codes is labor intensive and error prone, causing long development times. Yet another benefit is the ability to do rapid prototyping of new algorithms and analysis of their parallel performance without the tedious work of parallelization. Finally, the scheduler also provides for increased performance, by identifying parallel scheduling opportunities where a human programmer would miss them. Also, it is resilient to fluctuations in task execution time, OS jitter and adverse effects of resource sharing. (Such adverse effects will cause a graceful degradation rather than a catastrophic performance loss).

Figures 1.5 and 1.6 further strengthen those points. Figures 1.5 shows the three DAGs of the three components of the matrix inversion and the aggregate DAG of the entire inversion. The DAG aggregation is a natural behavior of QUARK and happens automatically upon invocation of the three routines on Figure 1.4. The DAG created by stacking the DAGs of the three components on top of each other is taller and thinner, which means its sequential part (*the critical path*) is longer and its parallelism is more confined. The aggregate DAG is shorter and wider, meaning shorter critical path and more parallelism.

Figure 1.6 shows execution traces of the matrix inversion with and without barriers between phases. The parallelism of each separate phase is limited by its data dependencies, causing gaps in execution. However, when the barriers are removed the three phases fill out each others gaps, what results in a shorter overall execution time. Figure 1.7 shows the difference in performance when using a large system with 48 cores.

1.5 Task Aggregation

Task aggregation is a feature of QUARK allowing for creation of large tasks representing agglomerates of smaller tasks. These tasks not only have a large number of "inherited" dependencies, but also the number of dependencies is not known at compile time, and determined at runtime instead. This feature

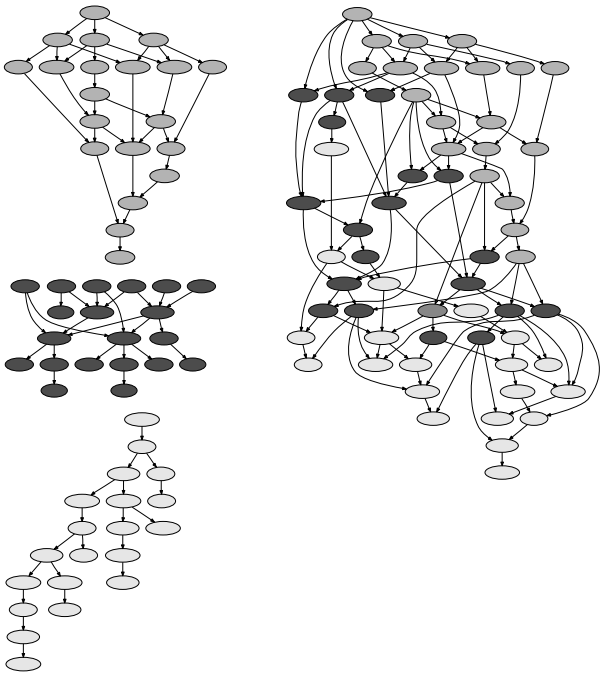

FIGURE 1.5: Three separate DAGs for the components of matrix inversion and their combined DAG.

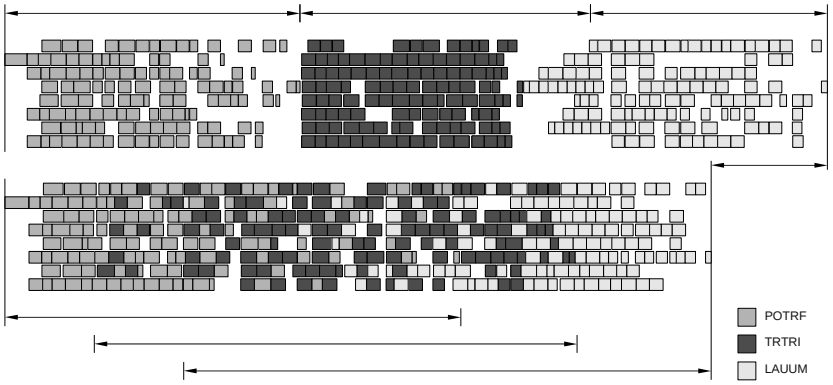

FIGURE 1.6: Trace of a small matrix inversion run on eight cores, with and without barriers between phases.

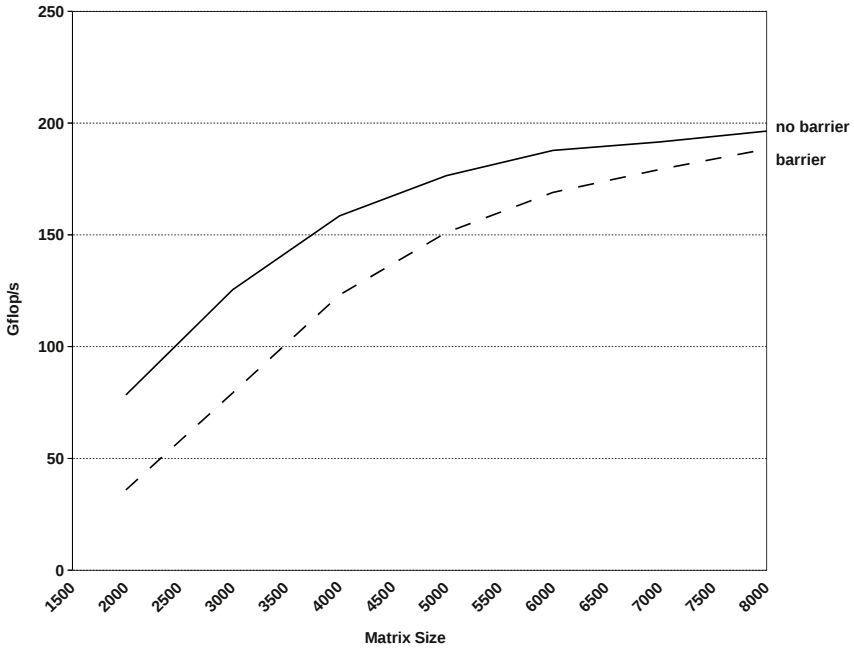

FIGURE 1.7: Performance difference between matrix inversion codes with and without barriers on a 48-core AMD-bases system.

has two main applications. One is to mitigate scheduling overheads of very fine granularity tasks, by increasing the level of granularity. Another is to offload work to data-parallel devices, such as GPUs. Here the latter case will serve as an example.

The main problem of existing approaches to accelerating dense linear algebra using GPUs is that GPUs are used like monolithic devices, i.e., like another "core" in the system. The massive disproportion of computing power between the GPU and the standard cores creates problems in work scheduling and load balancing. As an alternative, the GPU can be treated as a set of cores, each of which can efficiently handle work at the same granularity as a standard CPU core. The difficulty here comes from the fact that GPU cores cannot synchronize work in any other way than through a global barrier. In other words, the tasks offloaded to the GPU have to be independent.

The crucial concept here is the one of task aggregation. The GPU kernel is an aggregate of many CPU kernels, i.e., one call to the GPU kernel replaces many calls to CPU kernels. Because of the data-parallel nature of the GPU, the CPU calls constituing the aggregate GPU call cannot have dependencies among them. The GPU task call can be pictured in the DAG as a cluster of CPU tasks with arrows coming into the cluster and out of the cluster,

```
for (k = 0; k < SIZE; k++) {
    QUARK_Insert_Task(CORE_sgeqrt, ...);
    for (m = k+1; m < SIZE; m++)
        QUARK_Insert_Task(CORE_stsqrt, ...);
    for (n = k+1; n < SIZE; n++)
        QUARK_Insert_Task(CORE_sormqr, ...);
    for (m = k+1; m < SIZE; m++)
        for (n = k+1; n < SIZE; n++)
            QUARK_Insert_Task(CORE_stsmqr, ...);
}
```

FIGURE 1.8: QUARK code for the QR factorization using CPU cores only.

but no arrows connecting the task inside the cluster. In order to support this model, QUARK allows for queueing of tasks with a dynamic range of dependencies. Initially, a task with no dependencies is created through a call to `QUARK_Task_Init`. Than any number of dependencies can be added to the task using calls to `QUARK_Task_Pack_Arg`. Finally, the task can be queued with a call to `QUARK_Insert_Task_Packed`.

The tile QR factorization will serve as an example here. Figure 1.8 shows a QUARK implementation of the tile QR factorization using CPUs only. This particular example consists of five loops with three levels of nesting. It is built out of four `core_blas` kernels: GEQRT, TSQRT, ORMQR and TSMQR. More details can be found in PLASMA literature. Figure 1.9 shows modifications necessary to offload some of the tasks to the GPU. Here the last loop nest is split into the CPUs part and the GPU part. The split is done along the n dimension. While the CPUs get *lookahead* columns of the matrix to process, the GPU gets $SIZE - lookahead$ columns to process. The `cuda_stsmqr` kernel

```
for (k = 0; k < SIZE; k++) {
    QUARK_Insert_Task(CORE_sgeqrt, ...);

    for (m = k+1; m < SIZE; m++)
        QUARK_Insert_Task(CORE_stsqrt, ...);
    for (n = k+1; n < SIZE; n++)
        QUARK_Insert_Task(CORE_sormqr, ...);
    for (m = k+1; m < SIZE; m++)
        for (n = k+1; n < k+1+lookahead; n++)
            QUARK_Insert_Task(CORE_stsmqr, ...);
    task = QUARK_Task_Init(cuda_stsmqr, ...);
    for (m = k+1; m < SIZE; m++)
        for (n = k+1+lookahead; n < SIZE; n++) {
            QUARK_Task_Pack_Arg(task, &C1, INOUT);
            QUARK_Task_Pack_Arg(task, &C2, INOUT);
            QUARK_Task_Pack_Arg(task, &V2, INPUT);
            QUARK_Task_Pack_Arg(task, &T, INPUT);
        }
    QUARK_Insert_Task_Packed(task);
}
```

FIGURE 1.9: QUARK code for the QR factorization using CPUs and a GPU.

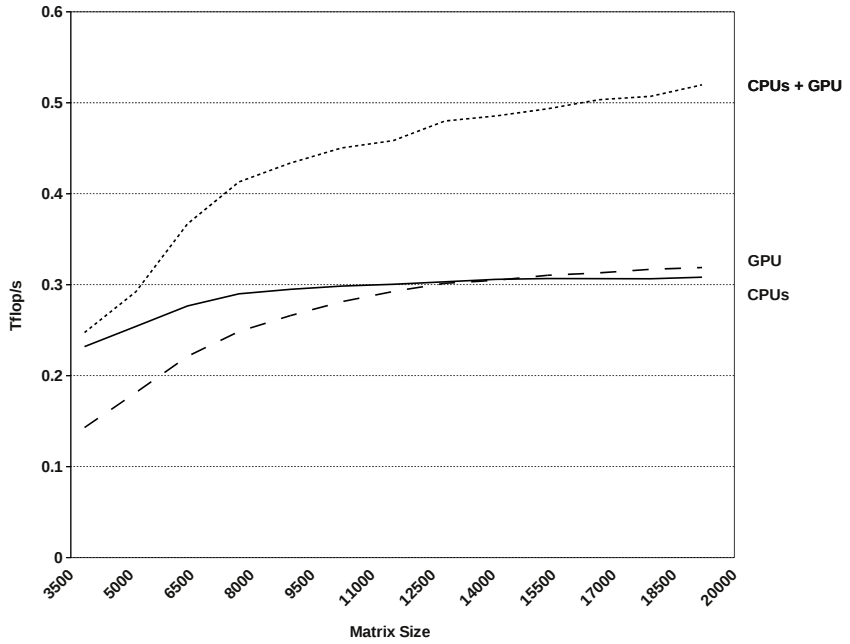

FIGURE 1.10: Performance of the QR factorization using 24 AMD-based cores and an NVIDIA Fermi GPU.

implements the GPU work, such that one tile of the matrix is (approximately) processed by one multiprocessor of the GPU. At the same time all dependencies corresponding to all the tile operations are created inside the double loop nest.

Although GPU acceleration in PLASMA is currently in a prototype stage, Figure 1.10 shows clearly that this approach allows to efficiently combine the power of a GPU and a big number of conventional CPU cores. In this particular case, the 14 cores (SMs) of the Fermi GPU combined with 24 conventional AMD cores were capable of delivering performance in excess of half a TeraFlop/s.

1.6 Nested Parallelism

This Section describes two unique contributions, which are the use of nested parallelism in QUARK and fine grained parallelization of the LU factorization of a matrix panel using the recursive algorithm [18].

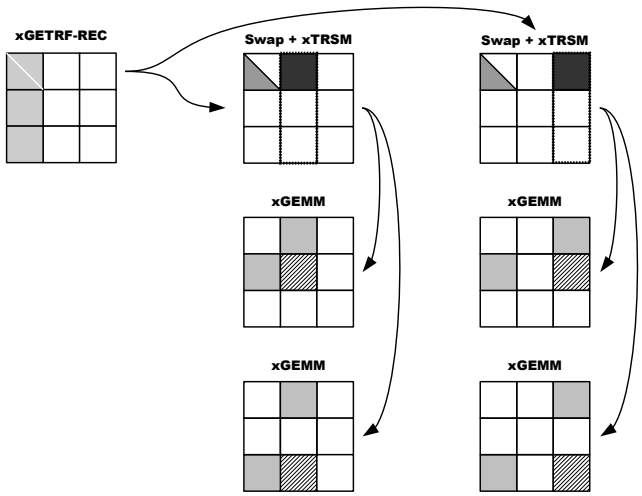

FIGURE 1.11: Execution breakdown for recursive tile LU factorization: factorization of the first panel using the parallel kernel is followed by the corresponding updates to the trailing submatrix.

1.6.1 The Case of Partial Pivoting

Figure 1.11 shows the initial factorization steps of a matrix subdivided into 9 tiles (a 3-by-3 grid of tiles). The first step is a recursive parallel factorization of the first panel consisting of three leftmost tiles. Only when this finishes, the other tasks may start executing, which creates an implicit synchronization point. To avoid the negative impact on parallelism, we execute this step on multiple cores (see Section 1.6.2 for further details) to minimize the running time. However, we use nested parallelism model as most of the tasks are handled by a single core and only the panel tasks are assigned to more than one core. Unlike similar implementations [13], we do not use all cores to handle the panel. There are two main reasons for this decision. First, we use dynamic scheduling that enables us to hide the negative influence of the panel factorization behind more efficient work performed by concurrent tasks. And second, we have clearly observed the effect of diminishing returns when using too many cores for the panel. Consequently, we do not use them all and instead we keep the remaining cores busy with other critical tasks.

The next step is pivoting to the right of the panel that has just been factorized. We combine in this step the triangular update (xTRSM in the BLAS parlance) because there is no advantage of scheduling them separately due to cache locality considerations. Just as the panel factorization locks the panel and has a potential to temporarily stall the computation, the pivot interchange has a similar effect. This is indicated by a rectangular outline encompassing the tile updated by xTRSM of the tiles below it. Even though

```
function xGETRFR(M, N, column) {
  if N == 1 {                                   single column, recursion stops
    idx = split_IxAMAX(...)                     compute local maximum of modulus
    gidx = combine_IxAMAX(idx)                  combine local results
    split_xSCAL(...)                            scale local data
  } else {
    xGETRFR(M, N/2, column)                     recursive call to factor left half
    xLASWP(...)                                 pivoting forward
    split_xTRSM(...)                            triangular solve
    split_xGEMM(...)                            Schur's complement
    xGETRFR(M, N-N/2, column+N/2)               recursive call to factor right half
    xLASWP(...)                                 pivoting backward
  }
}
```

FIGURE 1.12: Pseudo-code for the recursive panel factorization.

so many tiles are locked by the triangular update, there is still a potential for parallelism because pivot swaps and the triangular update itself for a single column is independent of other columns. We can then easily split the operations along the tile boundaries and schedule them as independent tasks. This observation is depicted in Figure 1.11 by showing two xTRSM updates for two adjacent tiles in the topmost row of tiles instead of one update for both tiles at once.

The last step shown in Figure 1.11 is an update based on the Schur complement. It is the most computationally intensive operation in the LU factorization and is commonly implemented with a call to a Level 3 BLAS kernel called xGEMM. Instead of a single call that performs the whole update of the trailing submatrix, we use multiple invocations of the routine because we use a tile-based algorithm. In addition to exposing more parallelism and the ability to alleviate the influence of algorithm's synchronization points (such as the panel factorization), by splitting the Schur update operation we are able to obtain better performance than a single call to a parallelized vendor library [4].

One thing not shown in Figure 1.11 is pivoting to-the-left because it does not occur in the beginning of the factorization. It is necessary for the second and subsequent panels. The swaps originating from different panels have to be ordered correctly but are independent for each column, which is the basis for running them in parallel. The only inhibitor of parallelism then is the fact that the swapping operations are inherently memory-bound because they do not involve any computation. On the other hand, the memory accesses are done with a single level of indirection, which makes them very irregular in practice. Producing such memory traffic from a single core might not take advantage of the main memory's ability to handle multiple outstanding data requests and

the parallelism afforded by NUMA hardware. It is also noteworthy to mention that the tasks performing the pivoting behind the panels are not located on the critical path and therefore, are not essential for the remaining computational steps in the sense that they could be potentially delayed toward the end of the factorization.

1.6.2 Implementation Details of Recursive Panel Factorization

Figure 1.12 shows a pseudo-code of our recursive implementation of panel factorization. Even though the panel factorization is a lower order term – $O(\mathsf{N}^2)$ – from the computational complexity perspective [7], it still poses a problem in the parallel setting from the theoretical [5] and practical standpoints [13]. To be more precise, the combined panel factorizations' complexity for the entire matrix is $O(\mathsf{N}^2\mathsf{NB})$, where N is panel height (and matrix dimension) and NB is panel width. For good performance of BLAS calls, panel width is commonly increased. This creates tension if the panel is a sequential operation because a larger panel width results in larger Amdahl's fraction [17]. Our own experiments revealed this to be a major obstacle to proper scalability of our implementation of tile LU factorization with partial pivoting – a result consistent with related efforts [13].

Aside from gaining high level formulation that is free of low level tuning parameters, recursive formulation affords us to dispense of a higher level tuning parameter commonly called algorithmic blocking. There is already panel width – a tunable value used for merging multiple panel columns together. Non-recursive panel factorizations could potentially establish another level of tuning called *inner-blocking* [3, 4]. This is avoided in our implementation.

1.6.3 Data Partitioning

The challenging part of the parallelization is the fact that the recursive formulation suffers from inherent sequential control flow that is characteristic of the column-oriented implementation employed by LAPACK and ScaLAPACK. As a first step then, we apply a 1D partitioning technique that has proven successful before [13]. We employed this technique for the recursion-stopping case: single column factorization. The recursive formulation of the LU algorithm poses another problem, namely the use of Level 3 BLAS call for triangular solve – `xTRSM()` and LAPACK's auxiliary routine for swapping named `xLASWP()`. Both of these calls do not readily lend themselves to the 1D partitioning scheme due to two main reasons:(1) each call to these functions occurs with a variable matrix size, and (2) 1D partitioning makes the calls dependent upon each other thus creating synchronization overhead. The latter problem is fairly easy to see as the pivoting requires data accesses across the entire column and memory locations may be considered random. Each pivot element swap would then require coordination between the threads that the

column is partitioned amongst. The former issue is more subtle in that the overlapping regions of the matrix create a memory hazard that may be at times masked by the synchronization effects occurring in other portions of the factorization. To deal with both issues at once, we chose to use 1D partitioning across the rows and not across the columns as before. This removes the need for extra synchronization and affords us parallel execution, albeit a limited one due to the narrow size of the panel.

The Schur's complement update is commonly implemented by a call to Level 3 BLAS kernel xGEMM() and this is also a new function that is not present within the panel factorizations from LAPACK and ScaLAPACK. Parallelizing this call is much easier than all the other new components of our panel factorization. We chose to reuse the across-columns 1D partitioning to simplify the management of overlapping memory references and to again reduce resulting synchronization points.

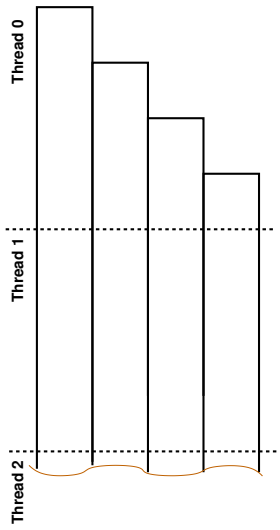

FIGURE 1.13: Fixed partitioning scheme used in the parallel recursive panel factorization.

To summarize the observations that we made throughout the preceding text, we consider data partitioning among the threads to be of paramount importance. Unlike the PCA method [13], we do not perform extra data copy to eliminate memory effects that are detrimental to performance such as TLB misses, false sharing, etc. By choosing the recursive formulation, we rely instead on Level 3 BLAS to perform these optimizations for us. Not surprisingly, this was also the goal of the original recursive algorithm and its sequential implementation [18]. What is left to do for our code is the introduction of parallelism that is commonly missing from Level 3 BLAS when narrow rectangular matrices are involved.

Instead of low level memory optimizations, we turned our focus towards avoiding synchronization points and let the computation proceed asynchronously and independently as long as possible until it is absolutely necessary to perform communication between threads. One design decision that stands out in this respect is the fixed partitioning scheme. Regardless of the current column height (within the panel being factored), we always assign the same amount of rows to each thread except for the first thread. Figure 1.13 shows that this causes a load imbalance as the thread number 0 has progressively smaller amounts of work to perform as the panel factorization progress from the first to the last column. This is counter-balanced by the fact that the panels are relatively tall compared to the number of threads and the first

thread usually has greater responsibility in handling pivot bookkeeping and synchronization tasks.

1.6.4 Scalability Results of the Parallel Recursive Panel Kernel

Figure 1.14 shows a scalability study on a NUMA machine featuring a six-core AMD processor of our parallel recursive panel LU factorization with four different panel widths: 32, 64, 128, and 256 against equivalent routines from LAPACK. We limit our parallelism level to 16 cores because our main factorization needs the remaining cores for trailing matrix updates. When compared with the panel factorization routine xGETF2() (mostly Level 2 BLAS), we achieve super-linear speedup for a wide range of panel heights with the maximum achieved efficiency exceeding 550%. In an arguably more relevant comparison against the xGETRF() routine, which could be implemented with mostly Level 3 BLAS, we achieve perfect scaling for 2 and 4 threads and easily exceed 50% efficiency for 8 and 16 threads. This is consistent with the results presented in the related work section [13].

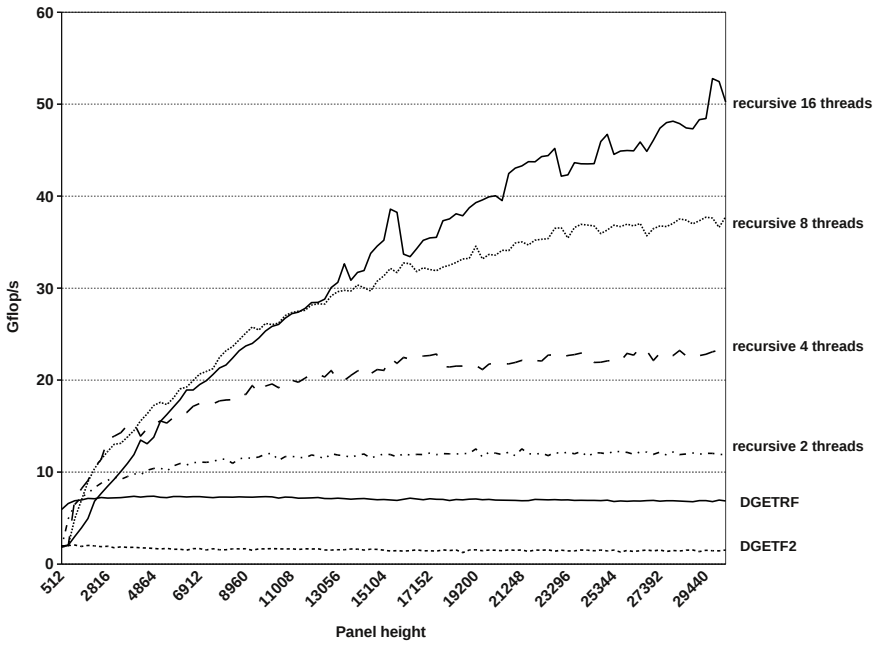

FIGURE 1.14: Scalability study of the recursive parallel panel factorization with various panel width 256.

1.6.5 Further Implementation Details and Optimization Techniques

We exclusively use lockless data structures [23] throughout our code. This choice was dictated by fine granularity synchronization, which occurs during the pivot selection for every column of the panel and at the branching points of the recursion tree. Synchronization using mutex locks was deemed inappropriate at such frequency as it has a potential of incurring system call overhead.

Together with lockless synchronization, we use *busy waiting* on shared-memory locations to exchange information between threads using a coherency protocol of the memory subsystem. While fast in practice [13], this causes extraneous traffic on the shared-memory interconnect, which we aim to avoid. We do so by changing busy waiting for computations on independent data items. Invariably, this leads to riching the parallel granularity levels that are most likely hampered by spurious memory coherency traffic due to false sharing. Regardless of the drawback, we feel this is a satisfactory solution as we are motivated by avoiding busy waiting, which creates even greater demand for inter-core bandwidth because it has no useful work to interleave with the shared-memory polling. We refer this optimization technique to *delayed waiting*.

Another technique we use to optimize the inter-core communication is what we call *synchronization coalescing*. The essence of this method is to conceptually group unrelated pieces of code that require a synchronization into a single aggregate that synchronizes once. The prime candidate for this optimization is the search and the write of the pivot index. Both of these operations require a synchronization point. The former needs a parallel reduction operation while the latter requires global barrier. Neither of these are ever considered to be related to each other in the context of sequential parallelization. But with our synchronization coalescing technique, they are deemed related in the communication realm and, consequently, we implemented them in our code as a single operation.

Finally, we introduced a *synchronization avoidance* paradigm whereby we opt for multiple writes to shared memory locations instead of introducing a memory fence (and potentially a global thread barrier) to ensure global data consistency. Multiple writes are usually considered a hazard and are not guaranteed to occur in a specific order in most of the consistency models for shared memory systems. We completely side step this issue, however, as we guarantee algorithmically that each thread writes exactly the same value to memory. Clearly, this seems as an unnecessary overhead in general, but in our tightly coupled parallel implementation this is a worthy alternative to either explicit (via inter-core messaging) or implicit (via memory coherency protocol) synchronization. In short, this technique is another addition to our contention-free design.

Portability, and more precisely, performance portability, was also an im-

portant goal in our overall design. In our lock-free synchronization, we heavily rely on shared-memory consistency – a problematic feature from the portability standpoint. To address this issue reliably, we make two basic assumptions about the shared-memory hardware and the software tools. Both of which, to our best knowledge, are satisfied on majority of modern computing platforms. From the hardware perspective, we assume that memory coherency occurs at the cache line granularity. This allows us to rely on global visibility of loads and stores to nearby memory locations. What we need from the compiler toolchain is an appropriate handling of C's volatile keyword. This, combined with the use of primitive data types that are guaranteed to be contained within a single cache line, is sufficient in preventing unintended shared-memory side effects.

Bibliography

[1] E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *Proceedings of the 9th International Meeting on High Performance Computing for Computational Science, VECPAR'10*, Berkeley, CA, June 22-25 2011.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009.

[4] Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[5] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Atlantic City, N.J., APR 18-20 1967. AFIPS Press, Reston, VA.

[6] 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.

[7] E. Anderson and J. Dongarra. Implementation guide for LAPACK. Technical Report UT-CS-90-101, University of Tennessee, Computer Science Department, April 1990. LAPACK Working Note 18.

[8] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Computat. Pract. Exper.*, 2010. (to appear).

[9] Barcelona Supercomputing Center. *SMP Superscalar (SMPSs) User's Manual, Version 2.0*, 2008.

[10] Basic Linear Algebra Technical Forum. *Basic Linear Algebra Technical Forum Standard*, August 2001.

[11] 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.

[12] 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.

[13] Anthony M. Castaldo and R. Clint Whaley. Scaling LAPACK panel operations using parallel cache assignment. *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 223–232, 2010.

[14] C interface to the BLAS. `http://www.netlib.org/blas/blast-forum/cblas.tgz`.

[15] CLAPACK (f2c'ed version of LAPACK). `http://www.netlib.org/clapack/`.

[16] f2c. `http://www.netlib.org/f2c/`.

[17] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of ACM*, 31(5):532–533, 1988.

[18] Fred G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[19] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[20] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency Computat.: Pract. Exper.*, 21(1):15–44, 2009.

[21] Piotr Luszczek, Hatem Ltaief, and Jack Dongarra. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In *Proceedings of IPDPS 2011*, Anchorage, Alaska, May 16-20 2011.

[22] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, 2008.

[23] Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures*. Department of computer science, Chalmers University of Technology, Göteborg, Sweden, November 5 2004. PhD dissertation.

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

[25] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. QUARK users' guide: Queuing and runtime for kernels. technical report UT-ICL-11-02, University of Tennessee Innovative Computing Laboratory, Knoxville, Tennessee 37996, April 2011.