Leveraging Task-Parallelism in Message-Passing Dense Matrix Factorizations using SMPSs

Rosa M. Badia[a], Alberto F. Martín[b], Enrique S. Quintana-Ortí[c], Ruymán Reyes[d]

[a] *Barcelona Supercomputing Center (BSC-CNS), 08034–Barcelona, Spain. Artificial Intelligence Research Institute (IIAA), Spanish National Research Council (CSIC).* rosa.m.badia@bsc.es

[b] *Centre Internacional de Mètodes Numèrics en Enginyeria (CIMNE), Universitat Politècnica de Catalunya, 08034–Barcelona, Spain.* amartin@cimne.upc.edu

[c] *Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12.071–Castellón, Spain.* quintana@uji.es

[d] *Edinburgh Parallel Computing Centre, University of Edinburgh,* rreyesc@epcc.ed.ac.uk

Abstract

In this paper, we investigate how to exploit task-parallelism during the execution of the Cholesky factorization on clusters of multicore processors with the SMPSs programming model. Our analysis reveals that the major difficulties in adapting the code for this operation in ScaLAPACK to SMPSs lie in algorithmic restrictions and the semantics of the SMPSs programming model, but also that they both can be overcome with a limited programming effort. The experimental results report considerable gains in performance and scalability of the routine parallelized with SMPSs when compared with conventional approaches to execute the original ScaLAPACK implementation in parallel as well as two recent message-passing routines for this operation.

In summary, our study opens the door to the possibility of reusing message-passing legacy codes/libraries for linear algebra, by introducing up-to-date techniques like dynamic out-of-order scheduling that significantly upgrade their performance, while avoiding a costly rewrite/reimplementation.

Keywords: Task parallelism, message-passing numerical libraries, linear algebra, clusters of multi-core processors

1. Introduction

Linear systems of equations

$$Ax = b, \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$ are given, and $x \in \mathbb{R}^n$ is the sought-after solution, are ubiquitous in scientific and engineering applications. When the coefficient matrix A in these problems is dense, the most efficient approach for their solution consists in a direct method that first decomposes A into a simpler form, and then solves the corresponding transformed system. A well-known decomposition for the solution of (1), when the coefficient matrix A is symmetric positive definite (s.p.d.), is the Cholesky factorization. Computing this factorization requires a cubic number of floating-point arithmetic operations (flops) in the problem dimension, compared with the quadratic computational cost that takes the subsequent solution of the factorized system, and therefore the first stage is crucial for the efficient solution of s.p.d. linear systems [1].

Large-scale instances of (1) with $n \sim O(100,000)$ and s.p.d. matrix A appear, e.g., in molecular dynamics, electromagnetism and space geodesy, usually requiring a cluster of computers (nodes) for their solution. ScaLAPACK [2] and PLAPACK [3] are two message-passing, dense linear algebra libraries for parallel distributed-memory architectures in general and clusters in particular. These packages were developed in the mid 90s, when a single processor (CPU) per node was mainstream, and thus are not well-suited to leverage the intra-node concurrency of as-of-today clusters nodes equipped with multicore technology. In particular, a feasible approach to exploit the two levels of hardware parallelism (inter-node and intra-node) in clusters of multicore processors from ScaLAPACK/PLAPACK is to employ one MPI rank (process) per core; i.e, use a pure MPI approach, with as many MPI ranks (i.e., processes) per node as cores in the platform. Alternatively, one can place one MPI rank per node and rely on a multithreaded implementation of the BLAS (hybrid MPI/MT-BLAS) to exploit the intra-node hardware parallelism. These two strategies provide a plain, though sub-optimal, path to leverage the hybrid architecture of clusters of multicore processors from existing dense message-passing numerical linear algebra libraries. Specifically, the problem of these two solutions is that they unnecessarily constrain the level of concurrency of the operation by imposing a strict ordering of the

computations. Although this drawback can be partially alleviated via advanced look-ahead techniques [4, 5], this greatly complicates programming. Furthermore, the introduction of one additional variable, the depth of the look-ahead, in the modeling/experimentation turns the optimization of the resulting code even more challenging.

SMPSs [6] is a portable and flexible framework to exploit task-level parallelism on shared-memory multiprocessors (including multicore processors). The framework is composed of a few OpenMP-like compiler directives which allow the programmer to annotate certain functions (routines) in the code as tasks, a source-to-source compiler, and a runtime that detects dependencies among tasks and efficiently issues them for execution in due order, attaining dynamic load balancing with an out-of-order schedule. A significant part of the benefits observed from the application of SMPSs to dense linear algebra operations are similar to those obtained with a carefully tuned look-ahead; on the other hand, tackling these problems with SMPSs requires a very moderate effort from the programmer [7].

In this paper we describe our experience using SMPSs to leverage multithreaded parallelism from within message-passing implementations of dense linear algebra kernels, using the implementation of the Cholesky factorization in ScaLAPACK (routine `pdpotrf`) as a case study. The major contributions of this paper are summarized as follows:

- We illustrate the difficulties encountered during the parallelization of the Cholesky factorization in ScaLAPACK, e.g. due to algorithmic restrictions embedded in the library, the data layout, or the semantics of the SMPSs programming model.

- We report high performance as well as remarkable scalability on a large facility which clearly outperform those obtained with the conventional pure MPI and hybrid MPI/MT-BLAS approaches. We also compare our solution with two recent alternatives to ScaLAPACK [8, 9], showing the superior performance of the MPI/SMPSs routine against those modern codes as well.

- We provide experimental evidence that the exploitation of task-parallelism enabled by the dependency-aware out-of-order execution of tasks intrinsic to SMPSs is the source of this superior performance and scalability.

- Our exercise provides a practical demonstration that the MPI/SMPSs

framework applies to a significant part of the contents of ScaLAPACK-/PLAPACK (and possibly also to task-parallel operations in other libraries). Furthermore, we show that advanced techniques like dynamic out-of-order scheduling (and, as a result, adaptive look-ahead) can be integrated into of these libraries, without requiring a costly rewrite of their contents.

The rest of the paper is structured as follows. We first elaborate a compact discussion of related work in Section 3. Next, we briefly revisit the SMPSs framework for the exploitation of task-level parallelism on multicore architectures in Section 3. The main contributions of this paper are contained in Sections 4 to 6. These parts address the parallelization of message-passing routine in ScaLAPACK for the Cholesky factorization using SMPSs; report an experimental comparison of the resulting code with conventional parallelization approaches and the more modern ones in [8, 9]; and apply the `Paraver` [10] performance analyzer in order to comprehensively demonstrate that the exploitation of task-parallelism enabled by SMPSs is the source of superior performance and scalability over conventional approaches. Finally, a few remarks close the paper in Section 7.

2. Related Work

Cilk [11] is likely among the pioneer projects to target the exploitation of general-purpose task-level parallelism with dependencies on multiprocessor systems. From then on, several other projects have adopted a similar approach for task-parallel general-purpose applications (e.g., OmpSs and its instances CellSs [12], SMPSs [6] and GPUSs [13]; StarPU [14], Harmony [15], XKaapi [16], etc.) as well as for the specific domain of linear algebra (SuperMatrix+`libflame` [17] and PLASMA/MAGMA [18, 19]).

The afore-mentioned projects have clearly demonstrated the advantages of decomposing dense linear algebra operations into a collection of fine-granularity tasks, express the dependencies among them as a directed acyclic graph (DAG), and schedule the execution of the tasks using this information. The application of the same approach to the iterative and direct solution of sparse linear systems on multi-core processors has also demonstrated its benefits in [20] and [21], respectively.

ScaLAPACK and PLAPACK contain message-passing routines for dense linear algebra operations, but mimic their "sequential" counterparts (LAPACK [22] and `libflame` [23] respectively), in that they do not exploit all

the concurrency intrinsic to the operation/DAG. Elemental [9] is a modern replacement for PLAPACK, but with the same restriction. The number of approaches that aim at exploiting a more flexible scheduling on message-passing environments is more limited. In [8] the authors introduce a depth-one look-ahead message-passing Cholesky which enables effective overlapping of communication/computation. Although static and dynamic scheduling variants of the algorithm are designed (with the latter aiming at removing the so-called spurious synchronizations from the former), the opportunities to leverage the operation concurrency are limited in both cases by a fixed look-ahead depth hard-coded in the algorithm/code itself. DAGuE [24] goes one step further and adopts dynamic scheduling to enhance the extraction of parallelism. In particular, this tool is a DAG-based scheduling framework where the nodes represent sequential tasks and the arcs correspond to data movements. Furthermore, in order to build this graph, DAGuE includes a specific language to express how the flow of data circulates between kernels. Finally, a portable but not scalable approach is investigated in [25] using SuperMatrix and `libflame` to execute dense linear algebra algorithms directly on (small scale) message-passing environments.

Our effort with MPI/SMPSs departs from previous approaches in that we commence with an existing legacy code, in our case the implementation of the Cholesky factorization in ScaLAPACK, and study what are the changes that need to be introduced into this code so as to obtain an efficient task-parallel dynamic out-of-order execution. The benefits of our approach lie thus in the reuse of existing, legacy code that it enables. A related exercise was performed in [26] using the implementation of the LU factorization with partial pivoting in the HPL (LINPACK) benchmark. However, the implementation of the Cholesky factorization in ScaLAPACK is a much more challenging case, mainly due to the symmetric nature of the operation.

3. A Brief Review of SMPSs

StarSs is an active project that targets multiple different hardware platforms (Grids; multicore architectures and shared-memory multiprocessors; platforms with multiple hardware accelerators: GPUs, Cell B.E., Clearspeed boards; heterogeneous systems, etc.) with distinct implementations of the framework. SMPSs is an instance of the StarSs framework tailored for shared-memory multiprocessors. It combines a language with a much reduced number of OpenMP-like compiler directives, a source-to-source compiler, and

a runtime system to leverage task-level parallelism in sequential codes. In SMPSs, the programmer employs compiler directives to annotate certain routines (functions) appearing in the code as tasks, indicating the directionality of their operands (input, output or input/output) by means of clauses. The runtime exploits task-level parallelism by decomposing the code (transformed by the source-to-source compiler) into a number tasks during the execution, dynamically identifying dependencies among these, and issuing *ready tasks* (i.e., those with all dependencies satisfied) out-of-order for execution in the cores of the system.

Listing 1 illustrates the parallelization of a sequential blocked routine, `dblock_chol`, that computes the Cholesky factorization $\mathtt{A} = \mathtt{U}^T\mathtt{U}$ of an $\mathtt{n} \times \mathtt{n}$ matrix `A` with entries stored in column-major order, overwriting the upper triangular part of `A` with the entries of the triangular factor `U`. For simplicity, we assume that the matrix size is an integer multiple of the block size `bs`. Routines `dpotrf` (Cholesky factorization), `dtrsm` (triangular system solve), `dgemm` (matrix-matrix product), `dsyrk` (symmetric rank-`bs` update) simply correspond to well-known computational kernels from LAPACK (the former) and BLAS (the latter three), with the functionality specified as comments at the beginning of the corresponding routine. In principle, it could appear surprising that, in order to compute the Cholesky factorization of a matrix `A`, we recursively call a kernel (`dpotrf`) that performs the same computation, but on a smaller chunk of data. However, this is a usual technique in blocked algorithms which aim at leveraging the multi-layered organization of the memory subsystem by amortizing the cost of accessing data in the main memory with a large number of flops.

```
1  #define Ad(i,j) A[(j-1)*n+(i-1)]
2
3  void dblock_chol( int n, int bs, double A[] ){
4   int i, j, k;
5
6   for (k=1; k<=n; k+=bs)
7     // Perform unblocked Cholesky factorization on k-th block
8     dtile_chol( bs, &Ad(k,k), n );
9
10    if (k+bs<=n){
11      // Form the row panel of U using the triangular solver
12      for (j=k+bs; j<=n; j+=bs)
13        dtile_trsm( bs, &Ad(k,k), &Ad(k,j), n );
14
15      // Update the trailing submatrix, A := A - U^T * U
16      for (j=k+bs; j<=n; j+=bs){
17        for (i=1; i<=j; i+=bs)
18          dtile_gemm( bs, &Ad(k,i), &Ad(k,j), &Ad(i,j), n );
```

```
19          dtile_syrk( bs, &Ad(k,j), &Ad(j,j), n );
20        }
21      }
22  }
23
24  #pragma css task input( bs, ldm ) inout( A[1] )
25  void dtile_chol( int bs, double A[], int ldm )
26  {
27   // Obtain the Cholesky factorization A = U^T U, where A is bs x bs
28   int info;
29
30   dpotrf( "Upper", &bs, A, &ldm, &info );
31  }
32
33  #pragma css task input( bs, A[1], ldm ) inout( B[1] )
34  void dtile_trsm( int bs, double A[], double B[], int ldm )
35  {
36   // B := A^-1 * B, where A and B are both bs x bs, and A is considered
37   // to be upper triangular
38   double done = 1.0;
39
40   dtrsm( "Left", "Upper", "Transpose", "Non-unit",
41          &bs, &bs, &done, A, &ldm,
42                    B, &ldm );
43  }
44
45  #pragma css task input( bs, A[1], B[1], ldm ) inout( C[1] )
46  void dtile_gemm( int bs, double A[], double B[], double C[], int ldm )
47  {
48   // C := C - A^T * B, where A, B and C are all bs x bs
49   double dmone = -1.0, done = 1.0;
50
51   dgemm( "Transpose", "No transpose",
52          &bs, &bs, &bs, &dmone, A, &ldm,
53                                 B, &ldm,
54                         &done,  C, &ldm );
55  }
56
57  #pragma css task input( bs, A[1], ldm ) inout( C[1] )
58  void dtile_syrk( int bs, double A[], double C[], int ldm )
59  {
60   // C := C - A^T * A, where A and C are both bs x bs
61   double dmone = -1.0, done = 1.0;
62
63   dsyrk( "Upper", "Transpose",
64          &bs, &bs, &dmone, A, &ldm,
65                    &done,  C, &ldm );
66  }
```

Listing 1: Parallelization of blocked right-looking Cholesky factorization using SMPSs.

In order to parallelize this code with SMPSs, the programmer employs the `#pragma css task` directive to mark which functions will become tasks during the execution of the code. The associated clauses `input`, (`output`,)

and `inout` specify the directionality of the function arguments, which help the runtime to capture all data dependencies among tasks. For example, in the invocations to `dtile_chol`, the block size `bs` is an input (not modified inside the routine), while `A` is both an input and an output (this block has to be factorized and the results overwrite the corresponding entries). In the blocked code for the Cholesky factorization, we exploit that the blocks involved in the kernel calls do not overlap. Thus, we use the top-left (i.e., first) entry of each $\mathtt{bs} \times \mathtt{bs}$ block as a "sentinel" (representant) for all its elements, relying on the SMPSs runtime to keep track of dependencies among tasks based solely on this first entry. It is precisely this non-overlapped property that enables the application of the technique of sentinels [27]. Although there exists a more formal solution which explicitly recognizes the real dimensions of a block, see in particular [28], we select the one described above for its simplicity.

Apart from directionality clauses, SMPSs also offers the `highpriority` clause. This clause gives a hint to the runtime system about the "urgency" of scheduling a given task for execution (as soon as its dependencies are satisfied). Tasks marked with the `highpriority` clause are scheduled for execution earlier than tasks without this clause. This mechanism allows a programmer with global understanding of the critical computations to influence the actual schedule.

Figure 1 shows a DAG that captures the tasks (nodes) and data dependencies (arcs) intrinsic to the execution of routine `dblock_chol`, when applied to a matrix composed of 4×4 blocks (of dimension $\mathtt{bs} \times \mathtt{bs}$ each). There are 4 tasks of type C (`dtile_chol`), 6 of types T and S each (`dtile_trsm` and `dtile_syrk` respectively), and 4 of type G (`dtile_gemm`). (For matrices with a large number of blocks, though, tasks of type G dominate, their abundance being of $\mathcal{O}(\mathtt{b}^3)$, with the number of blocks $\mathtt{b} = \mathtt{n}/\mathtt{bs}$; tasks of types Tand S appear $\mathcal{O}(\mathtt{b}^2)$; and the number of tasks of type C is $\mathcal{O}(\mathtt{b})$.) The subindices associated to a task (e.g., C_{11}) indicate the block of matrix `A` the task overwrites, and the colors distinguish between tasks belonging to different iterations of loop `k` in routine `dblock_chol`. It is this set of dependencies that expresses the true concurrency in the computation of the Cholesky factorization. The SMPSs runtime derives this information from the annotated version of the code to maximize the concurrent execution of the routine, dynamically balancing the load, while fulfilling all dependencies intrinsic to the operation.

One appealing instance of the StarSs programming model is MPI/SMPSs, which provides specific support for MPI applications [26]. In this particular

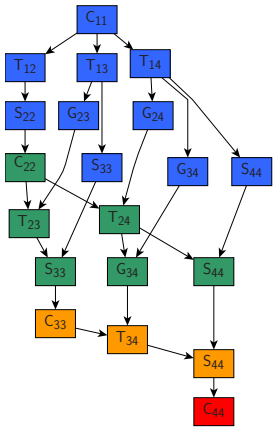


Figure 1: DAG with the tasks and data dependencies of routine `dblock_chol` applied to a 4×4 matrix, with blocks of dimension `bs` $\times$ `bs` each.

version there exists the possibility of embedding calls to MPI primitives as SMPSs tasks, so that communication can be overlapped with computation transparently to the programmer. To indicate that a particular task comprises a communication (e.g., a task that invokes a BLACS/MPI primitive), in MPI/SMPSs the developer employs the `target device` clause, with the `comm_thread` option, (i.e., `#pragma css task target device (comm_thread)`). A separate thread devoted to communication is created dynamically by the runtime and those dependencies required to enforce the correct communication order are automatically added. Depending on the target platform, the SMPSs runtime may also configure the priority of the communication thread dynamically, in order to enforce communication to occur as soon as possible.

4. The Cholesky Factorization in ScaLAPACK

4.1. Review of ScaLAPACK routine `pdpotrf`

Routine `pdpotrf` encodes a message-passing algorithm to compute the upper (or lower) triangular Cholesky factor of an s.p.d. matrix. The entries of the matrix are distributed following a block-cyclic 2D layout among a `p` $\times$ `q` (logical) grid of processes. Thus, `A` is partitioned into square blocks, of size `ds` (the data distribution blocking factor), which are then mapped to the 2D process grid in a block-cyclic fashion. The distribution layout determines the communication pattern of the message-passing algorithm, which transfers

data via calls to BLACS (usually built on top of MPI). Standard kernels from LAPACK and BLAS are employed in the routine for local vector and matrix operations.

Listing 2 contains an excerpt of code that mimics the computations performed in the main loop of the ScaLAPACK routine for the Cholesky factorization. `descA` is a descriptor containing information on the data distribution, the grid shape, etc. (See [2] for details.) For simplicity, we consider hereafter that the first element of `A` is stored in the top-left process of a square $\mathtt{q} \times \mathtt{q}$ grid.

```
1   // ...
2   for (k=1; k<=n; k+=ds){
3     // Perform unblocked Cholesky factorization on k-th block
4     pdpotf2( "Upper", &ds, A, &k, &k, &descA, &info );
5     // ...
6
7     if (k+ds<=n){
8       m = n-k-ds+1;
9       j = k+ds;
10
11      // Form the row panel of U using the triangular solver
12      pdtrsm( "Left", "Upper", "Transpose", "Non-unit",
13              &ds, &m, &done, A, &k, &k, &descA,
14                              A, &k, &j, &descA );
15
16      // Update the trailing submatrix, A := A - U^T * U
17      pdsyrk( "Upper", "Transpose",
18              &m, &m, &dmone, A, &k, &j, &descA,
19                      &done,  A, &j, &j, &descA );
20    }
21  }
```

Listing 2: Simplified version of ScaLAPACK routine `pdpotrf`.

Routines `pdpotf2`, `pdtrsm`, and `pdsyrk` encode message-passing kernels to compute, respectively, the Cholesky factorization, triangular system solve, and symmetric rank−`ds` update of a distributed (sub)matrix. Compared with the code in Listing 1, the invocation of `pdtrsm` in the ScaLAPACK code fragment performs all the computations in the loop `j` in lines 12–13 (several invocations of `dtile_trsm`); and the invocation of `pdsyrk` those of the nested loops `j` and `i` in lines 16–20 (multiple invocations of `dtile_gemm` and `dtile_syrk`).

Let us now consider each one of these message-passing kernels and the implications from the viewpoint of the communications for a given iteration, namely `k`. Define $\mathtt{r} = (\mathtt{k}-1)/\mathtt{ds}+1$ and assume, for simplicity, that the problem size is an integer multiple of the distribution block size; i.e., $\mathtt{n} = \mathtt{d}\cdot\mathtt{ds}$ with `d` an integer; consider also the following partition of the matrix `A` into

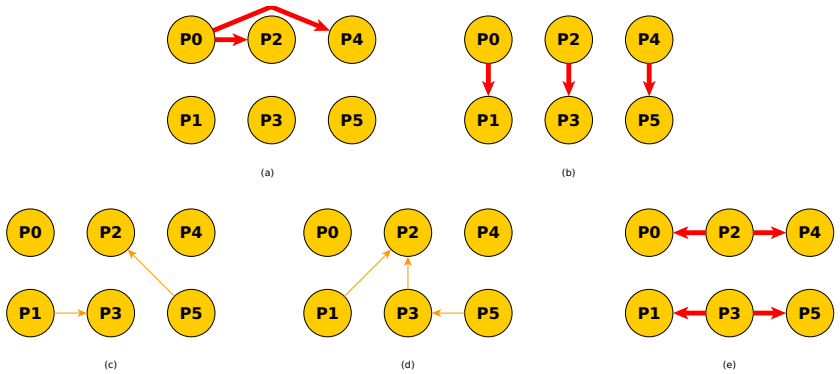


Figure 2: Communication pattern during an iteration of the Cholesky factorization in ScaLAPACK for a 2×3 process grid. (a) The Cholesky factor of the leading diagonal block, computed in `dpotrf`, is broadcast rowwise. (b) When the triangular systems are solved, the computed panel is broadcast columnwise. In (c) and (d), local portions of a new block are redistributed, using point-to-point operations, within the second column of processes, where they are transposed. (e) Finally, processes in the second column broadcast rowwise the transposed panel, in preparation for the trailing submatrix update.

a grid of $\mathtt{d} \times \mathtt{d}$ blocks,

$$\mathtt{A} \rightarrow [\mathtt{A}_{ij}] \rightarrow \begin{bmatrix} \mathtt{A}_{1,1} & \cdots & \mathtt{A}_{1,\mathtt{r}} & \cdots & \mathtt{A}_{1,\mathtt{d}} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathtt{A}_{\mathtt{r},1} & \cdots & \mathtt{A}_{\mathtt{r},\mathtt{r}} & \cdots & \mathtt{A}_{\mathtt{r},\mathtt{d}} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathtt{A}_{\mathtt{d},1} & \cdots & \mathtt{A}_{\mathtt{d},\mathtt{r}} & \cdots & \mathtt{A}_{\mathtt{d},\mathtt{d}} \end{bmatrix}, \tag{2}$$

with each block $\mathtt{A}_{ij}$ of dimension $\mathtt{ds} \times \mathtt{ds}$.

Routine `pdpotf2`. This call obtains the Cholesky factorization of block $\mathtt{A}_{\mathtt{r},\mathtt{r}}$. Given that all blocks are $\mathtt{ds} \times \mathtt{ds}$, these data are local to a single process of the grid and, therefore, the computation can be performed in that process via a single invocation of kernel `dpotrf`.

Routine `pdtrsm`. This call solves a triangular system with the upper triangular factor computed by `dpotrf` (i.e., the Cholesky factor of $\mathtt{A}_{\mathtt{r},\mathtt{r}}$) as the coefficient matrix, and for the right-hand side blocks $\mathtt{A}_{\mathtt{r},\mathtt{r}+1}, \mathtt{A}_{\mathtt{r},\mathtt{r}+2}, \ldots, \mathtt{A}_{\mathtt{r},\mathtt{d}}$,

which are distributed among a row of processes of the grid. Thus, to perform this operation, the triangular factor is first broadcast from the process that contains it to the remaining processes in the same row of the grid. All processes can then operate concurrently, obtaining their local part of the right-hand side via a single call to `dtrsm`. For example, given the block cyclic distribution of data, the process that owns $\mathtt{A}_{\mathtt{r},\mathtt{r}+1}$ will solve a triangular system with the right-hand side matrix composed of blocks $\left[\mathtt{A}_{\mathtt{r},\mathtt{r}+1}, \mathtt{A}_{\mathtt{r},\mathtt{r}+\mathtt{q}+1}, \mathtt{A}_{\mathtt{r},\mathtt{r}+2\mathtt{q}+1}, \ldots\right]$.

Routine `pdsyrk`. Internally, `pdsyrk` encodes two distinct algorithmic variants, `pdsyrkAC` and `pdsyrkA`. Here we will focus on this second routine as, except for the last iterations, this is the common case invoked from the ScaLAPACK routine for the Cholesky factorization.

Routine `pdsyrkA` performs the symmetric update of the trailing submatrix

$$\mathtt{C} := \mathtt{C} - \mathtt{B}^T\mathtt{B} \equiv$$

$$\begin{bmatrix} \mathtt{A}_{\mathtt{r}+1,\mathtt{r}+1} & \mathtt{A}_{\mathtt{r}+1,\mathtt{r}+2} & \cdots & \mathtt{A}_{\mathtt{r}+1,\mathtt{d}} \\ \mathtt{A}_{\mathtt{r}+2,\mathtt{r}+1} & \mathtt{A}_{\mathtt{r}+2,\mathtt{r}+2} & \cdots & \mathtt{A}_{\mathtt{r}+2,\mathtt{d}} \\ \vdots & \vdots & \ddots & \vdots \\ \mathtt{A}_{\mathtt{d},\mathtt{r}+1} & \mathtt{A}_{\mathtt{d},\mathtt{r}+2} & \cdots & \mathtt{A}_{\mathtt{d},\mathtt{d}} \end{bmatrix} := \begin{bmatrix} \mathtt{A}_{\mathtt{r}+1,\mathtt{r}+1} & \mathtt{A}_{\mathtt{r}+1,\mathtt{r}+2} & \cdots & \mathtt{A}_{\mathtt{r}+1,\mathtt{d}} \\ \mathtt{A}_{\mathtt{r}+2,\mathtt{r}+1} & \mathtt{A}_{\mathtt{r}+2,\mathtt{r}+2} & \cdots & \mathtt{A}_{\mathtt{r}+2,\mathtt{d}} \\ \vdots & \vdots & \ddots & \vdots \\ \mathtt{A}_{\mathtt{d},\mathtt{r}+1} & \mathtt{A}_{\mathtt{d},\mathtt{r}+2} & \cdots & \mathtt{A}_{\mathtt{d},\mathtt{d}} \end{bmatrix} - \begin{bmatrix} \mathtt{A}^T_{\mathtt{r},\mathtt{r}+1} \\ \mathtt{A}^T_{\mathtt{r},\mathtt{r}+2} \\ \vdots \\ \mathtt{A}^T_{\mathtt{r},\mathtt{d}} \end{bmatrix} \begin{bmatrix} \mathtt{A}_{\mathtt{r},\mathtt{r}+1} & \mathtt{A}_{\mathtt{r},\mathtt{r}+2} & \cdots & \mathtt{A}_{\mathtt{r},\mathtt{d}} \end{bmatrix} \tag{3}$$

as follows. The processes that own part of the `r`-th block row of the matrix, (renamed as `B`,) first broadcast their local portions column-wise (i.e., inside the same column of processes of the grid). Then, `B` is transposed onto a column of processes, yielding $\mathtt{B}^T$; and these processes next broadcast row-wise their local parts of it. Finally, all the processes in the grid update their local portions of `C` with respect to $\mathtt{B}^T$ and `B` as in (3).

For general grids ($\mathtt{p} \neq \mathtt{q}$), the transposition of `B` from a row of processes onto a column of processes essentially requires several point-to-point communications. The blocks of `B` to be exchanged are packed by the source process into temporary buffers, and unpacked/transposed into an auxiliary workspace in the destination. The communication patterns described above for these three stages of the iteration are illustrated in Figure 2. For details,

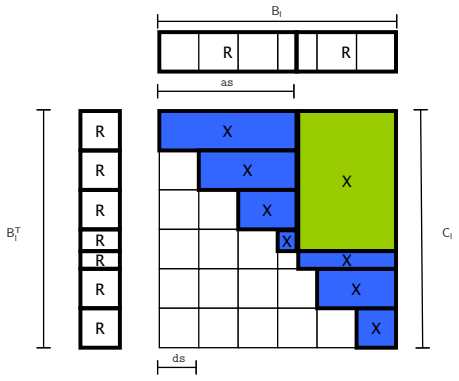


Figure 3: Aggregation of level 3 BLAS calls in `pdsyrkA` from the perspective of a single process in the grid. Blocks labelled with an "`R`" are only read, while blocks annotated with an "`X`" are both read and written. The update of the green region is done via a call to `dgemm`, while blue regions are updated via invocations to `dsyrk`.

see [29].

The parallel update of `C` in (3) decouples the algorithmic blocking factor, `as`, from the data distribution blocking factor, `ds`, in order to increase the granularity of the invocations to the level-3 BLAS that are necessary to perform the local computations. The diagonal blocks induced by this algorithmic partitioning are updated via several fine-grained calls to the level-3 BLAS, possibly spanning more than one process. The update of the off-diagonal blocks, on the other hand, may span more than one process but involves a single coarse-grain invocation to the level 3 BLAS per process. This is captured in Figure 3, which shows the operations performed by a single process to update the local portions of `C` with the received parts of $\mathtt{B}^T$ and `B`. Local parts of `C` and `B` are referenced in the figure as $\mathtt{C}_l$ and $\mathtt{B}_l$ respectively.

4.2. Taskification of the computational layer

We next describe how the computations ocurring during the message-passing Cholesky factorization can be decomposed into a collection of tasks (and dependencies) of finer granularity, in a process similar to those proposed in [17, 18] to obtain *algorithms-by-blocks* or *tile-based algorithms* for multicore platforms. We point out the differences though, in that we commence with a message-passing routine and, as a result, the DAG that is obtained locally (at the node level) not only includes communication tasks, but is also quite

different from that obtained from the "sequential" (i.e., non message-passing) Cholesky factorization.

4.2.1. Capturing dependencies

Our parallelization of the Cholesky factorization routine using the MPI/SMPSs programming model is conditioned by two restrictions[1] of this programming model. First, dependencies among tasks are identified by comparing the base-address of the corresponding operands, so that memory *regions* corresponding to different operands cannot overlap. Second, data for matrix/vector operands accessed by tasks must be stored contiguously in memory. This latter constraint has important implications in our case because ScaLAPACK, BLACS, MPI and BLAS all employ column-major storage for the data. Therefore, operands involved by the computational and communication kernels of these libraries used from within `pdpotrf` are actually spread in discontiguous regions of the memory. Among the different solutions that were considered, we decided to adopt the usage of sentinels [27], as it is applicable to most dense linear algebra codes and avoids significant recoding; see [7] and our discussion below.

Notice that the use of sentinels implies that we must enforce that (regions in memory corresponding to) distinct operands accessed by tasks are separated, with no overlapping among them, though they may still lie in discontiguous regions. In the next two subsections, we will describe how we adapted the codes to enforce this in the message-passing kernels from ScaLAPACK.

4.2.2. Local computations

The parallelization of the computations performed within `pdpotrf` and `pdtrsm` using SMPSs is simple. For the first one, we simply define it as a task, and use its first entry as a representant for the rest of the block. In principle, we could have also done the same with `pdtrsm`, with one single task per process to perform the local triangular system solve. However, to increase the degree of concurrency and attain better load balancing inside (multicore) nodes, we decided to subdivide the local triangular solve into multiple ones, one for each `ds` $\times$ `ds` block of $\mathtt{A}_{\mathtt{r},\mathtt{r}+1}, \mathtt{A}_{\mathtt{r},\mathtt{r}+2}, \ldots, \mathtt{A}_{\mathtt{r},\mathtt{d}}$. Consider, e.g., the computations local to the process that owns $\mathtt{A}_{\mathtt{r},\mathtt{r}+1}$. Instead of a single call

[1]While other instances of StarSs do not suffer from these limitations, they cannot be used in the parallelization of `pdsyrkA` as they provide no support for MPI applications [28].

to the kernel `dtrsm` to compute the triangular system with the right-hand side $\left[\mathtt{A}_{\mathtt{r},\mathtt{r}+1}, \mathtt{A}_{\mathtt{r},\mathtt{r}+\mathtt{q}+1}, \mathtt{A}_{\mathtt{r},\mathtt{r}+2\mathtt{q}+1}, \ldots\right]$, this process will invoke `dtrsm` multiple times, one for each of its local $\mathtt{ds} \times \mathtt{ds}$ blocks. Each one of these calls then becomes a task and the top-left entry of the corresponding block acts as a representant for it.

From the implementation point of view, in order to semi-automatically attain the decomposition of `pdtrsm` into tasks of finer granularity, the original call to `dtrsm` was replaced by a wrapper which decomposes the code into tasks and invokes the actual level 3 BLAS to perform the corresponding computations (solves) on the appropriate blocks. Proceeding in this manner, we can taskify the code without rewriting the original routines, by simply linking in the intermediate wrapper for the `dtrsm` kernel.

The update of the trailing submatrix from within routine `pdsyrkA` is more challenging as this routine decouples the algorithmic and distribution block sizes with the purpose of casting all local computations in terms of a reduced number of coarse-grain level 3 BLAS; see Figure 3. This was natural when ScaLAPACK was designed, as in the mid 90s most clusters were equipped with single-processor nodes. It also made sense when these systems incorporated a very reduced number of processors per node that could be more efficiently exploited using MT-BLAS. Nevertheless, this approach unnecessarily constrains concurrency in current multicore nodes.

In order to parallelize the local computations inside `pdsyrkA`, one could simply try to encapsulate the calls to the level 3 BLAS from within `pdsyrkA` as SMPSs tasks. However, as we describe next, this naive approach unnecessarily limits the concurrency of the update and, besides, has the potential of yielding an unbalanced load distribution.

Let us deal with the first problem which, unless corrected, may result in the SMPSs runtime not taking advantage of all the concurrency intrinsic to the symmetric update. Inside an MPI process, routine `pdsyrkA` attempts to maximize the size of the operands by grouping together several local invocations to BLAS kernels. Thus, local updates to the off-diagonal blocks are aggregated into one large `dgemm`, while diagonal blocks are updated via repeated invocations to the `dsyrk` kernel; see Figure 4 (a). In principle, the update of each one of the blocks in $\mathtt{C}_\mathtt{l}$ marked with an "X" corresponds to an SMPSs task. The problem here is that the algorithmic partitioning used for the parallel update of `C` does not need to be "aligned" with the data distribution partitioning. Thus, employing the first (top-left) entry of the block of $\mathtt{C}_\mathtt{l}$ highlighted with a red circle as a sentinel for the two `dsyrk` operations

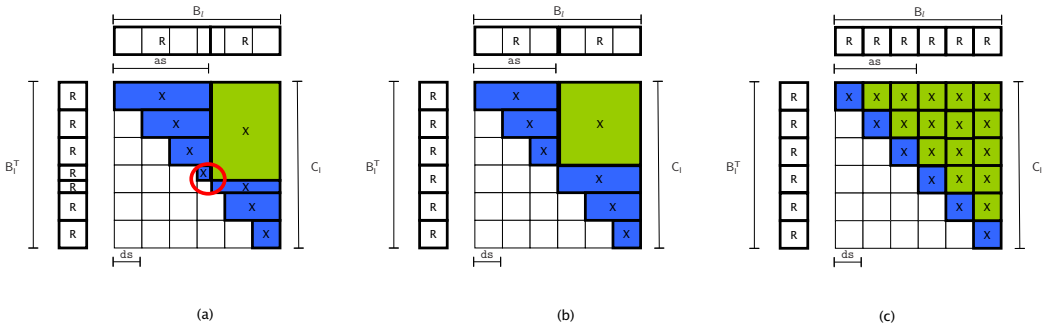


Figure 4: Example of the update of local blocks performed in a single process during the execution of `pdsyrkA`. (a) Update with unaligned partitionings shows an overlapped region (highlighted in red) which leads to a false dependence detection when using SMPSs. (b) Aforementioned false dependence detection when using SMPSs is easily solved using aligned partitionings. (c) Taskification of the updates yields a more balanced distribution.

that update part of its data leads to erroneously identifying a dependency between these two tasks. In other words, the base-address match performed by the runtime detects a false data dependency between these two operations, serializes their execution and, in consequence, does not leverage all the parallelism intrinsic to the update. To deal with this problem, we propose to align both partitionings by readjusting the algorithmic block size `as` to the closest integer multiple of the data distribution block size. Figure 4 (b) illustrates how the realignment solves the problem of false dependencies. In practice, the optimal distribution block size is relatively small (in our experiments, in the range of 128 to 384) so that we do not expect that this minor restriction on the algorithmic block size poses a major obstacle to attain high performance.

Consider now the second problem, related with the potential load unbalancing, and which already appeared in the triangular system solve. Here we apply the same solution, decomposing the coarse-grain calls of the local update into multiple finer-grain tasks. To do this, we develop wrappers for the `dgemm` and `dsyrk` kernels so as to avoid a major rewrite of the routine.

The result from these two techniques is illustrated in Figure 4 (c): no overlapping occurs now, which allows the runtime to properly identify dependencies among the tasks which operate on these blocks; and the granularity of tasks is much smaller and homogeneous, enabling a balanced workload distribution. (The computational cost of a task is proportional to the size of

its operands, of $\mathcal{O}(\texttt{ds}^3)$ for all types of tasks.)

Although it could be desirable that the granularity of the taskifying decomposition was determined independently of the data distribution blocking factor, the packing/unpacking operations for the transposition of row panels are blocked conformally to the data distribution partitioning. Therefore, in order to correctly track data dependencies from within the MPI/SMPSs runtime, the taskifying grain must be aligned with the data distribution blocking factor.

Let us finally remark that the tasks resulting from the taskification of `pdpotrf` and `pdtrsm` were marked with the `highpriority` clause. The same applies for those local symmetric rank-k updates and matrix-matrix multiplications within `pdsyrkA` required for the computation of the next panel (i.e., those that update the first block row of the trailing submatrix). With this, we pursue an adaptive, dynamic look-ahead effect that accelerates the execution of the critical path of the parallel algorithm. The performance benefit of this strategy will be analyzed in detail in Section 6.

4.3. Taskification of the communication layer

Two types of BLACS primitives are invoked from the `pdpotrf` routine: point-to-point messages (exchanged to transpose `B`) and broadcasts (to propagate copies of certain blocks within the same row/column of processors); see subsection 4.1. The taskification strategy discussed next for the point-to-point BLACS send/receive calls was also used for the BLACS broadcast primitives.

To identify data dependencies among communication and computation tasks in MPI/SMPSs, the former need to be blocked conformally with the data distribution partitioning. Although this can be achieved by decomposing a BLACS send/receive invocation into a number of sends/receives (as was done for the computational kernels), this option was abandoned *i)* to preserve the communication pattern of the initial distributed algorithm in ScaLAPACK; *ii)* and, for programmability and simplicity, to avoid recoding (taskifying) the communication primitives in BLACS.

The communication-preserving taskification scheme is illustrated in Figure 5. A receive call is annotated as a real receive task plus several "artificial" tasks, one per block of the reception buffer. Both classes of tasks are mapped to the communication thread (see Section 3). The receive task actually receives the message, while the artificial tasks do nothing, they only receive the address of the corresponding block as an argument. (The overhead due to

the introduction of these artificial tasks is negligible.) By specifying this parameter as an output of the corresponding task, the correct data dependency is created between computation and communication tasks. When the actual data-flow execution takes place, the communication thread first issues the receive task. Once the execution of this task is completed (i.e., the point-to-point communication is done), the data dependencies of the artificial tasks are satisfied, so that they are immediately executed by the communication thread. A similar strategy was developed for the send calls, but in this case the directionality of the data dependency was reversed using the input clause for the artificial tasks.

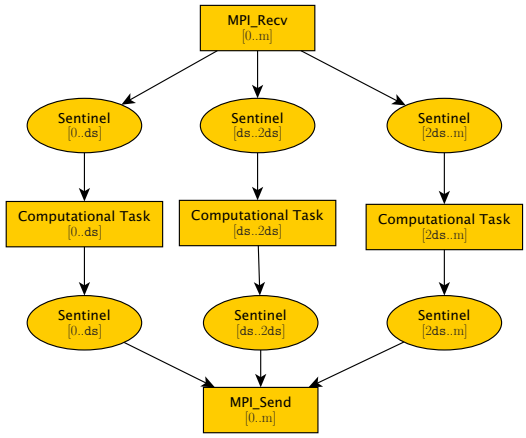


Figure 5: Example of taskification of communication kernels. Each node represents an SMPSs task. Assume that communication kernels operate with a buffer of m elements. Computational tasks operate with $m/\texttt{ds}$ elements ($m/\texttt{ds} = 3$ in current example). To track of the dependencies, "artificial" tasks are created for each portion of the buffer, thus maintaining only one communication task.

Recall from subsection 4.1 that the blocks of `B` to be exchanged in a point-to-point communication are packed by the source process into temporary buffers, and unpacked/transposed into an auxiliary workspace in the destination. The subroutines responsible for this data pack/unpack were also taskified. In contrast to tasks performing the actual data exchange, these tasks are not mapped to the communication thread. They are however marked with the `highpriority` clause with the purpose of accelerating the critical path of the algorithm.

5. Experimental Results

The experiments in this section were obtained using IEEE double-precision (DP) arithmetic on JuRoPa, a large-scale computing infrastructure from the Jülich Supercomputing Centre (JSC). JuRoPa comprises 2,208 compute nodes arranged in a QDR Infiniband interconnected cluster architecture. Each node is equipped with two quad-core INTEL Xeon E5570 processors running at 2.93 GHz, and 24 GBytes of DDR3 memory. The codes were compiled using INTEL Fortran and C/C++ compilers (version 11.1) with recommended optimization flags and we used ParaStation MPI (version 5.0.2-1) tools and libraries for native message-passing (based on MPICH2). The codes were linked against the BLAS and LAPACK available on the INTEL MKL library (version 10.2, build 12). We employed the MPI/SMPSs compiler wrappers and runtime provided by the StarSs source code distribution (version 3.6).

Five competing implementations are evaluated in the experiments:

- `Reference`. This corresponds to a pure MPI implementation that employs the original `pdpotrf` routine from ScaLAPACK (in MKL) and a $\mathtt{p} \times \mathtt{q}$ grid of processes with one MPI process per core. In the experiments with 128 nodes/1,024 cores, the grid was set to $\mathtt{p} \times \mathtt{q} = 16 \times 64$; for 256 nodes/2,048 cores, $\mathtt{p} \times \mathtt{q} = 32 \times 64$; for 512 nodes/4,096 cores, $\mathtt{p} \times \mathtt{q} = 32 \times 128$; and for 1,024 nodes/8,192 cores, $\mathtt{p} \times \mathtt{q} = 64 \times 128$.

- `MT-BLAS`. This is a hybrid MPI/MT-BLAS parallel approach, with one MPI process per node and parallelism extracted at the node level from ScaLAPACK (in MKL) with an implementation of MT-BLAS. The grids for the experiments with 128 nodes/1,024 cores, was set to $\mathtt{p} \times \mathtt{q} = 8 \times 16$; for 256 nodes/2,048 cores, $\mathtt{p} \times \mathtt{q} = 8 \times 32$; for 512 nodes/4,096 cores, $\mathtt{p} \times \mathtt{q} = 16 \times 32$; and for 1,024 nodes/8,192 cores, $\mathtt{p} \times \mathtt{q} = 16 \times 64$.

- `SMPSs`. This is the version that employs SMPSs to extract parallelism at the node level from the ScaLAPACK and BLACS source codes[2] (version 1.8.0 and 1.1, resp.) configured with one MPI process per node, and the same grids as in the `MT-BLAS` implementation. An SMPSs thread was launched per core, plus an additional thread per node to handle MPI communications.

[2]Available online at `http://www.netlib.org/`.

- `DSBPCholesky`. This corresponds to the pure MPI implementation of the Cholesky factorization presented in [8], with a `p`×`q` grid of processes and one MPI process per core. Its main features are: (1) tiled algorithm; (2) distributed square block packet format for half memory requirements (w.r.t. full storage) and less data movement in both BLAS and communication subroutines; (3) look-ahead depth one combined with an static schedule of the operations which allows for an effective overlapping of communication and computation via non-blocking MPI subroutines.[3] In the experiments with 128 nodes/1,024 cores and 512 nodes/4,096 cores, the grid was set to `p`×`q` = 16×16 and `p`×`q` = 64×64, respectively, in agreement with the set-up in [8] that uses square grids of processors whenever possible. For the experiments with the rest of nodes/cores, the same grids as in the `Reference` implementation were used. We used DSBPCholesky version 2008-10-28[4].

- `Elemental`. This corresponds to the pure MPI implementation of the Cholesky factorization available in Elemental [9], with a `p` × `q` grid of processes and one MPI process per core. Elemental is a framework for distributed-memory dense linear algebra that is designed to be a modern extension of the communication insights of PLAPACK [3] for element-wise distribution of matrices (versus the block distribution schemes of the rest of codes considered in this study). In the experiments, we let Elemental internally decide the grid shape for each number of cores explored. We used version 0.81 of this package[5].

A best effort was done to identify, *where applicable*, the optimal grid configuration for each implementation –leading to the values indicated above– as well as the distribution and algorithmic block sizes. For example, for `DSBPCholesky`, only the grid configuration and distribution block size had to be tuned (as the latter is equivalent to the algorithmic block size); while

[3]We note that in [8] the authors also present a variant of the code that dynamically schedules tasks within each MPI process with the purpose of alleviating the effect of the so-called spurious synchronizations. This variant is not considered in this work mainly because only small improvements are reported in [8] with respect to the static scheduling version. Furthermore, the DSBPCholesky codes publicly available do not include the dynamic variant.

[4]Available online at `http://www8.cs.umu.se/~larsk/`.

[5]Available online at `http://libelemental.org/`.

for `Elemental` the optimization comprised only the algorithmic block size, as this package uses distribution block size equal to one (i.e., element-wise data distribution), and we let it internally select the grid configuration.

We initially performed a series of experiments to test the selection of an optimal parameter configuration for the different codes. On one hand, the two microbenchmarks in [8] were successfully tested, implying that the combination of message-passing software and underlying hardware in JuRoPa is practically able to achieve an effective overlapping of communication and computation. This is required by `DSBPCholesky` for high performance. On the other hand, we performed a comparison of the different implementations at hand on 256 cores for scaled matrix size ranging from n=2,000 to 90,000. For the latter problem size, all implementations were very close to (and some of them even reached) its asymptotic flops/sec. regime (i.e., computation time largely dominated by `GEMM` operations). For "small-size" problems ($n \leq 14,000$), `SMPSs`, `DSBPCholesky` and `Elemental` achieve an average performance improvement of 105, 147, and 117%, respectively, over `Reference`. For "medium-size" problems ($14,000 < n \leq 60,000$), the performance improvements were 50, 37, and 19%; while for the largest test cases, the improvements were 15, 15, and 7%. These results are in agreement (if not superior) to those reported in [8, 9], confirming an appropriate set-up of the codes subject of study.

Figure 6 reports the performance attained by the five parallel distributed-memory implementations on JuRoPa. In the left-hand side plot, we evaluate the weak scalability of the solutions, testing their performance for a varied number of cores while maintaining the memory usage per node of the ScaLAPACK-based codes constant to approximately 2 GBytes (i.e., 256 MBytes per core). This is the maximum memory a process can address since the implementations of ScaLAPACK/PBLAS/BLACS only use 32-bit integers. Therefore, a problem of that size results in the most computationally-dominated scenario that can be evaluated under this constraint. Note that less computationally-dominated experiments (i.e., smaller loads per core) can only increase the benefit of `SMPSs` over pure ScaLAPACK codes. Problem dimensions are n=178,884; 252,980; 357,768; and n=505,964 for 1,024, 2,048, 4,096, and 8,192 cores, respectively. In this plot, we report performance in terms of TFLOPS (1 TFLOPS = 10^{12} flops/sec.); also, the y-axis for this plot ranges from 0 to the peak (theoretical) performance that can be attained using 8,192 cores of JuRoPa, which corresponds to 2.93 GHz $\times$ 4 DP flops/cycle $\times$ 8,192 cores $\approx$ 96 TFLOPS. The curves labeled as "`GEMM`"

provide a practical achievable peak, defined as the sequential matrix-matrix product kernel from INTEL MKL scaled by the number of cores. The results demonstrate that the `SMPSs` implementation clearly outperforms both the `Reference` and MT-BLAS versions of the codes, as well as `Elemental`. Compared to `DSBPCholesky`, our solution achieves mildly higher performance for 4,096 or less cores, but much higher for 8,192 cores. For instance, for the largest number of cores, `SMPSs` achieves 70.76 TFLOPS, which represents about 76% of the peak performance; while `Reference` and `MT-BLAS` attain around 49–51 TFLOPS, i.e., 51–53% of the peak performance; and `Elemental` and `DSBPCholesky`, around 55–57 TFLOPS, respectively.

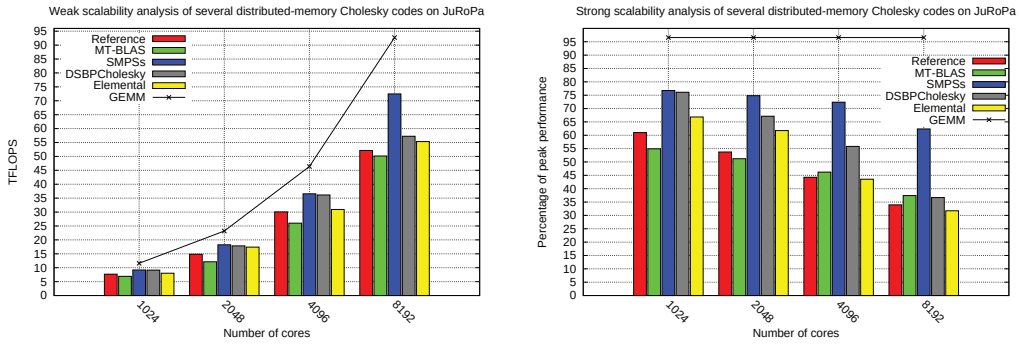


Figure 6: Performance of several distributed-memory Cholesky codes on JuRoPa.

The right-hand side plot of Figure 6 analyzes the strong scalability of the five competing codes. For this purpose, we evaluate their efficiency (in terms of attained percentage of peak TFLOPS rate) for a problem of fixed dimension n=178,884 and a range of cores from 1,024 to 8,192. The results show a rapid decrease in efficiency for the `Reference`, `MT-BLAS`, `DSBPCholesky` and `Elemental` implementations as the number of cores grows, much faster than that experienced under the same conditions by `SMPSs`, thus revealing the stronger scalability of our solution.

6. Sources of performance improvements

The purpose of this section is to demonstrate, with the help of `Paraver` [10] performance analyzer, that the exploitation of task-parallelism enabled by the dependency-aware out-of-order execution of tasks intrinsic to SMPSs is the source of the superior performance and scalability observed in Section 5.

In order to achieve this goal, the `Reference` and `SMPSs` implementations are comprehensively compared for a simple test case where a 2×8 process grid is employed to factorize a matrix of fixed dimension n=10,560 on a small Infiniband-interconnected cluster composed of 16 nodes with two Intel Xeon HexaCore processors E5645, running at 2.40 GHz, and 24 GBytes per node. The `Reference` implementation is mapped to the target computer with only one MPI process per node. The same applies to the `SMPSs` implementation, that spawns only two threads per MPI process/node (one worker thread and an additional thread devoted to communication tasks). A best effort was done to identify the optimal distribution and algorithmic block sizes. While the environment and set-up of the codes is different from the one in our raw scalability study (see Section 5), it is representative enough and greatly simplifies the presentation, as exactly the same computations are performed by both codes (although scheduled in a fairly different, more intelligent way, in the `SMPSs` implementation).

Given the block cyclic nature of the message-passing Cholesky factorization, for simplicity we found convenient to focus on a given cycle of the algorithm, in particular that comprising blocked iterations 3–10[6]. Figure 7 depicts several computations which are performed during this cycle, the corresponding MPI processes these computations are mapped to, and the iteration in which they take place. This set includes crucial computations and related events such as the factorization of a leading diagonal block (i.e., the start of a new iteration), the local transposition of blocks (i.e., the last computation the algorithm performs right before communication ends at a given iteration[7]), as well as symmetric rank-k updates and matrix-matrix multiplications required for the computation of the next panel. All computations in Figure 7 are marked as `highpriority` in the `SMPSs` implementation with the purpose of accelerating the critical path of the algorithm.

[6]Note that we refer to "blocked" iterations. In the main iteration loop for the Cholesky factorization the loop counter `k` takes on the values 1, $\texttt{ds}+1$, $2 \cdot \texttt{ds}+1, \ldots$ (see Listing 1), and by blocked iteration i we refer to that for which $\texttt{k}=(i-1) \cdot \texttt{ds}+1$. A cycle comprises a minimal set of consecutive iterations $i, i+1, \ldots, i+l-1$ such that no process performs more than one factorization of a leading diagonal block, while the same process is responsible for the factorization of a diagonal block in iterations i and $i+l$, where l, the length of the cycle, is given by the least common multiple (LCM) between the number of rows and columns in the process grid. For the particular case of a 2×8 process grid, $\text{LCM}(2, 8) = 8$.

[7]Recall that communication ends at a given iteration when the processes in a column broadcast rowwise the transposed panel; see Figure 2 (e).

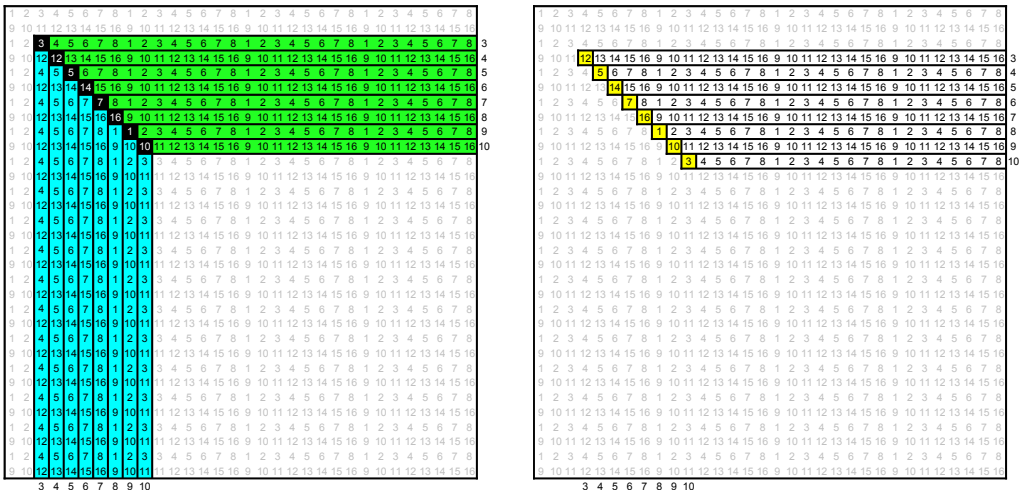


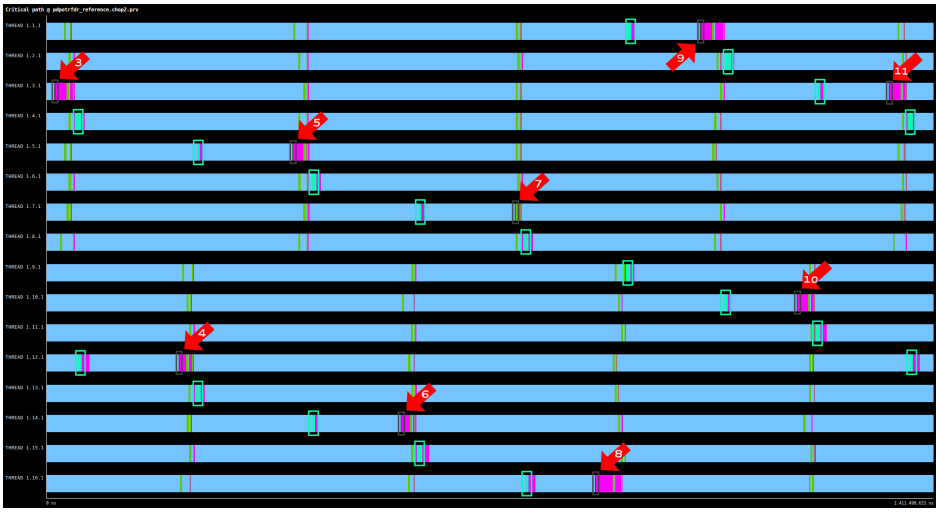
Figure 7: Several computations performed during a cycle comprising iterations 3-10 of the message-passing Cholesky factorization applied to a matrix cyclically distributed by blocks over a 2×8 process grid. The number within each block identifies the process identifier the corresponding computation is mapped to. The numbers at the bottom and right-hand side of the two boxes determine the iteration where the corresponding computation takes place. Correspondence between kernels and colors is as follows: Cholesky factorization of leading diagonal blocks in black, solution of triangular systems with several right hand sides in green, local transposition of blocks in light blue, and symmetric rank-`ds` updates and matrix-matrix multiplications required for the computation of the next panel in yellow and white, respectively. These computations are marked as `highpriority` in the `SMPSs` implementation.

Tracing information was extracted from the `Reference` and `SMPSs` implementations using an ad-hoc tracing scheme based on `Extrae` and `Paraver` [10] tools. `Extrae` is an instrumentation and trace generation software package, while `Paraver` is a parallel performance analyzer and visualization tool. The tracing scheme was implemented such that computation and communication subroutines are replaced by wrapper subroutines that trigger `Extrae` events on entry and exit. In consequence, each time the codes enter and exit a computation or communication subroutine, the trace records the corresponding events with a timestamp.

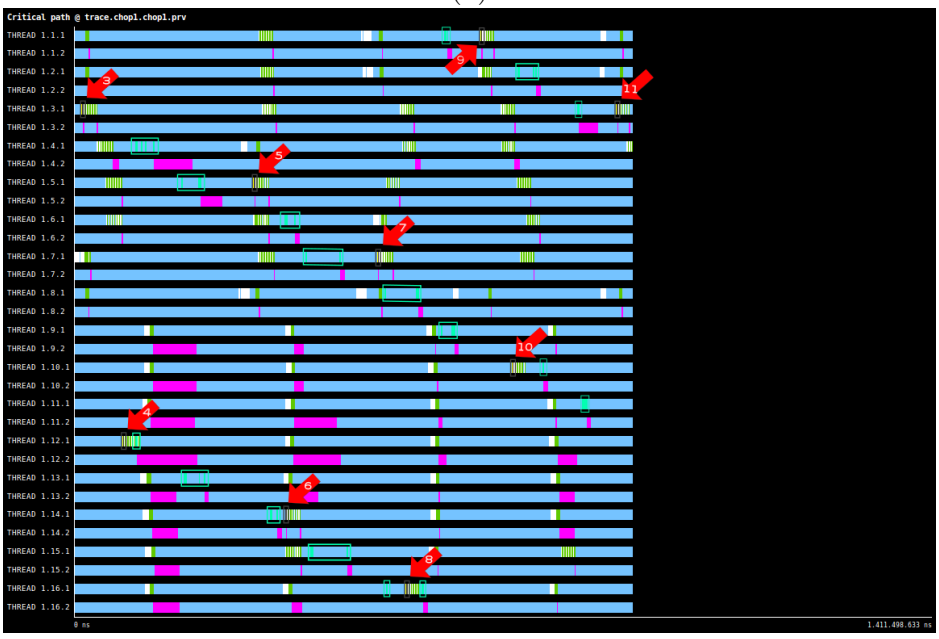
Figures 8 (a) and (b) show captures of the `Paraver` time-line view corresponding to traces (restricted to a cycle comprising iterations 3-10) of the `Reference` and `SMPSs` implementations, respectively. `Paraver` time-line view

shows the computations or communications performed by each thread of the parallel program along a horizontal time-line axis. The label "`THREAD 1.X.Y`" to the left of each horizontal bar identifies a particular thread involved in the parallel computation, where `X` and `Y` are the process and thread (within process) identifiers, respectively. In the `Reference` implementation, only one thread is spawned per process, so that `Y` always equals 1. In the `SMPSs` implementation, two threads are spawned per process, with `Y`=1 and `Y`=2 being the computation and communication threads, respectively. For both implementations, `X` ranges from 1 to 16, i.e., a 2×8 process grid, with the former eight identifiers assigned to processes in the first row of the grid from left to to right, while the latter eight to those in the second row. The filter options of the `Paraver` time-line view were used to filter and color only those events strictly related to computations depicted in Figure 7. Apart from these computations, the send side of a broadcast operation (i.e., BLACS's `DGEBS2D` subroutine) is also colored in magenta in Figure 8. (Note that in Figure 8 (b) broadcast operations are always performed by the communication thread.) Any time-interval that does not correspond to filtered events is colored in blue. In order to facilitate the comparison between the `Reference` and `SMPSs` implementations, the time-interval of Figure 8 (b) has been scaled so that it matches that of Figure 8 (a). The reader can easily note that the execution of iterations 3-10 is encompassed in a much smaller time interval in the case of the `SMPSs` implementation.

Figure 8 (a) clearly exposes the synchronous, in-order execution of the `Reference` implementation. Once all computations from the previous iteration have been completely executed, a new iteration starts with the factorization of a leading diagonal block, and then proceeds with the computation of a new panel (green colored time intervals in Figure 8 (a)). In preparation for the distributed symmetric rank-k update, the panel and its transpose are then broadcast across the process grid. When communication is complete (magenta colored time intervals immediately following light blue colored boxes in Figure 8 (a)), the next iteration only starts when the whole distributed symmetric rank-k update of the current iteration is complete. This can be readily observed in Figure 8 (a), by taking a look at the length of the time intervals encompassing the start of two consecutive iterations (i.e., the "distance" between two consecutive red arrows in the figure). For example, measured values with the help of `Paraver` are 157.3, 147.2, and 148.6 milliseconds for the time intervals encompassing the start of 7-8, 8-9 and 9-10, respectively. A quite uniform distribution can be observed in general looking at small



(a)



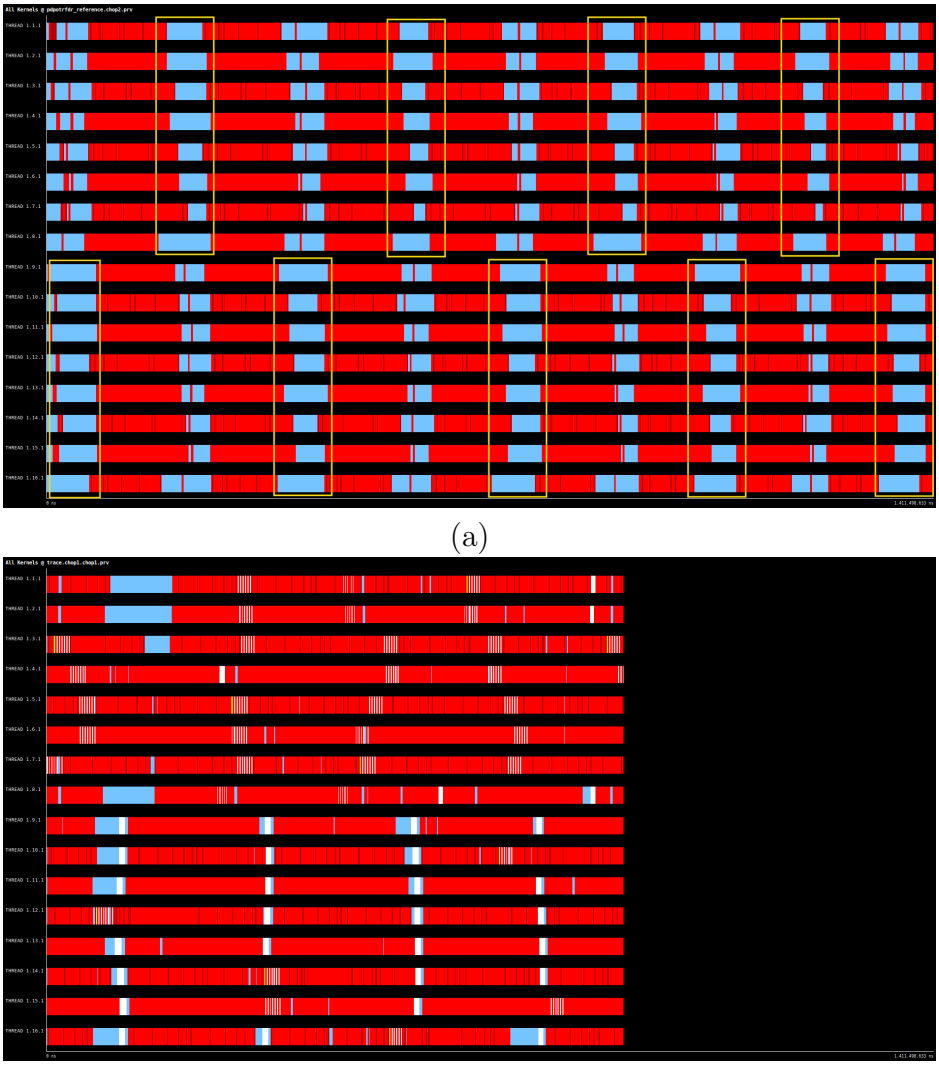
(b)

Figure 8: Capture of the `Paraver` time-line view of the (a) `Reference` and (b) `SMPSs` implementation on a 2×8 process grid. Focus is on a cycle comprising iterations 3-10 (see Figure 7). Red arrows are used to mark the start of a new iteration, with the white numbers within them representing the iteration identifier. The same correspondence among computations and colors as in Figure 7 is used. The send side of broadcast operations (i.e., BLACS's `DGEBS2D` subroutine) has been also colored in magenta, so that those (broadcasts) immediately following light blue colored boxes can be used to determine when does communication at each iteration is complete (see Figure 2 (e)).

clusters of consecutive iterations.

An immediate parallel performance penalty in the form of idle threads derived from the synchronous, in-order execution of the `Reference` implementation is depicted in Figure 9 (a). The filter options of the `Paraver` time-line view were used to filter and color in red only those events associated with the symmetric rank-k updates and the matrix-matrix multiplications. There it can observed that processes not involved in the computation of a new panel waste their time waiting on communication operations (see Figure 2). These periods of time (in blue) have been enclosed in yellow boxes in Figure 9 (a). This is indeed revealed as the most contributing parallel overhead in the `Reference` implementation.

Figure 8 (b) reveals a fairly different scenario to that observed for the `Reference` implementation in Figure 8 (a). In particular, the length of the time intervals encompassing the start of two consecutive iterations is now significantly reduced. Measured values for the time intervals encompassing the start of iterations 3–4, 4–5, 5–6, 6–7, 7–8, 8–9, 9–10, and 10–11 are 67.2, 216.8, 53.5, 151.9, 48.3, 125.2, 52.6, and 169.9 milliseconds, respectively, in contrast to 200.3, 181.7, 175.1, 181.1, 168.1, 157.3, 147.2, and 148.6 milliseconds for the `Reference` implementation. This is essentially consequence of a combined effect of the out-of-order execution intrinsic to `SMPSs` and the overlapping of computation and communication achieved with the aid of an additional thread devoted to communication operations. When a given iteration starts, there are still pending SMPSs tasks (in particular, regular symmetric rank-k updates and matrix-matrix multiplications) from previous iterations, so that those threads that are not involved in the computation of a new panel can overlap the execution of these pending SMPSs tasks with the communication required to prepare the next distributed symmetric rank-k update. One can indeed observe in Figure 8 (b) that the execution of a new iteration does not start right after communication of the previous iteration is complete, i.e., there is a non-negligible time between magenta colored time intervals immediately following light blue colored boxes and the start of a new iteration. This is the time that the `SMPSs` implementation takes to schedule and execute those pending tasks from previous iterations that are required to fulfill the dependencies of those tasks colored in yellow and white in Figure 7. The latter tasks are executed immediately as long as their dependencies are fulfilled since they are marked as `highpriority`. However, the order on which the former tasks are executed cannot be controlled and depends on how the SMPSs runtime orchestrates the execution of multiple



(a)

(b)

Figure 9: Capture of the `Paraver` time-line view of the (a) `Reference` and (b) `SMPSs` implementation on a 2×8 process grid. Focus is on a cycle comprising iterations 3-10. Only symmetric rank-k updates and matrix-matrix multiplications are considered, which are both highlighted in red, but those marked as `highpriority` in the `SMPSs` implementation are highlighted in yellow and white, respectively. In the `Reference` implementation, processors not involved in the computation of a new panel idle on communication operations (see Figure 2). These periods of time (in blue) have been enclosed in yellow boxes in (a). In the `SMPSs` implementation, these idle periods are significantly reduced as they are filled with useful work (i.e., regular symmetric rank-k updates and matrix-matrix multiplications) from previous iterations.

regular tasks with their dependencies fulfilled. It may indeed happen that regular symmetric rank-k updates and matrix-matrix multiplications from previous iterations that are not in the dependency path that ends on the former tasks are executed before those that are in the dependency path. This partly explains why the time between the start of two consecutive iterations varies significantly from iteration to iteration. Although it would be desirable to minimize the time between two consecutive iterations, the `SMPSs` implementation still achieves excellent performance as idle periods that were present in Figure 9 (a) have mostly disappeared from in Figure 9 (b). This is clearly a consequence of the exploitation of task-parallelism enabled by the dependency-aware out-of-order execution of tasks intrinsic to SMPSs.

7. Concluding Remarks

In this paper, we demonstrate how the SMPSs runtime scheduler can efficiently exploit intra-node hardware concurrency, improving the performance and scalability of the ScaLAPACK routine for the Cholesky factorization on a cluster of multicore processors. This superior efficiency is identified as being rooted in the out-of-order execution enabled by the detection of task dependencies and the adaptive scheduling performed by SMPSs, which prioritizes the execution of tasks in the critical path, similarly to what would be obtained with a dynamic look-ahead, but without incurring in the programming complexity of this technique.

Our analysis exposes that the major difficulties in porting the reference implementation of the distributed-memory routines from ScaLAPACK are due to certain restrictions of SMPSs, but also that these can be easily addressed with a limited programming effort. The introduction of the SMPSs programming model is thus revealed as a powerful tool to improve the performance of numerical kernels in legacy, message-passing libraries for dense linear algebra on clusters equipped with multicore technology. This solution thus exhibits software reusability as a clear advantage over other existing approaches.

Acknowledgements

This research was supported by Project EU INFRA-2010-1.2.2 "TEXT: Towards EXaflop applicaTions". The researcher at BSC-CNS was supported by the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the Spanish

Ministry of Education (CICYT TIN2011-23283, TIN2007-60625 and CSD2007-00050), and the Generalitat de Catalunya (2009-SGR-980). The researcher at CIMNE was partially funded by the UPC postdoctoral grants under the programme "BKC5-Atracció i Fidelització de talent al BKC". The researcher at UJI was supported by project CICYT TIN2008-06570-C04-01 and FEDER.

We thank Jesus Labarta, from BSC-CNS, for helpful discussions on SMPSs and his help with the performance analysis of the codes with `Paraver`. We thank Vladimir Marjanovic, also from BSC-CNS, for his help in the set-up and tuning of the MPI/SMPSs tools on JuRoPa. Finally, we thank Rafael Mayo, from UJI, for his support in the preliminary stages of this work.

The authors gratefully acknowledge the computing time granted on the supercomputer JuRoPa at Jülich Supercomputing Centre.

References

[1] G. H. Golub, C. F. Van Loan, Matrix Computations, 3rd Edition, Johns Hopkins University Press, Baltimore, MD, 1996.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK Users' Guide, SIAM, 1997.

[3] R. van de Geijn, Using PLAPACK: Parallel Linear Algebra Package, The MIT Press, 1997.

[4] P. Strazdins, A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization, Tech. Rep. TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia (1998).

[5] J. J. Dongarra, P. Luszczek, A. Petitet, The LINPACK benchmark: past, present and future, Concurrency and Computation: Practice and Experience 15 (9) (2003) 803–820.

[6] J. Perez, R. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: 2008 IEEE International Conference on Cluster Computing, 2008, pp. 142–151.

[7] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, G. Quintana-Ortí, Parallelizing dense and banded linear algebra

libraries using SMPSs, Concurrency and Computation: Practice and Experience 21 (2009) 2438–2456.

[8] F. Gustavson, L. Karlsson, B. Kågström, Distributed SBP Cholesky factorization algorithms with near-optimal scheduling, ACM Transactions on Mathematical Software 36 (2) (2009) 11:1–11:25.

[9] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, N. A. Romero, Elemental: A new framework for distributed memory dense matrix computations, ACM Transactions on Mathematical Software 39 (2) (2013) 13:1–13:24.

[10] `Paraver`: The flexible analysis tool, `http://www.bsc.es/paraver/`.

[11] Cilk project home page, `http://supertech.csail.mit.edu/cilk/`.

[12] J. M. Pérez, P. Bellens, R. M. Badía, J. Labarta, CellSs: Programming the Cell/B.E. made easier (2007).

[13] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. M. Pérez, J. Planas, E. S. Quintana-Ortí, Extending OpenMP to survive the heterogeneous multi-core era, International Journal of Parallel Programming 38 (5-6) (2010) 440–459.

[14] C. Augonnet, S. Thibault, R. Namyst, P. Wacrenier, StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput. : Pract. Exper. 23 (2) (2011) 187–198.

[15] G. F. Diamos, S. Yalamanchili, Harmony: An execution model and runtime for heterogeneous many core systems, in: Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC '08, ACM, New York, NY, USA, 2008, pp. 197–200.

[16] T. Gautier, J. V. Lima, N. Maillard, B. Raffin, XKaapi: A runtime system for data-flow task programming on heterogeneous architectures, in: Parallel and Distributed Processing Symposium, International, IEEE Computer Society, Los Alamitos, CA, USA, 2013, pp. 1299–1308.

[17] G. Quintana-Ortí, E. S. Quintana-Ortí, R. van de Geijn, F. V. Zee, E. Chan, Programming matrix algorithms-by-blocks for thread-level

parallelism, ACM Transactions on Mathematical Software 36 (3) (2009) 14:1–14:26.

[18] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Computing 35 (1) (2009) 38–53.

[19] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid gpu accelerated manycore systems, Parallel Comput. 36 (5-6) (2010) 232–240.

[20] J. I. Aliaga, M. Bollhöfer, A. F. Martín, E. S. Quintana-Ortí, Exploiting thread-level parallelism in the iterative solution of sparse linear systems, Parallel Comput. 37 (3) (2011) 183–202.

[21] K. Kim, V. Eijkhout, Scheduling a Parallel Sparse Direct Solver to Multiple GPUs, in: The 14th IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, 2013.

[22] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide, SIAM, Philadelphia, 1992.

[23] F. G. V. Zee, `libflame`: The Complete Reference, `www.lulu.com`, 2009.

[24] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, DAGuE: A Generic Distributed DAG Engine for High Performance Computing, Parallel Comput. 38 (1-2) (2012) 37–51.

[25] F. D. Igual, G. Quintana-Ortí, R. van de Geijn, Scheduling algorithms-by-blocks on small clusters, Concurrency and Computation: Practice and Experience 25 (3) (2013) 367–384.

[26] V. Marjanović, J. Labarta, E. Ayguadé, M. Valero, Overlapping communication and computation by using a hybrid MPI/SMPSs approach, in: Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, ACM, New York, NY, USA, 2010, pp. 5–16.

[27] J. Perez, R. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: Cluster Computing, 2008 IEEE International Conference on, 2008, pp. 142 –151.

[28] J. M. Perez, R. M. Badia, J. Labarta, Handling task dependencies under strided and aliased references, in: Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, ACM, New York, NY, USA, 2010, pp. 263–274.

[29] A. Petitet, J. Dongarra, Algorithmic redistribution methods for block-cyclic decompositions, IEEE Trans. Parallel and Distributed Systems 10 (12) (1999) 1201 –1216.