Int J Parallel Prog
DOI 10.1007/s10766-016-0441-6

Porting the PLASMA Numerical Library to the OpenMP Standard

Asim YarKhan[1] · **Jakub Kurzak**[1] · **Piotr Luszczek**[1] · **Jack Dongarra**[1]

Received: 4 January 2016 / Accepted: 31 May 2016
© Springer Science+Business Media New York 2016

Abstract PLASMA is a numerical library intended as a successor to LAPACK for solving problems in dense linear algebra on multicore processors. PLASMA relies on the QUARK scheduler for efficient multithreading of algorithms expressed in a serial fashion. QUARK is a superscalar scheduler and implements automatic parallelization by tracking data dependencies and resolving data hazards at runtime. Recently, this type of scheduling has been incorporated in the OpenMP standard, which allows to transition PLASMA from the proprietary solution offered by QUARK to the standard solution offered by OpenMP. This article studies the feasibility of such transition.

Keywords Parallel computing · Multithreading · Multicore processors · Programming models · Runtime systems · Task scheduling · Numerical libraries · Linear algebra

This work has been supported in part by the National Science Foundation Grants Numbers: 1339822 and 1527706.

✉ Jakub Kurzak
kurzak@eecs.utk.du

Asim YarKhan
yarkhan@eecs.utk.du

Piotr Luszczek
luszczek@eecs.utk.du

Jack Dongarra
dongarra@eecs.utk.du

[1] Electrical Engineering and Computer Science, University of Tennessee, 1122 Volunteer Blvd, Ste 203 Claxton, Knoxville, TN 37996, USA

Published online: 14 June 2016

1 Introduction

Dataflow scheduling is a very old idea that offers numerous benefits to traditional multithreading and message-passing, but it never gained wide acceptance until the dawn of the multicore era. Part of the problem is cultural, as dataflow programming takes some control away from the programmer by forcing a more declarative, rather than imperative, style of coding. One needs to let the system decide what work is executed where and at what time.

The multicore revolution of the late 2000's brought the idea back and put it in the mainstream. One of the first task-based multithreading systems that received attention in the multicore times, was the Cilk programming language, originally developed at MIT in the 1990s. Cilk offers nested parallelism based on a tree of threads that is heavily geared towards recursive algorithms. About the same time, the idea of superscalar scheduling gained traction, based on scheduling tasks by resolving data hazards in real time, in a similar way that superscalar processors dynamically schedule instructions on CPUs.

The appeal of the superscalar model is its simplicity. The user presents the compiler with serial code, where tasks are, in principle, functions. They have to be free of side-effects (no internal state and no access to global state). In addition, the parameters have to be marked as passed by-values or as references to memory locations that will be used as input, output, input-and-output, or a temporary buffer (scratch). This is very similar to the attributes available in modern Fortran standard. Based on this information, the task graph is built at runtime and tasks are scheduled in parallel and assigned to multiple cores by resolving data dependencies and avoiding data hazards (Sect. 3.2).

2 Related Work

This superscalar technique was pioneered by a software project from the Barcelona Supercomputer Center, which went through multiple names as its hardware target was changing: GridSs, CellSs, SMPSs, OMPSs, StarSs, where "Ss" stands for superscalar [8,18,30]. A similar development was the StarPU project from INRIA, which applied the same methodology to systems with GPU accelerators, named for its capability to schedule work to "C"PUs and "G"PUs, hence the name *PU, transliterated into StarPU [7]. Yet another scheduler was developed at the Uppsala University, called SuperGlue [34]. Finally, a system called QUARK was developed at the University of Tennessee [36], and used for implementing the PLASMA numerical library [2].

The OpenMP community has been swiftly moving forward with standardization of new scheduling techniques for multicores. First, the OpenMP 3.0 standard [29] adopted the Cilk scheduling model, then the OpenMP 4.0 standard [29] adopted the superscalar scheduling model. Not without significance is the fact that the GNU compiler suite was also quick to follow with high quality implementations of the new extensions. The preliminary results, presented in this article, indicate that, at this point, there are no roadblocks to fully migrate the PLASMA library from QUARK to OpenMP.

Sadly, the situation is worse in programming distributed memory machines, where the MPI+X model is currently the predominant solution (meaning MPI+OpenMP / POSIX threads / CUDA / OpenCL / OpenACC / etc.) While the superscalar model can be extended to support distributed memory, the paradigm is inherently non-scalable to very large core numbers, due to the serial bottleneck of unrolling the task graph at runtime. Numerous project tackled the problem, Charm++ from UIUC [26], Swift from Argonne [35,37], ParalleX from LSU (now Indiana) [19,25], just to name a few. The PaRSEC [9] and PULSAR [15] projects were developed at UTK. However, while dataflow scheduling seems inevitable at exascale, there is currently no paradigm on the radar that could serve as basis for standardization.

3 Background

3.1 PLASMA

Parallel Linear Algebra Software for Multicore Architectures (PLASMA) [2] was developed to address the performance deficiency of the LAPACK library [6] on multicore processors. LAPACK's shortcomings stem from the fact that its algorithms are coded in a serial fashion, with parallelism only possible inside the set of *Basic Linear Algebra Subroutines* (BLAS) [17], which forces a fork-and-join style of multithreading. In contrast, algorithms in PLASMA are coded in a way that allows for, much more powerful, dataflow parallelism [11,28]. Currently, PLASMA supports a substantial subset of LAPACK's functionality, including solvers for linear systems of equations, linear least squares problems, singular value problems, symmetric eigenvalue problems and generalized symmetric eigenvalue problems. It also provides a full set of Level 3 BLAS routines for matrices stored in tile layout and a highly optimized (cache-efficient and multithreaded) set of routines for layout translation [21]. PLASMA is based on three principles: tile algorithms, tile matrix layout and task-based scheduling, all of which are geared towards efficient multithreaded execution.

Tile algorithms are the cornerstone of PLASMA [11]. In tile algorithms, matrix entries are represented by square tiles of relatively small size, such that multiple cores can operate on different tiles independently. At the same time, a tile can be cached and fully processed, before being evicted from the cache, which helps to minimize the number of capacity misses. This is in contrast with LAPACK, where one tall panel (block of columns) is eliminated at a time, making it difficult to achieve cache efficiency and apply multithreading. In the course of the PLASMA project, tile algorithms have been developed for a wide range of algorithms, including: Cholesky, LU and QR factorizations [11,14,16], as well as reductions to band forms for solving the singular value problem or the eigenvalue problem [23,31].

The development of tile algorithms made it natural to lay out the matrix in memory by tiles, where each tile occupies a contiguous block of memory. PLASMA provides cache-efficient, multithreaded routines for in-place translation of matrices from the LAPACK layout to the tile layout [21]. Tile layout prevents the possibility of conflict/collision misses, which are due to cache lines being mapped to the same set in a set-associative cache. It also minimizes the possibility of false sharing of cache lines

crossing tile boundaries. Most importantly, though, tile layout hugely simplifies the use of dataflow scheduling by making sure that each piece of data is contiguous and can be easily manipulated by the runtime system, e.g., copied to eliminate certain types of dependencies.

Initially, PLASMA routines were implemented using the *Single Program Multiple Data* (SPMD) programming style, where each thread executed a statically scheduled preassigned set of tasks, while coordinating with other threads using progress tables, with busy-waiting as the main synchronization mechanism. While seemingly very rudimentary, this approach actually produced very efficient, systolic-like, pipelined processing patterns, with good data locality and load balance. At the same time, the codes were hard to develop, error prone and hard to maintain, which motivated a move towards dynamic scheduling.

3.2 QUARK

The basic principle of QUARK's operation is automatic parallelization of code that is structured as a sequence of side effect free tasks (no internal state and no access to global state) with arguments annotated as input, output or inout. QUARK tracks data pointers and creates a list of tasks accessing each data pointer. Given the order in which tasks are inserted and the data annotations, the following data dependencies can be tracked:

- *read after write:* Read access has to wait until a preceding write completes. Otherwise an outdated copy would be read.
- *write after read:* Write access has to wait until a preceding read completes. Otherwise new value would overwrite the old value, while the old value is still needed.
- *write after write:* Write access has to wait until a preceding write completes. Otherwise the new value would be overwritten by the old value.

These dependencies define the *Direct Acyclic Graph* (DAG) of the algorithm. However, the DAG is never explicitly created. Instead, DAG traversal is implicitly realized in the way tasks are queued for execution and data accesses are blocked to preserve correctness. This process relies on enforcement of two basic rules: (1) Writes cannot proceed until all previous reads and writes complete. (2) Reads cannot proceed until all previous writes complete, i.e., multiple reads can occur at the same time.

QUARK provides scheduling capabilities through an API with bindings for the C language. Figure 1 shows the primary functions of the QUARK API. QUARK is initialized by calling the `QUARK_New` function, which spawns the threads that are subsequently used as workers. The desired number of threads can be provided as a parameter to `QUARK_New` or specified in the environment variable `QUARK_NUM_THREADS`. QUARK is terminated by calling the `QUARK_Delete` function, which terminates the worker threads. Between the calls to `QUARK_New` and `QUARK_Delete`, the user can insert tasks for execution, by calling the `QUARK_Insert_Task` function.

The first three parameters of `QUARK_Insert_Task` are: a pointer to the QUARK object, a pointer to the function that is to be executed, and a bitmask with a set of flags to provide task specific information. They are followed by a variable length list of parameters describing the task's data. Each data object is presented as a triplet: the

```
Quark *QUARK_New(int num_threads);

unsigned long long QUARK_Insert_Task(
    Quark *quark, void (*function)(Quark*),
    Quark_Task_Flags *task_flags,
    int sizeof_arg_1_in_bytes, void *arg_1, int arg_1_flags,
    int sizeof_arg_1_in_bytes, void *arg_1, int arg_1_flags,
    ...,
    0);

void QUARK_Delete(Quark *quark);
```

Fig. 1 The primary functions of the QUARK API

size of the data in bytes, a pointer to the data, and a flag indicating the way that the data is to be used. The sequence is terminated by a zero. The data flags have to include one of the following, mutually exclusive values:

- `INPUT`: The data is only read.
- `OUTPUT`: The data is only written.
- `INOUT`: The data is both read and written.
- `VALUE`: The data is passed by value and not used for dependency resolution.
- `NODEP`: The data is passed by reference, but not used for dependency resolution.
- `SCRATCH`: The data is temporary scratch space and, if the pointer is NULL, will be allocated by QUARK.

Optionally, one of the following flags may also be included:

- `ACCUMULATOR`: The data is subject to a reduction operation and successive accesses of that type can be safely reordered.
- `GATHERV`: The data is accessed in a non-conflicting manner and multiple accesses of that type can be performed simultaneously.
- `LOCALITY`: Task placement should attempt to follow this data item, i.e., consecutive tasks with this flag should be executed by the same thread.
- `QUARK_REGION_X`: A specific region of the data is accessed (QUARK_REGION_0 through QUARK_REGION_7). Accesses to different regions do not cause conflicts.

At the time of queuing, each task can be augmented with a number of settings, passed in an object of type `Quark_Task_Flags`. Through the use of this object, the user can:

- Set task priority such that tasks with higher priorities are executed before tasks with lower priorities.
- Lock a task to a thread with a specific number.
- Associate the task with a specific *task sequence*. In the case of an error, an entire sequence of tasks can be canceled. At the same time, unrelated tasks, assigned to other sequences can proceed independently.
- Designate the task as a multithreaded task. At the time of execution, the task function will be called by multiple threads. QUARK is oblivious to the type of

```
#pragma omp task depend(in:A[0:nA],B[0:nB]) depend(inout:C[0:nC])
call_some_function( A, B, C )
```

Fig. 2 OpenM `#depend` annotation example

multithreading (synchronization) taking place within the task and schedules it as one unit of work executed by multiple threads.

Finally, QUARK allows for creating an empty task, without any dependencies, and then adding dependencies incrementally. This is critical in situations when the task's dependencies are a function of the loop counter. For instance, in case of the LU factorization (Sect. 4.4), in order to build the list of dependencies for the panel, the code needs to loop over all the tiles of the panel.

3.3 OpenMP

OpenMP (*Open Multi-Processing*) is an *Application Programming Interface* (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, OS X, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

The first OpenMP specification, Fortran version 1.0, was published in October 1997. C/C++ version 1.0 was published in October 1998. Fortran version 2.0 was published in November 2000, and C/C++ version 2.0 in March 2002. Version 2.5 combined Fortran and C/C++ interfaces and was published in May 2005. Version 3.0 introduced the concept of tasks, and was published in May 2008. The critical development, from the standpoint of this work, was the introduction of task superscalar scheduling in OpenMP 4.0, published in July 2013.

Equally important is the support for the standard in compilers. The GNU compiler suite has been on the forefront of adopting the standard. Support for OpenMP 2.5 was added in GCC 4.2 (May 2007), support for OpenMP 3.0 was added in GCC 4.4 (April 2009), and support for OpenMP 4.0 was added in GCC 4.9 (April 2014).

The `#pragma omp task depend` clause can be used to inform the compiler that the following code block is to be executed as a task (Fig. 2), i.e., dynamically scheduled at runtime, and that it depends on earlier tasks that use the data referenced by the `depend` clause.

- The `in` dependence makes the task a descendant of any previously inserted task that lists the data items (i.e., A or B) in its `out` or `inout` dependency list.
- The `inout` or `out` dependence makes the task a descendant of any previously inserted task that lists the data item (i.e., C) in its `in`, `inout` or `out` dependency list.

The OpenMP runtime provides the capability to schedule the tasks at runtime, while avoiding data hazards by keeping track of dependencies. A task is not scheduled until its dependencies are fulfilled, and then, when a task completes, it fulfills dependencies for other tasks.

This type of scheduling is more complex than the simpler scheduling of OpenMP 3.0 tasking, and the overhead associated with keeping track of progress may not be negligible. Part of the objective of this article is to evaluate the quality of the GNU implementation, with respect of the QUARK scheduler currently used for PLASMA.

The OpenMP 4.0 standard is missing some of the linear algebra extensions that were implemented by QUARK (e.g. automatic scratch space allocation, data region flag to mark multi-region non-conflicting access, multithreaded tasks, etc). In implementing linear algebra algorithms, workarounds need to be restricted to the OpenMP paradigm, even though some of these solutions may not be as efficient as the original QUARK extensions. The following examples discuss some of these QUARK extensions in OpenMP, demonstrating a variety of simplicity and efficiency. The QUARK scratch data type is emulated in OpenMP by having the tasks themselves allocate data. In QUARK, data regions speed up data accesses which would cause a WAR constraint by allowing simultaneous non-conflicting data access to different parts of a data tile (upper/diagonal/lower regions in linear algebra terms). This is handled in OpenMP by inserting tasks to create copies of the data which are passed to the tasks that are only reading the data. The QUARK multi-threaded tasks are emulated in OpenMP by creating a dummy task that synchronizes the data using read-write data access, then inserts the actual task multiple times with data marked read-only (enabling multi-threaded parallel execution), finalizing with a dummy task that synchronizes using read-write data access. All this has to be carefully managed to avoid dead-locks. And, in the absence of task priorities, this OpenMP emulation is often substantially slower. In QUARK, the accumulator flag marks a data reduction, allowing the runtime to reorder tasks for efficiency; there is currently no equivalent in OpenMP to allow task reordering.

4 Implementations

4.1 Structure of PLASMA

Following the tradition of LAPACK and ScaLAPACK, PLASMA has a fairly shallow structure. The top level routines follow LAPACK naming conventions and are prefixed with `PLASMA_`, e.g., PLASMA routine for the Cholesky factorization in double precision is `PLASMA_dpotrf`, which stands for "d"ouble precision, "po"sitive definite, "tr"iangular, "f"actorization. This routine takes the input matrix in LAPACK layout (standard FORTRAN column major layout) and returns the result in the same layout. It also has the standard synchronous semantics, i.e., when the routine returns, all the work is done.

Internally, PLASMA operates on matrices in the tile layout only. All top level routines translate the matrix to the tile layout, do their work, and translate the result back to the LAPACK layout. The user also has the ability to work directly with the tile layout. In addition to the standard interface, PLASMA also provides the tile interface. The tile routine for the Cholesky factorization is `PLASMA_dpotrf_Tile`.

Finally, the main premise of PLASMA is dynamic scheduling and asynchronous operation. Therefore, PLASMA also exposes an asynchronous interface. The asyn-

chronous routine for the Cholesky factorization is `PLASMA_dpotrf_Tile_Async`. Asynchronous routines return as soon as all the work is queued for execution, but, possibly, before all the work has been done. The `PLASMA_Sequence_Wait` function can be used to wait for completion of asynchronous routines.

Actual parallel routines are called by the tile asynchronous routines. The tile synchronous routines call the asynchronous routines and wait for completion. The LAPACK interface routines call the tile synchronous routines, and also translate the matrix between the LAPACK layout and the tile layout.

The top level routines call the parallel routines where the actual algorithm is implemented. The parallel routines are prefixed with `plasma_p`. PLASMA may contain two versions of each routine, one with static multithreading and one with superscalar scheduling. The parallel Cholesky routine with static multithreading is `plasma_pdpotrf`. The parallel Cholesky routine with superscalar scheduling is `plasma_pdpotrf_quark`. Which one is called at runtime depends on the user's preference. The static one is called by default. It is possible that a routine has only one of the implementations, in which case the user's preference is ignored and the only available implementation is called. Threads are managed such that switching between static and dynamic scheduling at runtime is seamless to the user.

The parallel routines are composed of calls to a set of basic building blocks collected in the `core_blas` sublibrary. Some `core_blas` routines are simple wrappers for LAPACK and BLAS routines. E.g., `core_dpotrf` calls serial `LAPACKE_dpotrf_work` and returns, `core_dgemm` calls serial `cblas_dgemm` and returns. Other `core_blas` calls are more complex and call a combination of LAPACK and BLAS routines or loop over a set of LAPACK and BLAS routines. Some of the more complex `core_blas` routines are the routines for applying a set of Householder reflections in the tile QR factorization, e.g., `CORE_dtsmqr`.

4.2 Cholesky Factorization

The Cholesky decomposition, or Cholesky factorization, is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. When it is applicable, the Cholesky decomposition is roughly twice as efficient as the LU decomposition for solving systems of linear equations. Because of its amenability to the tasking model, the Cholesky factorization has been a target of numerous multicore implementations [11,24,27,28,33].

The Cholesky factorization is one of the most straightforward algorithms in PLASMA. Figure 3 shows PLASMA's superscalar implementation of the Cholesky factorization. This is the right looking algorithm implemented by four loops with three levels of nesting. First, the `dpotrf` task factors a diagonal block. Then, a sequence of `dtrsm` tasks updates a column of tiles below the diagonal block. Finally, a large sequence of `dsyrk` and `dgemm` tasks updates all the tiles to the right (the trailing submatrix). The figure was simplified from the original PLASMA code by assuming that the matrix is evenly divisible by the tile size *nb*, and removing all the calculations related to the handling of corner cases.

```
for (k = 0; k < nt; k++) {
    QUARK_CORE_dpotrf(
        plasma->quark, &task_flags,
        PlasmaLower,
        nb, nb,
        A(k, k), nb,
        sequence, request, nb*k);

    for (m = k+1; m < nt; m++) {
        QUARK_CORE_dtrsm(
            plasma->quark, &task_flags,
            PlasmaRight, PlasmaLower, PlasmaTrans, PlasmaNonUnit,
            nb, nb, nb,
            1.0, A(k, k), nb,
                 A(m, k), nb);
    }
    for (m = k+1; m < nt; m++) {
        QUARK_CORE_dsyrk(
            plasma->quark, &task_flags,
            PlasmaLower, PlasmaNoTrans,
            nb, nb, nb,
            -1.0, A(m, k), nb,
             1.0, A(m, m), nb);

        for (n = k+1; n < m; n++) {
            QUARK_CORE_dgemm(
                plasma->quark, &task_flags,
                PlasmaNoTrans, PlasmaTrans,
                nb, nb, nb, nb,
                -1.0, A(m, k), nb,
                      A(n, k), nb,
                 1.0, A(m, n), nb);
        }
    }
}
```

Fig. 3 QUARK based Cholesky implementation in PLASMA. Handling of corner cases removed for clarity

All calls for queuing the tasks basically take identical parameters as their corresponding LAPACK and BLAS calls. In addition, each task takes a QUARK handle and a reference to task flags. The `dpotrf` task is also passed a sequence handle and a request handle. This is for the purpose of error handling. In case of an error, the work is canceled and the reference to the offending request is returned in the sequence handle.

Queuing a task requires passing three parameters for every parameter that would normally be passed to a regular function: size of the argument, pointer to the argument and the type of access. Therefore, to simplify the bodies of the parallel functions, the queuing calls are encapsulated in a set of `QUARK_CORE_` functions. Figure 4 shows the function for queuing the `dpotrf` task.

When queuing the task, a pointer is passed to the function that will actually be called when the scheduler executes the task. In Fig. 4, the second argument of the

```
void QUARK_CORE_dpotrf(
    Quark *quark, Quark_Task_Flags *task_flags,
    PLASMA_enum uplo,
    int n, int nb,
    double *A, int lda,
    PLASMA_sequence *sequence, PLASMA_request *request,
    int iinfo)
{
    QUARK_Insert_Task(
        quark, CORE_dpotrf_quark, task_flags,
        sizeof(PLASMA_enum),       &uplo,     VALUE,
        sizeof(int),               &n,        VALUE,
        sizeof(double)*nb*nb,       A,        INOUT,
        sizeof(int),               &lda,      VALUE,
        sizeof(PLASMA_sequence*), &sequence, VALUE,
        sizeof(PLASMA_request*),  &request,  VALUE,
        sizeof(int),               &iinfo,    VALUE,
        0);
}
```

Fig. 4 QUARK function for queuing the `dpotrf` task (complete function)

```
void CORE_dpotrf_quark(Quark *quark)
{
    PLASMA_enum uplo;
    int n;
    double *A;
    int lda;
    PLASMA_sequence *sequence;
    PLASMA_request *request;
    int iinfo;

    int info;
    quark_unpack_args_7(quark, uplo, n, A, lda, sequence, request, iinfo);
    info = LAPACKE_dpotrf_work(
        LAPACK_COL_MAJOR,
        lapack_const(uplo),
        n, A, lda);
    if (sequence->status == PLASMA_SUCCESS && info != 0)
        plasma_sequence_flush(quark, sequence, request, iinfo+info);
}
```

Fig. 5 QUARK function executing the `dpotrf` task (complete function)

`QUARK_Insert_Task` call is the `CORE_dpotrf_quark` function pointer. Figure 5 shows the body of that function.

The `CORE_dpotrf_quark` function retrieves the arguments by using the `quark_unpack_args_7` macro and passes them to the `LAPACKE_dpotrf_work` function, which is the C interface to the LAPACK `dpotrf` function. This function also takes care of error handling through the `info` parameter.

Figure 6 shows the OpenMP implementation of the Cholesky factorization. Here, there is no need for any wrappers since the OpenMP compiler and runtime provide that functionality. Tasks are queued directly by calls to LAPACKE and CBLAS routines, preceded by `#pragma omp task depend` annotations, and the entire code is enclosed by `#pragma omp parallel` and `#pragma omp master`. The `depend` clauses provide dependency information (in,out,inout) and sizes for all parameters for which dependencies have to be tracked.

4.3 SPD Matrix Inversion

Given the Cholesky factorization of a matrix, it is straightforward to compute the inverse of the matrix. This operation has applications in statistics, where it is used to form the inverse covariance matrix, also known as the concentration matrix or precision matrix. The elements of the precision matrix have an interpretation in terms of partial correlations and partial variances.

In order to explicitly form the inverse, the factorization ($A = LL^T$) is followed by computing the inversion of the triangular factor L, which is straightforward, and multiplication of the inverted triangular factor by its transpose ($A^{-1} = L^{-1^T} L^{-1}$). In PLASMA, it can be done by invoking the `pdpotrf` function followed by the `pdtrtri` function and the `pdlauum` function. This case has been a subject of extensive studies, because dynamic scheduling leads to extremely efficient pipelining of the three operations [1,10,32].

The QUARK implementations of `pdtrtri` and `pdlauum` are similar to the QUARK implementation of `pdpotrf` from Fig. 6, and rely on two wrapper functions for each `core_blas` call (`QUARK_CORE_...` and `CORE_..._quark`). QUARK implementations are omitted here. Instead, Figs. 7 and 8 show their OpenMP implementations. As in the case of `pdpotrf`, the OpenMP implementations are very straightforward, the only differences from serial code being the `#pragma omp task depend` annotations and enclosure of the code with `#pragma omp parallel` and `#pragma omp master`.

4.4 LU Decomposition

LU decomposition, or LU factorization, factors a matrix as the product of a lower triangular matrix and an upper triangular matrix and usually also includes a permutation matrix as well. The LU decomposition can be viewed as the matrix form of Gaussian elimination. It is usually used to solve square systems of linear equations and is also a key step when inverting a matrix or computing the determinant of a matrix.

The main difficulty in efficiently implementing the LU factorization is introduced by partial (row) pivoting, necessary to preserve the algorithm's stability. Because of the row pivoting, the panel factorization (factorization of a block of columns) cannot be tiled, and the panel has to be dealt with as a whole, which handicaps cache efficiency. In recent years, this problem has been addressed by algorithms that try to achieve good

```
#pragma omp parallel
#pragma omp master
{
    for (k = 0; k < nt; k++) {
        #pragma omp task depend(inout:A(k, k)[0:nb*nb])
        LAPACKE_dpotrf_work(
            LAPACK_COL_MAJOR,
            'L', nb, A(k, k), nb);

        for (m = k+1; m < nt; m++) {
            #pragma omp task depend(in:A(k, k)[0:nb*nb]) \
                             depend(inout:A(m, k)[0:nb*nb])
            cblas_dtrsm(
                CblasColMajor,
                CblasRight, CblasLower,
                CblasTrans, CblasNonUnit,
                nb, nb,
                1.0, A(k, k), nb,
                     A(m, k), nb);
        }
        for (m = k+1; m < nt; m++) {
            #pragma omp task depend(in:A(m, k)[0:nb*nb]) \
                             depend(inout:A(m, m)[0:nb*nb])
            cblas_dsyrk(
                CblasColMajor,
                CblasLower, CblasNoTrans,
                nb, nb,
                -1.0, A(m, k), nb,
                 1.0, A(m, m), nb);

            for (n = k+1; n < m; n++) {
                #pragma omp task depend(in:A(m, k)[0:nb*nb]) \
                                 depend(in:A(n, k)[0:nb*nb]) \
                                 depend(inout:A(m, n)[0:nb*nb])
                cblas_dgemm(
                    CblasColMajor,
                    CblasNoTrans, CblasTrans,
                    nb, nb, nb,
                    -1.0, A(m, k), nb,
                          A(n, k), nb,
                     1.0, A(m, n), nb);
            }
        }
    }
}
```

Fig. 6 OpenMP based Cholesky implementation in PLASMA. Handling of corner cases removed for clarity

level of cache residency for this operation [12,16]. Many alternative approaches have also been developed [14], but this article focuses on the classic formulations.

Figure 9 shows a pseudo-code of the recursive implementation. Even though the panel factorization is a lower order term—$O(N^2)$—from the computational complex-

```
for (k = 0; k < nt; k++) {

    for (m = k+1; m < nt; m++) {
        #pragma omp task depend(in:A(k, k)[0:nb*nb]) \
                         depend(inout:A(m, k)[0:nb*nb])
        cblas_dtrsm(
            CblasColMajor,
            CblasRight, CblasLower,
            CblasNoTrans, CblasNonUnit,
            nb, nb,
            -1.0, A(k, k), nb,
                  A(m, k), nb);
    }
    for (m = k+1; m < nt; m++) {

        for (n = 0; n < k; n++) {
            #pragma omp task depend(in:A(m, k)[0:nb*nb]) \
                             depend(in:A(k, n)[0:nb*nb]) \
                             depend(inout:A(m, n)[0:nb*nb])
            cblas_dgemm(
                CblasColMajor,
                CblasNoTrans, CblasNoTrans,
                nb, nb, nb,
                1.0, A(m, k), nb,
                     A(k, n), nb,
                1.0, A(m, n), nb);
        }
    }
    for (n = 0; n < k; n++) {
        #pragma omp task depend(in:A(k, k)[0:nb*nb]) \
                         depend(inout:A(k, n)[0:nb*nb])
        cblas_dtrsm(
            CblasColMajor,
            CblasLeft, CblasLower,
            CblasNoTrans, CblasNonUnit,
            nb, nb,
            1.0, A(k, k), nb,
                 A(k, n), nb);
    }
    #pragma omp task depend(inout:A(k, k)[0:nb*nb])
    LAPACKE_dtrtri_work(
        LAPACK_COL_MAJOR,
        'L', 'N',
        nb, A(k, k), nb);
}
```

Fig. 7 OpenMP based implementation of the `pdtrtri` routine in PLASMA. Handling of corner cases removed for clarity

ity perspective [5], it still poses a problem in the parallel setting from the theoretical [4] and practical standpoints [13]. To be more precise, the combined panel factorizations' complexity for the entire matrix is $O(N^2NB)$, where N is the panel height (and

```
for (k = 0; k < nt; k++) {

    for(n = 0; n < k; n++) {
        #pragma omp task depend(in:A(k, n)[0:nb*nb]) \
                         depend(inout:A(n, n)[0:nb*nb])
        cblas_dsyrk(
            CblasColMajor,
            CblasLower, CblasTrans,
            nb, nb,
            1.0, A(k, n), nb,
            1.0, A(n, n), nb);

        for(m = n+1; m < k; m++) {
            #pragma omp task depend(in:A(k, m)[0:nb*nb]) \
                             depend(in:A(k, n)[0:nb*nb]) \
                             depend(inout:A(m, n)[0:nb*nb])
            cblas_dgemm(
                CblasColMajor,
                CblasTrans, CblasNoTrans,
                nb, nb, nb,
                1.0, A(k, m), nb,
                     A(k, n), nb,
                1.0, A(m, n), nb);
        }
    }
    for (n = 0; n < k; n++) {
        #pragma omp task depend(in:A(k, k)[0:nb*nb]) \
                         depend(inout:A(n, n)[0:nb*nb])
        cblas_dtrmm(
            CblasColMajor,
            CblasLeft, CblasLower,
            CblasTrans, CblasNonUnit,
            nb, nb,
            1.0, A(k, k), nb,
                 A(k, n), nb);
    }
    #pragma omp task depend(inout:A(k, k)[0:nb*nb])
    LAPACKE_dlauum_work(
        LAPACK_COL_MAJOR,
        'L', nb, A(k, k), nb);
}
```

Fig. 8 OpenMP based implementation of the `pdlauum` routine in PLASMA. Handling of corner cases removed for clarity

matrix dimension) and NB is the panel width. For good performance of BLAS calls, the 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 [20]. It can be determined experimentally that this is a major obstacle to proper scalability of implementation of tile LU factorization with partial pivoting—a result consistent with related efforts that do not involve tile LU [13].

```
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
  }
}
```

Fig. 9 Pseudo-code for the recursive panel factorization

Aside from gaining a high level formulation, free of low level tuning parameters, this recursive formulation avoids a higher level tuning parameter commonly called algorithmic blocking. There is already the panel width—a tunable value used to merge multiple panel columns and process them together. Non-recursive panel factorizations could potentially establish another level of tuning called *inner-blocking* [2,3]. This is avoided in the recursive implementation.

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, 1D data partitioning is applied because it has proven successful before [13]. The data partitioning is used for the recursion-stopping case: effectively a 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 `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 and 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 and known only at runtime. Each pivot element swap would then require coordination between the threads that process the column. 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, the 1D partitioning across the rows and not across the columns as before. This removes the need for extra synchronization and allows for 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 a Level 3 BLAS kernel `xGEMM()` and this is also a new function that is not present in the panel

factorizations from LAPACK and ScaLAPACK. Parallelizing this call is much easier than all the other new components of the panel factorization. The chosen solution is to reuse the across-columns 1D partitioning to simplify the management of overlapping memory references and to again reduce resulting synchronization points.

To summarize the observations made throughout the preceding text, the data partitioning among the threads is of paramount importance. Unlike the PCA method [13], there are no extra data copies to eliminate memory effects that are detrimental to performance such as TLB misses, false sharing, etc. Instead, the choice of recursive formulation allows Level 3 BLAS to perform architecture-specific optimizations behind the scenes. Not surprisingly, this was also the goal of the original recursive algorithm and its sequential implementation [22]. What is left to do 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, the focus is on avoiding synchronization points and allow the computation to 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), the same amount of rows is assigned to each thread except for the first thread.

5 Results and Discussion

5.1 Hardware / Software

The experiments were run on a machine containing 2 sockets with 10 Intel Haswell cores on each socket (Xeon E5-2650 v3 2.30 GHz) for a total of 20 cores. The GCC-5.1.0 compiler suite was used for compilation and GCC libgomp provided the OpenMP 4.0 implementation. Intel's MKL math library (2016.0.109) was used for optimized BLAS operations.

5.2 Cholesky Factorization

Figure 10 shows the execution trace of the OpenMP implementation of the Cholesky factorization, and Fig. 11 shows performance in Gflop/s for matrices of size up to $20,000 \times 20,000$ in comparison to the QUARK implementation and the multithreaded implementation in the Intel MKL library.

In this trace, different colored tiles represent different types of tasks (`dtrsm`, `dgemm`, etc) in the tile Cholesky factorization, with a row of tiles representing the time based execution of tasks on a computational thread. The execution produces a very well compacted trace. There are no visible gaps at the beginning, which indicates no overhead from scheduling. Idle space shows up at the end, when the algorithms runs out of parallel work and the critical path starts dominating the execution, which is a natural and expected behavior for this type of workload.

The quality of scheduling is also reflected in the performance. The OpenMP implementation outperforms the QUARK implementation, and also outperforms the MKL

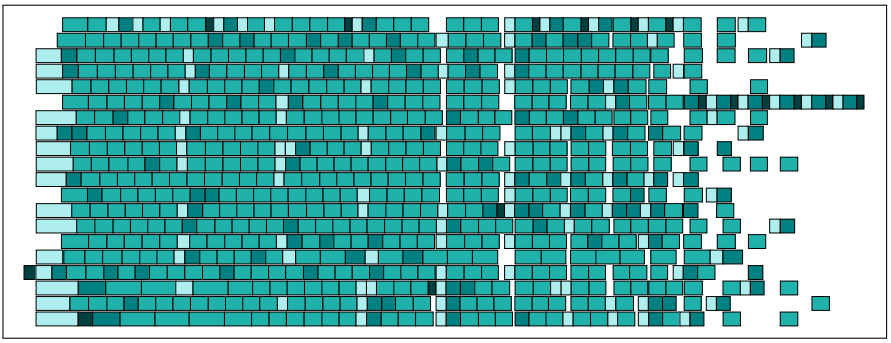

Fig. 10 Execution trace of the OpenMP implementation of the Cholesky factorization. Tiles are of size 224 × 224 elements. The matrix is of size 16 × 16 tiles. The system consists of 20 Intel Haswell cores

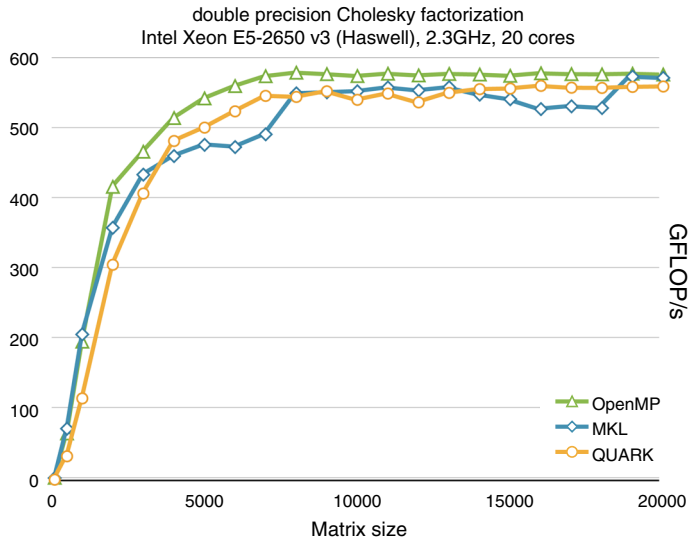

Fig. 11 Performance of OpenMP, QUARK and MKL implementations of the Cholesky factorization using a system with 20 Intel Haswell cores. The peak double precision performance is 736 GFlops

implementation. This is not only an indicator of low scheduling overheads, but also of striking a good balance between data locality and dynamic work balancing.

5.3 SPD Matrix Inversion

Figure 12 shows the execution trace of the SPD matrix inversion. The upper part shows the execution in the case when each stage is enclosed in its own `#pragma omp parallel` section (reflecting a synchronization between stages), while the lower part shows the execution in the case when all the stages are enclosed in a single `#pragma omp parallel` section.

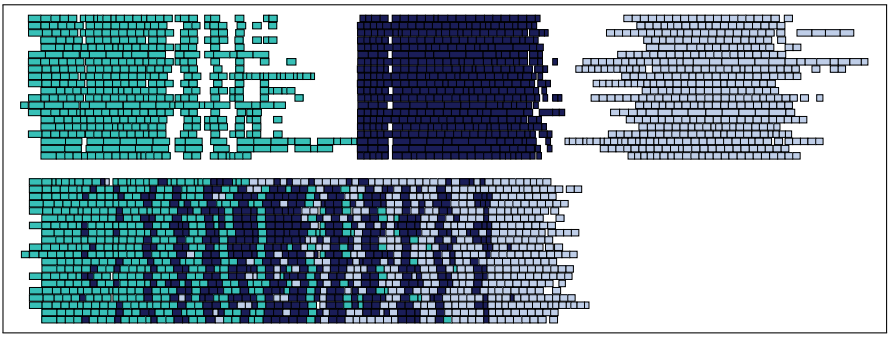

Fig. 12 Execution trace of the OpenMP implementation of the SPD matrix inversion. Tiles are of size 224 × 224 elements. The matrix is of size 13 × 13 tiles. The system consists of 20 Intel Haswell cores. The *upper part* shows execution with artificial synchronizations between the steps (`dpotrf`, `dtrtri`, `dlauum`). The *bottom part* shows natural execution without synchronizations

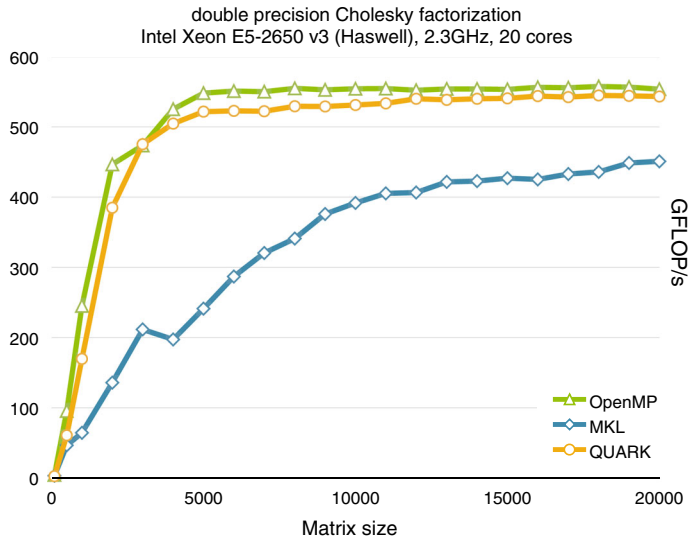

Fig. 13 Performance of OpenMP, QUARK and MKL implementations of the Cholesky inversion using a system with 20 Intel Haswell cores. The peak double precision performance is 736 GFlops

The latter case where the stages are merged shows the huge benefits of applying dataflow scheduling across the multiple dependent stages of multi-part algorithm. While in the former case each stage incurs its own load imbalance, in the latter case the load imbalance is amortized across multiple stages. This is due to the fact that the critical path of the combined task graphs is significantly shorter than the sum of the individual critical paths.

This benefit manifest itself in the big performance advantage over MKL, which does not apply dataflow scheduling across stages (Fig. 13). This task merging behavior is common to all superscalar schedulers, and QUARK exploits the same effect almost

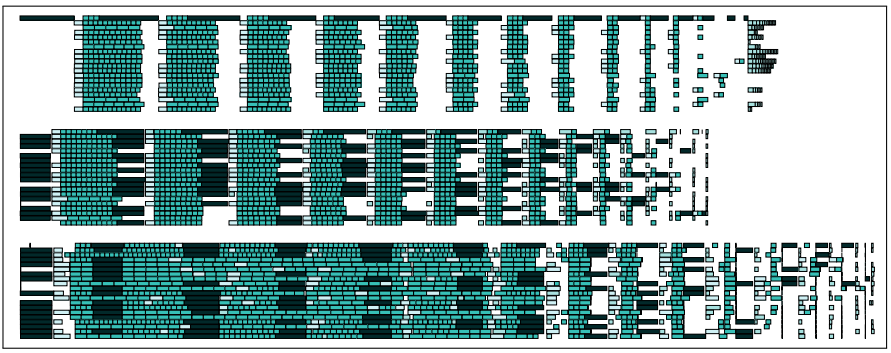

Fig. 14 Execution traces of LU factorization. *Top* OpenMP with single-threaded panel factorization. *Middle* OpenMP with multithreaded panel factorization. *Bottom* QUARK with multithreaded panel factorization. Tiles are of size 288 × 288 elements. The matrix is of size 15 × 15 tiles. The system consists of 20 Intel Haswell cores

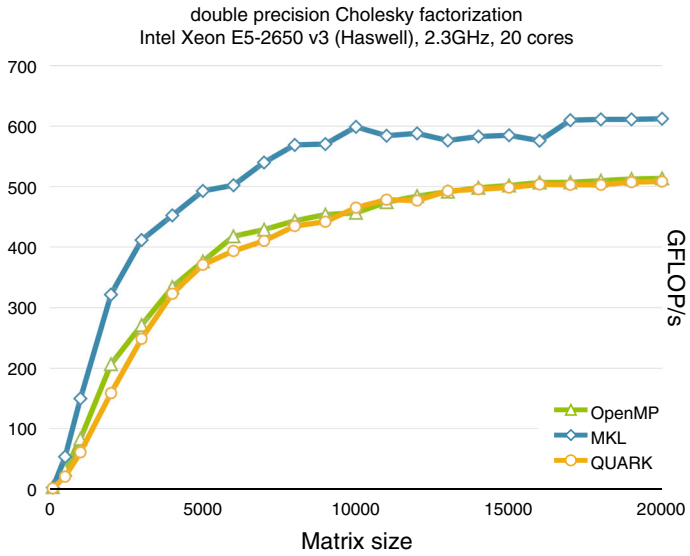

Fig. 15 Performance of OpenMP, QUARK and MKL implementations of the LU factorization using a system with 20 Intel Haswell cores. The peak double precision performance is 736 GFlops

equally well. Nevertheless, the GNU OpenMP implementation manages to deliver the highest speed, again.

5.4 LU Factorization

The LU factorization presents a little tougher case. The key to fast execution is fast panel factorization. Figure 14 shows three traces for the LU factorization. The top one is an OpenMP implementation with serial panel factorization, the middle one is an

OpenMP implementation with multithreaded panel factorization, and the bottom one is a QUARK implementation with multithreaded panel factorization.

The first one suffers from the inability to hide the slow, serial panel factorization. The panel tasks are scheduled early on, but simply take more time than the other operations. The second case addresses the problem by multithreading the panel factorization, but suffers from a new problem of not scheduling all the multithreaded panel tasks to execute early. The scheduler is not aware of the need for prioritizing the panels and there is no way to inform it. QUARK addresses this problem by supporting task priorities, and the third case shows how prioritizing the panel tasks overlaps them with trailing matrix updates, only to produce inferior performance due to poor data locality and a deteriorating schedule towards the end.

With the same tile size, OpenMP consistently produces higher performance than QUARK. Extensive tuning of the tile size allows QUARK to achieve similar performance to OpenMP, while both of them achieve lower performance than MKL (Fig. 15). Fortunately, task priorities are already included in the OpenMP 4.5 standard (Nov 2015), and will almost certainly allow for closing the performance gap to MKL, which is currently roughly 15 %.

6 Conclusions

At this point, there are no major roadblocks for porting the PLASMA library to the OpenMP standard. Most routines, such as linear solvers and least squares solvers, can be translated in a straightforward fashion, while some routines, such as singular value solvers and eigenvalue solvers, may require some effort. Even though OpenMP does not provide an exact match for all of QUARK's functionality, at this point the benefits of the transition outweigh the costs of making the accommodations. This transition will create a very clean solution, based purely on OpenMP standards and standard dense linear algebra components, such as BLAS and LAPACK. This will dramatically improve the library from the software engineering standpoint, by providing excellent portability and substantially improving maintainability. The results of this work also indicate that the move will also improve performance.

7 Future Directions

The Intel Xeon Phi co-processor has been overlooked as a target for PLASMA. It has been labeled as an accelerator, programmed in the offload mode, and targeted by MAGMA. However, starting with the Knights Landing, and continuing to the Knights Hill and onwards, the Phi will be a self-hosted multicore processor, with OpenMP as its preferred programming paradigm. Under such circumstances, PLASMA is extremely well positioned to harvest the Phi's power through innovations in algorithms, scheduling and data layout.

Interesting opportunities for the PLASMA project may also be created by the penetration of the server market by multicore ARM processors, which have already reached the level of 48-cores per chip, 96-cores per board, as well as proliferation of tile mul-

tiprocessor architectures, such as Tilera's TILE64, Adapteva's Parallella, and similar designs.

References

1. Agullo, E., Bouwmeester, H., Dongarra, J., Kurzak, J., Langou, J., Rosenberg, L.: Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In: High Performance Computing for Computational Science—VECPAR 2010, pp. 129–138. Springer (2011)
2. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. In: Journal of Physics: Conference Series, vol. 180, p. 012037. IOP Publishing (2009)
3. Agullo, E., Hadri, B., Ltaief, H., Dongarrra, J.: 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, pp. 1–12. New York (2009)
4. Amdahl, G.M.: Validity of the single-processor approach to achieving large scale computing capabilities. In: AFIPS Conference Proceedings, vol. 30, pp. 483–485, Atlantic City, N.J., APR 18–20 1967. AFIPS Press, Reston (1967)
5. Anderson, E., Dongarra, J.: Implementation guide for LAPACK. Technical Report UT-CS-90-101, University of Tennessee, Computer Science Department, LAPACK Working Note 18 (1990)
6. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammerling, S., McKenney, A., et al.: LAPACK Users' Guide, vol. 9. SIAM, Philadelphia (1999)
7. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exp. **23**(2), 187–198 (2011)
8. Badia, R.M., Herrero, J.R., Labarta, J., Pérez, J.M., Quintana-Ortí, E.S., Quintana-Ortí, G.: Parallelizing dense and banded linear algebra libraries using SMPSs. Concurr. Comput.: Pract. Exp. **21**(18), 2438–2456 (2009)
9. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.J.: PaRSEC: exploiting heterogeneity to enhance scalability. Comput. Sci. Eng. **15**(6), 36–45 (2013)
10. Bouwmeester, H.: Tiled algorithms for matrix computations on multicore architectures. arXiv preprint arXiv:1303.3182 (2013)
11. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009)
12. Castaldo, A.M., Whaley, R.: Clint: acaling lapack panel operations using parallel cache assignment. In: ACM Sigplan Notices, vol. 45, pp. 223–232. ACM (2010)
13. Castaldo, A.M., Whaley, R.: Clint: scaling LAPACK panel operations using parallel cache assignment. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 223–232 (2010)
14. Donfack, S., Dongarra, J., Faverge, M., Gates, M., Kurzak, J., Luszczek, P., Yamazaki, I.: A survey of recent developments in parallel implementations of Gaussian elimination. Concurr. Comput.: Pract. Exp. **27**(5), 1292–1309 (2015)
15. Dongarra, J., Kurzak, J., Luszczek, P., Yamazaki, I.: PULSAR Users' Guide: Parallel Ultra-Light Systolic Array Runtime. Technical Report UT-EECS-14-733, EECS Department, University of Tennessee (2014)
16. Dongarra, J., Faverge, M., Ltaief, H., Luszczek, P.: Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. Concurr. Comput.: Pract. Exp. **26**(7), 1408–1431 (2014)
17. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. (TOMS) **16**(1), 1–17 (1990)
18. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OMPSS: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
19. Gao, G.R., Sterling, T., Stevens, R., Hereld, M., Weirong Z.: Parallex: a study of a new parallel computation model. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pp. 1–6. IEEE (2007)
20. Gustafson, J.L.: Reevaluating Amdahl's Law. Commun. ACM **31**(5), 532–533 (1988)

21. Gustavson, F., Karlsson, L., Kågström, B.: Parallel and cache-efficient in-place matrix storage format conversion. ACM Trans. Math. Softw. (TOMS) **38**(3), 17 (2012)
22. Gustavson, F.G.: Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM J. Res. Dev. **41**(6), 737–755 (1997)
23. Haidar, A., Kurzak, J., Luszczek, P.: An improved parallel singular value algorithm and its implementation for multicore hardware. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 90. ACM (2013)
24. Haidar, A., Ltaief, H., YarKhan, A., Dongarra, J.: Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. Concurr. Comput.: Pract. Exp. **24**(3), 305–321 (2012)
25. Kaiser, H., Brodowicz, M., Sterling, T.: Parallex an advanced parallel execution model for scaling-impaired applications. In: International Conference on Parallel Processing Workshops, 2009. ICPPW'09, pp. 394–401. IEEE (2009)
26. Kale, L.V., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. In: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, vol. 28, pp. 91–108. ACM (1993)
27. Kurzak, J., Buttari, A., Dongarra, J.: Solving systems of linear equations on the Cell processor using Cholesky factorization. IEEE Trans. Parallel Distrib. Syst. **19**(9), 1175–1186 (2008)
28. Kurzak, J., Ltaief, H., Dongarra, J., Badia, R.M.: Scheduling dense linear algebra operations on multicore processors. Concurr. Comput.: Pract. Exp. **22**(1), 15–44 (2010)
29. OpenMP Architecture Review Board: OpenMP Application Program Interface, version 4.5 edition (2015)
30. Pérez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: making it easier to program the Cell Broadband Engine processor. IBM J. Res. Dev. **51**(5), 593–604 (2007)
31. Pichon, G., Haidar, A., Faverge, M., Kurzak, J.: Divide and conquer symmetric tridiagonal eigensolver for multicore architectures. In: Proceedings of the International Parallel and Distributed Processing Symposium, pp. 51–60. IEEE (2015)
32. Quintana, E.S., Quintana, G., Sun, X., van de Geijn, R.: A note on parallel matrix inversion. SIAM J. Sci. Comput. **22**(5), 1762–1771 (2001)
33. Quintana-Ortí, G., Quintana-Ortí, E.S., Geijn, R.A., Van Zee, F.G., Chan, E.: Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Trans. Math. Softw. (TOMS) **36**(3), 14 (2009)
34. Tillenius, M.: Superglue: a shared memory framework using data versioning for dependency-aware task-based parallelization. SIAM J. Sci. Comput. **37**(6), C617–C642 (2015)
35. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: a language for distributed parallel scripting. Parallel Comput. **37**(9), 633–652 (2011)
36. YarKhan, A.: Dynamic Task Execution on Shared and Distributed Memory Architectures. PhD thesis, University of Tennessee (2012)
37. Zhao, Y., Hategan, M., Clifford, B., Foster, I., Von Laszewski, G., Nefedova, V., Raicu, I., Stef-Praun, T., Wilde, M.: Swift: fast, reliable, loosely coupled parallel computation. In: Services, 2007 IEEE Congress on, pp. 199–206. IEEE (2007)