The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, Month 2011, 1–42

RESEARCH ARTICLE

Performance Analysis and Structured Parallelization of the STAP Computational Kernel on Multi-core Architectures

Daniele Buono, Gabriele Mencagli*, Alessio Pascucci and Marco Vanneschi
Department of Computer Science, University of Pisa
Largo B. Pontecorvo, 3, I-56127 Pisa, Italy
(*Received 00 Month 200x; in final form 00 Month 200x*)

The development of radar systems on general-purpose off-the-shelf parallel hardware represents an effective means of providing efficient implementations with reasonable realization costs. However, the fulfillment of the required real-time constraints poses serious problems of performance and efficiency: parallel architectures need to be exploited at best, providing scalable parallelizations able to reach the desired throughput and latency levels. In this paper we discuss the implementation issues of the computational kernel of a well-known radar filtering technique - the Space-time Adaptive Processing (STAP) - on today's general-purpose parallel architectures (multi-/many-core platforms). In order to address the performance constraints imposed by the real-time implementation of this filtering technique, we apply a structured approach (Structured Parallel Programming) to develop parallel computations as instances and compositions of well-known parallelization patterns. This paper provides a thorough description of the implementation issues and discusses the performance peaks achievable on a broad range of existing multi-core architectures.

Keywords: Space-time Adaptive Processing, Tile Algorithms, Multi-core Architectures, Structured Parallel Programming, Performance Evaluation.

1. Introduction

Space-time Adaptive Processing (STAP) [1–4] was originally proposed to suppress clutter and jamming interferences received by radar sensors on board moving platforms, e.g. aircrafts. The central point is that the radar optimum performance is achieved if a vector of adaptive weights is available for each range cell; in the real world, this stringent requirement can be relaxed considering the stationarity of the clutter plus directional interference processes which allows the weight vectors to be maintained constant for a set of nearby range cells (*training data*).

STAP is especially useful for applications of next-generation radar systems (e.g. for Homeland Security and I-Transportation), oriented towards the fast and accurate detection of very slow targets like men, small boats and vehicles. In this case STAP gives a unique and not questionable advantage, being capable of achieving strong clutter cancellation while allowing the detection of slow targets whose

*Corresponding author. Email: mencagli@di.unipi.it

ISSN: 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/17445760.YYYY.CATSid
http://www.informaworld.com

Doppler falls near to the mainbeam clutter Doppler.

However, the performance benefit of this technique needs to be traded-off with its computational cost. In fact, the STAP processor goal is to solve a large set of linear systems for calculating the adaptive weights. For this reason high-performance implementations, able to efficiently exploit special-purpose or general-purpose parallel computing architectures, are of great importance for meeting the real-time constraints imposed by the real-life radar applications.

In this context, existing research works [5–10] do not provide a comprehensive analysis of the STAP parallelization issues. Some of them present throughput-oriented parallelizations that consist in a straightforward replication of the entire computational chain onto special-purpose processing elements (e.g. FPGAs). Although FPGA is a successful technology, there are many circumstances where it is not efficient, mainly for cost, flexibility and expandability reasons. On the other hand, developing efficient STAP implementations on general-purpose hardware (e.g. multicores) is not free of difficulties, especially for defining highly-scalable versions able to achieve acceptable results on a broad range of parallel platforms like the ones available today (i.e. the so-called *performance portability*).

In this paper we study high-performance implementations of the computational kernel of the STAP algorithm, i.e. *the calculation of the weight vectors*. We design and develop parallel computations by instantiating well-known parallelism schemes (e.g. task-farm and data-parallel patterns) whose effects on throughput and computation latency can be clearly analyzed. This approach, known as *Structured Parallel Programming* (SPP), besides offering a large-degree of programmability and compositionality, is an interesting technique for performance portability. Furthermore, structured parallelism schemes can be nested and composed in complex and potentially hierarchical structures.

This work describes the application of structured parallelizations to accelerate the STAP computational kernel. We propose several parallelizations based on different parallelism patterns. We evaluate our implementations on three multi-core architectures: a flagship Intel platform based on the Xeon family, an AMD Opteron multiprocessor and an IBM architecture composed of four Power7. The goal is to provide a wide overview of the effectiveness of our implementations on architectures that cover the entire field of the most important commercially available general-purpose machines.

The paper provides the following contributions:

- this work is targeted to practitioners in the field of radar systems. The paper shows experimental results which are an interesting description of the peak performance reachable by STAP implementations on today's multi-core architectures. In particular, for the broad set of architectures used, we claim that this work is of relevance to understand the potential of general-purpose architectures, and provides the basis for a comparison with future implementations on emerging hardware technologies;
- the paper has a meaningful content targeting experts in the parallel programming field. Structured parallelism patterns and their combination are extensively used in this work, by identifying their effect on the throughput and latency components of the overall performance. This represents a notable example of applying

SPP design principles on a significant real-life application;

- we present an integration between structured parallelism patterns and existing linear algebra libraries. Although such libraries already provide parallel implementations of the most costly routines (e.g. for matrix decomposition), to solve the STAP scenario we need to apply parallelizations at different levels of the problem. In this paper we discuss the use of task-farm and data-parallel patterns in order to minimize the completion time. Furthermore, for the typical problem size of STAP, we show that our data-parallel implementation achieves a better performance compared to parallel matrix decompositions recognized in the literature as *de-facto* standard for their performance on multicores.

In the past, the authors of this paper have already studied the problem of providing efficient implementations of linear algebra kernels on multicores using structured parallelism patterns [11–13]. This paper represents an extension of our previous work, and it is focused on the STAP use-case. The semantics of this algorithm and the presence of an input stream of computational tasks, notably the calculation of a set of matrix decompositions, fosters the optimization process by relying on parallelism patterns and their composition, representing a concrete application of the SPP design methodology.

The organization of this paper is the following. In the next section we give a brief overview about the STAP technique. In Section 3 we describe the main features of the existing approaches to the STAP parallelization and SPP. In Section 4 we discuss the importance of a proper selection of the most suitable sequential algorithms for designing high-performance solutions. In Section 6 and 7 we describe our parallelization approach, the basic properties that a portable (in the performance sense) implementation should have, and the results achieved on the three test-bed multicore architectures. Section 8 gives the conclusion of this paper.

2. Space-time Adaptive Processing

STAP is a well-known technique for suppressing clutter and jamming effects in airborne environments. The procedure operates in the spatio-temporal domain and assumes the presence of an array of D antennas, which transmit a coherent burst of P radar pulses every PRT seconds (Pulse Repetition Time). Echoed signals are received during a coherent time period of $P \cdot PRT$ seconds, namely *Coherent Processing Interval* (CPI). Data samples for each CPI are collected in N range samples or *range cells*, where N is the product of PRT and the sampling frequency. Data received during a CPI can be represented as a tri-dimensional cube, i.e. N different matrices of $M = D \times P$ complex numbers, that can be alternatively represented as N *space-time vectors* $\mathbf{x} \in \mathbb{C}^M$ of complex elements. A representation of a CPI data-cube is depicted in Fig. 1.

A radar is aimed at ascertaining whether targets are present in the input samples from echoed signals. Given a primary range cell (also called *cell under test*) where the target is expected to be located, the STAP algorithm establishes the presence of the target by calculating an optimal weight vector and applying it to the samples from the range cell of interest. To do that, an *interference covariance*

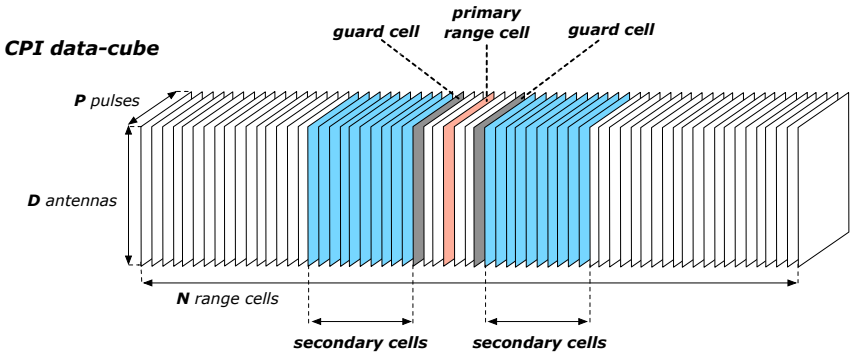


Figure 1.: CPI data-cube: primary range cell and secondary range cells used to estimate the interference covariance matrix.

matrix $\mathbf{Q} \in \mathbb{C}^{M \times M}$ must be estimated using directly the sample data. One method is to compute and average this for many range cells surrounding but excluding the primary range cell. The covariance matrix is defined as follows:

$$\hat{\mathbf{Q}} = E[\mathbf{x}_i \cdot \mathbf{x}_i^H] = \frac{1}{\sigma} \sum_{i=1}^{\sigma} \mathbf{x}_i \, \mathbf{x}_i^H \tag{1}$$

where the estimation involves σ training cells (called *secondary range cells*) close to the cell under test. To achieve a good estimate, specific rules must be followed. In particular, for the matrix being invertible we need at least M training cells. Moreover, in order to obtain a sufficiently small SINR, we need at least $\sigma \geq 2\,M$ training cells (according to the Brennan's rule [14]).

After the covariance matrix estimation, the weight vector $\mathbf{w} \in \mathbb{C}^M$ is calculated by solving the following system of linear equations:

$$\hat{\mathbf{Q}}\,\mathbf{w} = \mathbf{s} \tag{2}$$

where $\mathbf{s} \in \mathbb{C}^M$ is the steering vector identifying the target, formed by the vector product of the vector representing the Doppler frequency and the vector representing the antenna angle of elevation and azimuth. The system is solved using a decomposition of the covariance matrix (e.g. QR or Cholesky decompositions) and performing the substitutions to calculate the final weights.

The final step is to determine the presence of the target. This is performed by calculating the inner product between the weight vector and the data from the cell under test, i.e. $z = \mathbf{w}^H \cdot \mathbf{x}$. The result is a scalar z which will be compared against a specific threshold which depends on the used false alarm probability.

Although the same covariance matrix can be used with multiple targets at the same range, more generally Γ different covariance matrices (from few tens to hundreds depending on the radar configuration) must be estimated for each CPI data-cube, in order to cover the entire range of the radar.

2.1 STAP computational requirements

There are two fundamental issues arising from the application of STAP to real scenarios: the computational load to compute the weight vectors, and the number of training samples required for a correct estimation of the interference covariance matrices. To solve these issues, several sub-optimal techniques have been proposed over the last years (called partially-adaptive STAP [15]), that reduce the size of the problem. In terms of computational cost, the covariance matrices estimation and the calculation of the weight vectors are the most computationally demanding phases of the entire algorithm. In particular the calculation of each weight vector requires to solve a linear system of equations which needs $\mathcal{O}(M^3)$ floating-point operations (where M is the size of the coefficient matrix).

Radar devices are configured to meet precise real-time constraints. Notably *latency* and *throughput* represent a classical dichotomy that drives the system design. General requirements can be summarized as follows:

- CPI cubes arrive from the antenna elements with a frequency that depends on the radar characteristics. A first design requirement consists in a *throughput-oriented constraint*: the system should be able to produce the filtered results at the same frequency of the inputs;
- an optimal implementation from the throughput standpoint might be not effective to accurately track moving targets: e.g. albeit the results are produced at the same frequency of the arrivals, we further need that results completed at a time instant refer to data received not so far in the past. To this end a *latency-oriented constraint* is usually required.

Nowadays, to get close to such real-time requirements, it is possible to exploit the recent trend in general-purpose computer technology, in which the degree of parallelism became significant even inside single on-chip platforms. In this paper we consider the optimal STAP technique (also referred in the literature as fully-adaptive STAP) and the implementation of its computational kernel on general-purpose multicore architectures.

3. Related Work

STAP has been introduced in detail in [1, 2, 4], in which the rationale of this technique has been presented discussing real-world applications. As stated in these works, the STAP applicability is hampered by the computational burden required for estimating the covariance matrices and the weight vectors. During the 1990s, STAP parallelizations on multiprocessor architectures such as the Intel Paragon and the Cray T3E machines [5–8] were of special interest, especially because STAP was considered a benchmark for measuring the peak performance level of those architectures. Interesting parallelizations for massively parallel multicomputers were described in [16], providing the implementation details of ad-hoc optimizations on those special-purpose environments. We claim that, besides approaches based on FPGAs, GPUs and other special-purpose technologies, a study of portable STAP parallelizations on general-purpose architectures can be of great importance to un-

derstand whether such architectures may represent potential candidates to execute real-life radar applications or not.

Recently, STAP parallelizations have mainly faced *throughput-oriented* constraints, by replicating the computation on different parallel units exploited at best through efficient scheduling strategies [8, 9]. There are two critical aspects of such parallelizations: (i) they do not provide sufficient details of how the parallelization is performed inside each computation phase; (ii) it is not clear how these approaches deal with *latency-oriented* constraints that, to our knowledge, are much more difficult to meet and can significantly influence the parallelization design.

As stated in the literature [4], the most compute-intensive phase of STAP is the calculation of the weight vectors, which requires efficient implementations operating on streams of linear algebra kernels (matrix multiplication and linear system resolution). Two interrelated aspects deserve special attention when designing a highly scalable parallelization: (i) a design approach which helps in defining parallelizations by reasoning about their effect on throughput and latency metrics; (ii) routines and libraries to perform an efficient execution of linear algebra kernels. These two aspects will be the subject of the next two parts of this short review.

Structured Parallel Programming. The rationale of SPP [17–19] is the vision of parallel programs as instances of a limited set of parallelism paradigms with a well-defined semantics. Basic typical paradigms are task-farm, pipeline, data-parallel (available in several variants, notably map, reduce and stencils), divide-and-conquer and so on. SPP implies a high-level reasoning on parallel programming, by separating the computation part with the coordination scheme expressed in terms of interconnections between parallel entities performing specific functionalities, e.g. emitter, collector, workers, gather, scatter, multicast.

Over the last years, a lot of frameworks and programming environments have adopted the SPP approach. Some of them consist in coordination languages used by the programmer to instantiate parallelism paradigms (more properly called *skeletons*) through specific high-level constructs. Examples are SkiE [20], P3L [21] and SCL [22]. ASSIST [23] extends this vision by providing a more general concept of skeleton (called ParMod), able to be specialized to capture irregular or more complex patterns for which pre-defined skeletons have not been provided.

Especially for flexibility reasons, skeleton-based frameworks have been recently presented as libraries in established languages. Examples of skeleton libraries for Java are Calcium [24], Muskel [25], JaSkel [26] and Skandium [27]. They provide a rich set of skeletons (at least task-farm, map and reduce are common in all the libraries) and the possibility to define complex hierarchical structures in order to exploit the combination of different paradigms on the desired evaluation metrics (service time, latency and memory usage). Eden [28], Muesli [29], SKELib [30], FastFlow [31] and SkeTo [32] are examples of libraries written in C/C++.

By exploiting the decoupling between computation and coordination, SPP represents a software engineering method to develop parallel programs which entails important and desirable properties. Firstly, code portability can be highly improved using skeletons: by focusing on the coordination scheme, different implementations

of run-time support mechanisms for cooperation among threads/processes can be defined to port the same program on different architectures. Furthermore, high-level skeletons foster the development of parallel programs *portable in the performance sense*: by enabling the static/dynamic selection of the most suitable implementation variants of each skeleton.

In this paper we adopt the SPP approach in the definition of our STAP parallel solutions. Nevertheless, instead of using a specific framework, we prefer to be independent and general by using SPP as a design concept rather than exploiting a concrete framework. Therefore, we implement the parallelism paradigms necessary for our STAP parallelization by hand, using the separation of concerns between computation and coordination and the nesting of skeletons to achieve our goals.

Linear algebra libraries and tools. In the last twenty-five years a lot of experience has been gained by several research groups in the development of highly efficient libraries for solving linear algebra problems such as matrix decomposition, least-square algorithms and eigenvalues calculation. `LINPACK` [33] is a collection of Fortran sub-routines to solve basic problems using mainly level-1 Basic Linear Algebra Sub-Programs (`BLAS`) (vector-vector operations).

`LaPack` [34] improves the performance of `LINPACK` on shared-memory architectures. On these machines, `LINPACK` operations use inefficiently the memory hierarchy, spending a large fraction of the execution to move data between different levels of the hierarchy. The main goal of `LAPACK` is to run efficiently on shared-memory architectures featuring multi-layered memory hierarchies. To do that, `LAPACK` routines exploit cache-aware algorithms designed to reduce the amount of memory traffic by maximizing data reuse in higher level caches. The rationale is to maximize the usage of level-3 `BLAS` calls (matrix-matrix operations) that can be highly optimized in order to exploit to the full the memory hierarchies. `LAPACK` also offers parallel implementations of the most critical routines and functions. The general idea consists in a *fork-and-join* pattern, in which the execution flow alternates sequential phases involving small regions of the matrix and parallel ones operating on a large set of data. Nowadays, many `LAPACK` implementations optimized for different architectures exist, e.g. the `MKL` library [35] for Intel and `GotoBLAS` [36] for x86 and IBM Power architectures. ATLAS [37] is an open-source portable implementation of `BLAS` and some `LaPack` routines on shared-memory platforms. Its main feature is the possibility to automatically generate optimized code through an iterative compilation. Portability is at the detriment of performance, as `ATLAS` is slower than specialized implementations (`MKL` and `GotoBLAS` are usually two or more times faster).

On modern multi-/many-core architectures the fork-and-join parallelism pattern reveals unacceptable performance by limiting the maximum exploitable parallelism degree. To overcome these limitations, the `PLASMA` library [38] introduces a new set of algorithms (named *tile algorithms* [39–41]) in which parallelism is not bounded inside the `BLAS` kernel, but it is described at a higher level modeling the computation as a directed graph of tasks. Similar concepts have been applied to heterogeneous systems (CPU-GPU) using the `MAGMA` library [42].

On distributed-memory architectures, `ScaLAPACK` [43] is the reference library. The fundamental building blocks of `ScaLAPACK` are the distributed implementation of level-1 and level-2 `BLAS` and a set of Basic Linear Algebra Communication Subprograms (`BLACS`) for communicating tasks among distributed nodes.

4. STAP Computational Kernel

The estimation of covariance matrices and the calculation of weight vectors are the most compute-intensive parts of the STAP. Other phases, though potential and interesting candidates for a parallelization (e.g. Doppler Filtering, Beamforming and Pulse Compression), are less computationally demanding and can be efficiently implemented according to optimized sequential algorithms and libraries [5–10].

As we have seen in Section 2, the calculation of each covariance matrix involves a certain number of independent outer products and the sum of the partial results. The simple structure of this computation is amenable to being implemented directly at the hardware level, by exploiting dedicated FPGAs, SIMD co-processors and GPUs [44, 45]. On the other hand, the calculation of weight vectors is more interesting, leading to problems that are difficult to parallelize efficiently. For this reason we focus on this part of the STAP algorithm.

4.1 Weight vectors calculation

The calculation of a weight vector consists in solving a system of linear equations. A classic approach is based on performing a decomposition of the covariance matrix and then solving systems typically with sparse matrices. In principle the choice of a decomposition algorithm depends on the properties retained by the matrices. Most of the existing research work [5, 6, 9, 10] discusses parallel implementations of the QR decomposition, due to the generality of this method that requires no particular constraint on the input matrix. Nevertheless, the QR decomposition is not always the best candidate, and other techniques can be more efficient.

Along this line the *Cholesky decomposition* is a relevant example. It requires a smaller number of operations than the QR method, i.e. its time complexity is $\mathcal{O}(M^3/3)$. However, it is limited to Hermitian and positive-definite input matrices. In our case we can assume that these properties are satisfied since:

- each covariance matrix is certainly Hermitian by construction;
- if the set of secondary range cells is sufficiently large (of the same order of magnitude as M), the covariance matrices are also positive-definite (see [46]).

In the following section we investigate different variants of the Cholesky algorithm and their impact on shared-memory parallel architectures.

4.2 Improving cache utilization and parallelism

Working with parallel programs, the number of operations to be executed is not always the most meaningful metric. Given the memory hierarchy of any computer

Algorithm 1 Scalar algorithm.

1: **for** $k := 1$ **to** M **do**
2: $A_{k,k} := \sqrt{A_{k,k}}$
3: **for** $j := k+1$ **to** M **do**
4: $A_{j,k} := A_{j,k} * \overline{A_{k,k}}^{-1}$
5: **end for**
6: **for** $j := k+1$ **to** M **do**
7: **for** $i := j$ **to** M **do**
8: $A_{i,j} := A_{i,j} - A_{i,k} * \overline{A_{j,k}}$
9: **end for**
10: **end for**
11: **end for**

Algorithm 2 Tile algorithm.

1: **for** $k := 1$ **to** N_b **do**
2: $A(k,k) := Chol\,(A(k,k))$
3: **for** $j := k+1$ **to** N_b **do**
4: $A(j,k) := A(j,k) * A(k,k)^{-H}$
5: **end for**
6: **for** $j := (k+1)$ **to** N_b **do**
7: **for** $i := j$ **to** N_b **do**
8: $A(i,j) := A(i,j) - A(i,k) * A(j,k)^{H}$
9: **end for**
10: **end for**
11: **end for**

Figure 2.: Traditional (scalar) and Tile Cholesky decomposition algorithms.

architecture, it is of paramount importance to write algorithms that explicitly consider the presence of fast but small memories and their efficient exploitation. In the field of numerical algorithms this concept is well-known [47–50].

A "*de-facto*" standard for solving dense linear algebra problems is represented by the `LaPack` library [34]. The main goal of this library is to run efficiently on modern shared-memory architectures featuring multi-layered memory hierarchies. To do that, `LaPack` routines exploit *block algorithms* to reduce the amount of memory traffic by maximizing data reuse in higher level caches. The rationale is to recast the computations in a way that the most part is performed using matrix-matrix operations that can be highly optimized to exploit fully the cache hierarchy.

The resulting computational scheme can be parallelized by applying a *fork-and-join* pattern, in which parallelism is expressed inside matrix-matrix operations. This approach produces poor results on modern multicores [39]. To overcome this fact, the `PLASMA` library [38] introduces a new set of algorithms (named *tile algorithms* [39]) in which parallelism is expressed at a higher level, by re-organizing block algorithms using finer-granularity tasks operating on square tiles (blocks) of the matrix. As demonstrated in the literature, this approach makes it possible to achieve powerful trade-offs between cache locality and parallelism.

For our STAP parallelization purpose, we propose a comparison between the traditional Cholesky algorithm and the tiled variant. To have a better idea of the advantages of this class of algorithms, it is important to compare distinct versions in terms of number of operations and memory transfers. For the latter, we assume blocks (hereinafter we use block as a synonym of tile) of $B_s \times B_s$ scalar elements. Given $N_b = M/B_s$, the input matrix is composed of $N_b \times N_b$ blocks. The traditional and the tile versions of the right-looking Cholesky decomposition are described in Fig. 2. In the pseudo-code, $A_{i,j}$ indicates the element at row i and column j, whereas we denote with $A(i,j)$ the block with coordinates i,j. The structure of the algorithms is the same, except that in the tile version the operations involve sub-matrices instead of single complex values.

The algorithms work with the lower triangular part of the matrix A, and create the resulting L (i.e. $L \cdot L^H = A$) in place of A. At each iteration of the outermost loop, a single column of L is calculated. We start with the analysis of how many times each line of code is executed. We restrict our analysis to the execution of the innermost loop (lines 7-8), which determines the time complexity of the algorithms.

Expression (3) shows the number of times line 8 is executed, where n corresponds to the number of rows M for the traditional algorithm and the number of block rows N_b for the tile variant:

$$\sum_{k=1}^{n}\sum_{j=k+1}^{n}\sum_{i=j+1}^{n} 1 = \sum_{k=1}^{n}\sum_{j=k+1}^{n}(n-j) \simeq \sum_{k=1}^{n}\left[n^2 - nk - \frac{n^2+n}{2} - \frac{k^2+k}{2}\right]$$
$$\simeq \sum_{k=1}^{n}\left[\frac{n^2}{2} - nk + \frac{k^2}{2}\right] = \frac{n^3}{6} - \frac{n^2}{4} + \frac{n}{12} \tag{3}$$

In terms of the number of complex operations, the traditional version executes a multiplication and a sum at row 8. Things get more complicated with the tile version. At the same row we have a multiplication and a sum between sub-matrices, that we can estimate in the order of B_s^3 and B_s^2 operations. The total number of operations can be approximated as follows:

$$N_{op}^{trad} = 2\left(\frac{M^3}{6} - \frac{M^2}{4} + \frac{M}{12}\right) \simeq \mathcal{O}\left(\frac{M^3}{3}\right) \tag{4}$$

$$N_{op}^{block} = \left(B_s^3 + B_s^2\right)\left(\frac{N_b^3}{6} - \frac{N_b^2}{4} + \frac{N_b}{12}\right) \simeq \mathcal{O}\left(\frac{M^3}{6} + \frac{M^3}{6\,B_s}\right) \tag{5}$$

For the tile algorithm (5) we have substituted N_b with its definition M/B_s. These results allow us to appreciate that the size of the input matrix and the block size play a decisive role. With M relatively large compared to B_s, the tile algorithm performs in a similar way w.r.t the traditional version, having almost the same number of complex operations. On the other hand, with sufficiently large block sizes, the tile algorithm has a better multiplicative factor.

The second analysis consists in studying the two algorithms in terms of the memory hierarchy utilization: i.e. measuring the number of cache lines transferred from memory to caches during the execution. To this end we consider an abstract architecture featuring a single fully-associative cache with a line size of $\mathbf{C}$ bytes. Although this abstract view is far from the complexity of the current shared-memory architectures (featuring several levels of set-associative caches), this analysis is useful to understand the rationale behind the tile algorithm.

In the traditional algorithm, at each execution of line 8 we need to read three complex numbers (i.e. $A_{i,j}$, $A_{i,k}$ and $A_{j,k}$) and write the first one. We consider each complex number represented by a pair of single-precision floating-point values, for the real and the imaginary parts, for a total of 8 bytes. In terms of cache lines this means that at the worst case we need to access three cache lines. For the tile version the behavior is exactly the same, except that now elements are sub-matrices of $B_s \times B_s$ values. The access to a block not in cache requires an approximate number of transferred cache lines of $B_s^2/\mathbf{C}$. In order to have a better estimation we need to account for temporal locality. At line 8 of the scalar algorithm, all the

elements of the k-th column and of the k-th row of the original matrix are used for the iterations of the loop at line 7. With small enough matrices, we can assume that the entire column k and row k can stay completely in cache. Although this is a reasonable assumption for the traditional algorithm, for the tile version this could not be true anymore, as we are working with columns/rows of blocks. For this reason, for the traditional algorithm we suppose that each access to $A_{i,k}$ and $A_{j,k}$ never produces additional cache line transfers (except for the first access). For the tile algorithm we make a pessimistic assumption in which no temporal locality is exploited. The number of transferred cache lines can be approximated as follows:

$$N_{miss}^{trad} = \frac{8}{\mathbf{C}} \left(\frac{M^3}{6} - \frac{M^2}{4} + \frac{M}{12} \right) \simeq \mathcal{O}\left(\frac{M^3}{\mathbf{C}} \right) \tag{6}$$

$$N_{miss}^{block} = \frac{3 \cdot 8\, B_s^2}{\mathbf{C}} \left(\frac{N_b^3}{6} - \frac{N_b^2}{4} + \frac{N_b}{12} \right) \simeq \mathcal{O}\left(\frac{M^3}{\mathbf{C}\, B_s} \right) \tag{7}$$

As we can see, the tile algorithm reduces the theoretical number of cache misses by a factor B_s which is the square root of the block size. This reduction is negligible if we have large matrices and relatively small blocks. However, considering the size of existing caches and the fact that we only need a working set of few blocks, we can use relatively large block sizes that significantly reduce the amount of transferred data.

In conclusion, a proper selection of the block size is extremely important to have a decisive reduction in terms of number of operations and cache misses. However, finding the right block size for the target architecture is not obvious, but it depends on the structure of the cache hierarchy and the specific properties of the tile algorithm. Although attempts to define analytical models have been proposed over the last years [51], a common approach is to use heuristics or iterative compilation techniques [52], in which implementations using a limited set of various block sizes are compared in order to find the best configuration. In this paper, we will discuss the impact of different block sizes on the performance results achieved by our parallel implementations.

5. Test-bed Multi-core Architectures

In this section we describe three multi-core architectures that we have used to evaluate our parallelizations of the STAP computational kernel. For each architecture we focus on the most relevant hardware features, such as the organization in multiple chips and the interconnection networks, the number of cores and thread contexts, and the structure of the memory hierarchies.

Quad-CPUs Intel Xeon multi-processor. The first architecture is an Intel platform (Fig. 3) composed of four Xeon E7-4820 CPUs. Each CPU is a Westmere-Ex

architecture consisting in a multicore featuring 8 cores operating at 2 GHz clock rate. Each core supports two thread contexts (Hyper-Threading), and has access to a private L1 and L2 cache of size 32 KB (data) and 256 KB. The eight cores have access to a shared on-chip L3 cache of 18 MB. The four CPUs (for a total of 32 cores, 64 thread contexts) are interconnected in a complete fashion using the QuickPath network. Each CPU is equipped with a single memory controller using 4 different DDR3 channels.

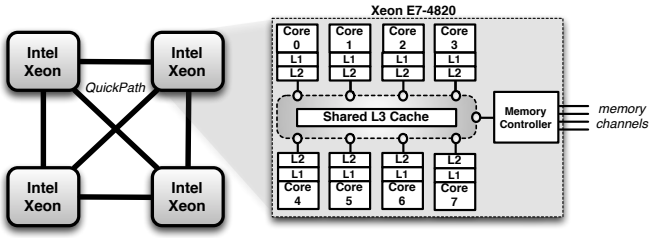


Figure 3.: Abstract representation of the Quad-CPUs Intel multi-processor.

Quad-CPUs IBM Power7 multi-processor. The IBM platform is a 8236-E8C Power server architecture composed of four Power7 CPUs as outlined in Fig. 4. Each Power7 is a multi-core architecture composed of eight out-of-order cores working at 3.3 GHz clock rate and equipped with a dedicated L1 and L2 cache of 32 KB (data) and 256 KB respectively. Each core features 4 thread contexts. The eight cores have access to a shared L3 cache of 32 MB. The four CPUs are interconnected by a complete crossbar network (for a total of 32 cores corresponding to 128 thread contexts). Each Power7 is equipped with two 4-channel DDR3 memory controllers for a total of 8 channels per CPU.

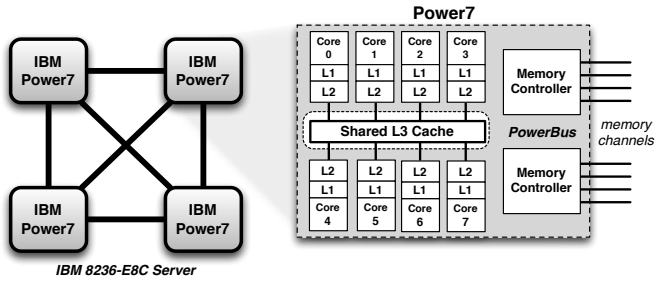


Figure 4.: Abstract representation of the Quad-CPUs Power7 multi-processor.

Dual-CPUs AMD Opteron multi-processor. The third execution platform is an AMD multi-processor architecture composed of two "Magny-Cours" Opteron 6176 CPUs. Each CPU is composed of two multi-core chips featuring six single-threaded cores bolted together (on two separated dies). Each core operates at 2.2 GHz clock

rate with a dedicated L1 and L2 cache of size 128 KB (data) and 512 KB per core and a L3 cache of 6 MB shared among the six cores of a chip. Interconnections between chips are implemented through several HyperTransport point-to-point links forming a crossbar network, as shown in Fig. 5. Each chip is equipped with one integrated memory controller enabling access to two DDR3 channels for a total of four memory channels per CPU.

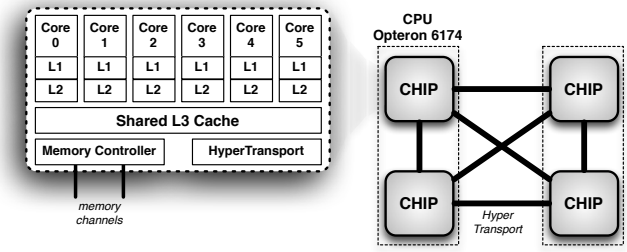


Figure 5.: Abstract representation of the Dual-CPUs AMD multi-processor.

6. Parallelization on Multi-core Architectures

STAP implementations require to operate within precise real-time constraints. In the ideal case, in order to process a number of cubes per time unit equal to the arrival rate λ, we need to replicate p^* times the whole weight vectors calculation phase, where $p^* = T_{weight} \cdot \lambda$ and T_{weight} denotes the time to calculate and apply the weights for each cell under test. Since the Cholesky decomposition dominates this phase, T_{weight} can be approximated as $T_{weight} \simeq \Gamma \cdot T_{fact}$, where T_{fact} indicates the time to apply a single decomposition.

Nevertheless, an effective parallelization usually needs to meet latency-oriented constraints in addition to reach proper levels of throughput. In real radar applications a computation latency of $\Gamma \cdot T_{fact}$ is not feasible. Therefore, more complex parallelizations need to be investigated. In the following sections we will describe different parallelism paradigms and how they can be exploited and composed in order to address this aspect of the STAP performance.

6.1 Task-farm parallelization

The computation latency to process a single CPI cube can be reduced by exploiting the fact that the calculation of each weight vector is independent of the others. The structured parallelism scheme that enables this behavior is the so-called *task-farm* pattern. The conceptual scheme (Fig. 6) of a task-farm is composed of: (i) a set of *workers* responsible for executing the Cholesky decomposition on different input matrices; (ii) an *emitter* responsible for receiving the covariance matrices and scheduling them to the workers according to a load-balancing strategy; (iii) a

collector that collects the weight vectors and transmits them to the other STAP phases.

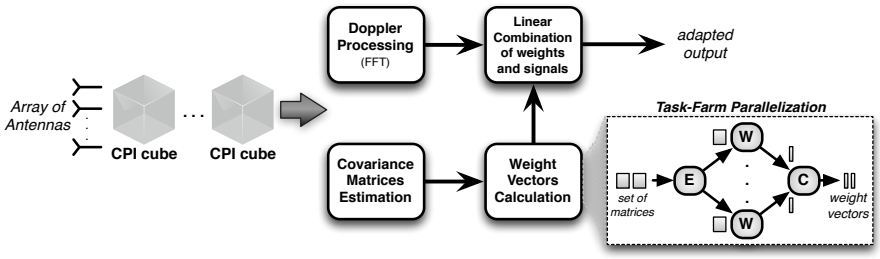


Figure 6.: STAP computational phases and task-farm parallelization.

The task-farm is a parallelism scheme able to reduce the completion time of a computation operating on a sequence of input tasks. If there do not exist architectural reasons that limit the scalability (such as the available memory bandwidth), this parallelism scheme is able to calculate, on average, a new weight vector every T_{fact}/p, where p is the number of workers. If the set of covariance matrices is much greater than the parallelism degree (i.e. $\Gamma >> p$), an approximation of T_{weight} is given by the following expression (the superscript denotes the parallelism degree):

$$T_{weight}^{(p)} \simeq \Gamma \cdot \frac{T_{fact}}{p} \tag{8}$$

Despite its simplicity, *this parallelization requires careful implementation to provide a near-optimal scalability on shared-memory architectures*. Important aspects are:

- the sequential algorithm executed by the workers. As we have seen in Section 4, several linear-algebra problems can be solved according to algorithms with different computational complexity and cache performance. Acceptable levels of performance with high parallelism degrees are possible if the memory bandwidth is exploited at best. To do that, a proper choice of the sequential algorithm is a first decisive aspect;
- the data structure layout and their allocation in memory. It is important to organize the data structures such that there is no undesired effect on the memory hierarchy (e.g. avoiding false sharing), and to efficiently exploit the memory bandwidth with an equal distribution of accesses to the available memory controllers (our test-bed architectures provide more than one interface - usually one or two per chip - to the memory).

In this section we evaluate our implementation on different multicores. We instantiate the task-farm scheme by using low-level mechanisms and libraries available in POSIX-compliant systems. The emitter, collector and the workers are implemented by a set of pthreads whose affinity is set on a corresponding number of dedicated cores using the `pthread_setaffinity_np` function.

Cooperation between threads has been implemented in a portable way by exploiting single/multiple-producer single-consumer *queues* protected by a spin-lock mechanism. Thread communication is performed *by reference*, i.e. through the exchange of memory pointers to shared data structures (e.g. matrices and vectors). This method has been already applied to stream-based parallel computations on multicores [31, 53, 54], with efficient performance especially for fine-grain parallel programs. Our communication mechanisms are widely portable, provided that proper implementations of locking mechanisms are implemented on the single architectures (e.g. using standard POSIX spin-locks).

Another aspect of the task-farm portability is the sequential code performed by workers. We use high-level `BLAS` and `LaPack` routines to execute the single phases of the Cholesky decomposition. Algorithm 3 shows the pseudo-code performed on each received matrix by a generic task-farm worker. The code can be compiled by linking *any* implementation of the standard `BLAS` and `LaPack` APIs.

Algorithm 3 *Tile Cholesky algorithm.*

for $k \leftarrow 1$ **to** N_b **do**
  `//A(k,k) = Scalar_Fact(A(k,k))`
  $A(k,k) :=$ **cpotf2**$(A(k,k))$
  for $j \leftarrow k+1$ **to** N_b **do**
    `//A(j,k) = A(j,k)/A(k,k)`^{H}
    $A(j,k) :=$ **cblas_ctrsm**$(A(j,k),\ A(k,k))$
  for $j \leftarrow k+1$ **to** N_b **do**
    `//A(j,j) = A(j,j)-A(j,k)*A(j,k)`^{H}
    $A(j,j) :=$ **cblas_cherk**$(A(j,j),\ A(j,k))$
    for $i \leftarrow j+1$ **to** N_b **do**
      `//A(i,j) = A(i,j)-A(i,k)*A(j,k)`^{H}
      $A(i,j) :=$ **cblas_cgemm**$(A(i,j),\ A(i,k),\ A(j,k))$

The code can be statically/dynamically linked with the most performant library available for the target architecture, without modifying our source code, thus enabling a high code portability. This allows us to analyze the task-farm performance using different versions of the libraries on the same architecture, or to compare the performance reached on different architectures as well.

A general overview of the task-farm in terms of threads and their interconnections is shown in Fig. 7. The emitter schedules each matrix according to an on-demand strategy, providing the initial memory address of the data structure of the current task to an available worker. The availability is explicitly notified through special messages using a multi-producer single-consumer availability queue.

The next sections are devoted to describing the performance of the task-farm on our test-bed architectures. In the experiments an important parameter is the size of the covariance matrices. To be independent from the specific STAP application, in this paper we focus on four possible sizes (128×128, 256×256, 512×512 and 1024×1024 single precision complex elements) in order to evaluate our parallelization in different conditions of computational grain and memory utilization.

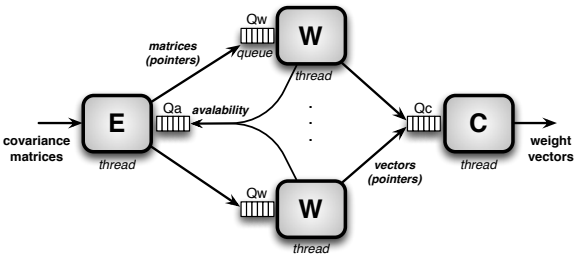


Figure 7.: Details of the task-farm implementation: threads and queues.

6.1.1 Intel experiments

On the Intel architecture, our task-farm code has been linked with the `MKL` library (version 11.0). The library uses the SSE2 SIMD instructions and, on more recent machines, the AVX instructions enabling up to three 256 bit operands.

As stated in [39], the matrix should be arranged in a block data layout to achieve the best cache utilization. At the beginning of the execution the input matrix is copied from the original layout (e.g. column-wise) to the block data layout, and then copied back when the decomposition has been completed. These phases (which are part of the decomposition) are not shown in the pseudo-code for brevity.

Tab. 1 summarizes the results of the mean decomposition time T_{fact}, averaging over a large number of executions. For the traditional algorithm we use a block size equal to the entire matrix (i.e. no tiling is actually used).

Block / Matrix	$\mathbf{128 \times 128}$	$\mathbf{256 \times 256}$	$\mathbf{512 \times 512}$	$\mathbf{1024 \times 1024}$
$\mathbf{8 \times 8}$	0.791	5.977	51.56	418.23
$\mathbf{16 \times 16}$	0.530	3.314	23.51	180.45
$\mathbf{32 \times 32}$	0.461	2.673	17.80	128.02
$\mathbf{64 \times 64}$	0.435	2.494	16.16	112.21
$\mathbf{128 \times 128}$	0.411	2.46	15.78	106.55
$\mathbf{256 \times 256}$	-	2.698	16.37	107.64
$\mathbf{512 \times 512}$	-	-	17.99	112.94
$\mathbf{1024 \times 1024}$	-	-	-	129.98

Table 1.: Decomposition time (milliseconds) on the Intel multi-processor.

We can observe that there exists an *optimal block size* depending on the library implementation and the characteristics of the cache hierarchy. Blocks of 128×128 elements give smaller decomposition times for any matrix size. The decomposition time T_{fact} has a non-monotonic behavior as a function of the block size: i.e. by increasing the block size we achieve a reduction of T_{fact} up to a certain point. After that, larger block sizes are completely useless and lead to higher decomposition times.

When applying the tile algorithm it is important to ensure that the working set fits into the lower levels of the memory hierarchy (L1 and L2 caches). 128×128 blocks are the largest size such that the working set is mostly contained in the second level cache (of 256 KB). With larger blocks the increase in the working set becomes dominant, and the resulting number of L2 cache misses leads to higher cal-

culation times. With small blocks the calculation time is significantly higher than the traditional algorithm. The main reason for this is mostly due to a greater number of calls to the `BLAS` and `LaPack` routines, which induces a significant overhead w.r.t using larger block sizes.

We evaluate the parallelization using two measures. The first one is the *throughput* $\mathcal{T}$, i.e. the average number of processed tasks per second. The second metric is the *scalability*, a relative metric of how a parallel version accelerates compared to the version with a single worker:

$$\mathcal{S}^{(p)} = \frac{\mathcal{T}^{(p)}}{\mathcal{T}^{(1)}} \tag{9}$$

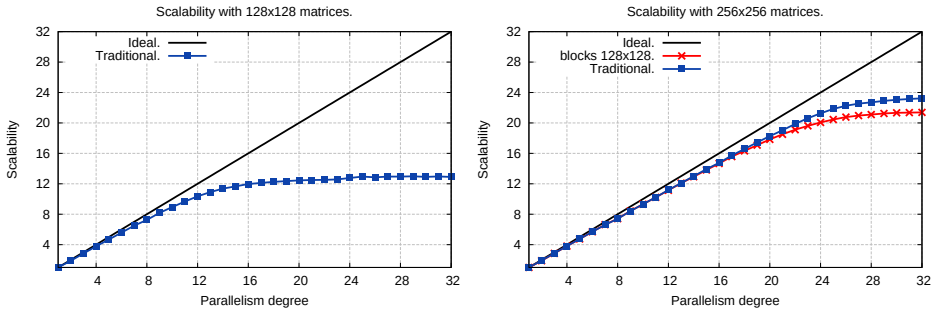
i.e, it is measured as the ratio between the throughput with parallelism degree p and using only one worker.

In order to study the maximum sustained level of throughput, the arrival rate of covariance matrices is infinitely fast (high enough) in our experiments.

Fig. 8 shows the scalability results. We compare the task-farm scalability using the traditional version of the Cholesky decomposition and the tile version with the best block size measured in Tab. 1. As we can observe, the tile algorithm performs better when the set of currently calculated matrices by the workers do not entirely fit in the cache hierarchy. Given the large L3 caches of the Intel multi-processor (18 MB per CPU, i.e. per 8 cores) this happens with 1024×1024 matrices and using a sufficiently high parallelism degree. In this case it is not possible to fully store all the matrices on which workers are concurrently operating on the L3 caches, and the tile algorithm achieves better scalability since it reduces the working set of the program leading to smaller traffic with the main memory. Conversely, when the matrices entirely fit the last level cache, the advantage of the tile algorithm in terms of memory traffic becomes negligible, since the number of L3 cache misses is nearly the same.

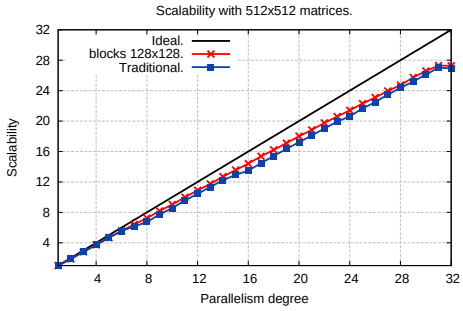
Nevertheless, the scalability is unsatisfactory with large parallelism degrees and far from the ideal one. To understand the reason, we can make a simple analysis of the exploited memory bandwidth of our application. The Cholesky decomposition features a time complexity which is one order of magnitude greater than the space complexity. This means that we can obtain a better scalability with bigger matrices (up to 512×512) because the number of floating-point operations increases faster than the memory requirement. Suppose that the task-farm is executed with the maximum parallelism degree. As stated before, the L3 caches are able to store almost the entire set of currently calculated matrices (except in the 1024×1024 case). This means that after the initial load of the matrix cache lines (compulsory cache misses), the number of main memory accesses is negligible. For each decomposition we need to load the lower triangular part of the matrix and write the decomposed matrix back to the memory. Therefore, we need to transfer $(M^2\, 8)/2$ bytes from the memory and the same amount of data back to the memory.

Tab. 2 shows the throughput and the required memory bandwidth to sustain it for three matrix sizes. We can clearly see that the memory bandwidth required by the 128×128 case to scale 13 is far higher than that required by the 512×512 case

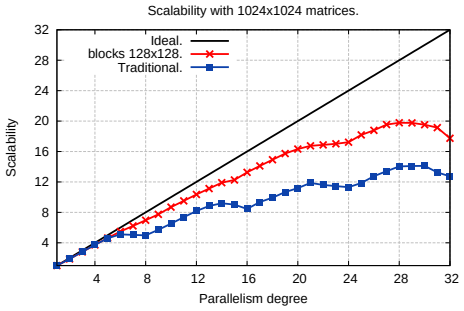


(a) Scalability 128×128.

(b) Scalability 256×256.



(c) Scalability 512×512.



(d) Scalability 1024×1024.

Figure 8.: Scalability of task-farm on the Intel platform.

to scale close to ideal. This demonstrates our expectations: the required memory bandwidth decreases by increasing the matrix size. Moreover, the tile algorithm requires a slightly higher bandwidth, since due to a better utilization of the L1 and L2 caches the decomposition time becomes smaller and requests to the main memory are issued more frequently. It is worth noting that with 128×128 and 256×256 matrices the required bandwidth is close to 4 GB/s. For this reason we can suppose that, for the access pattern of our application, this represents the physical limit of the memory sub-system.

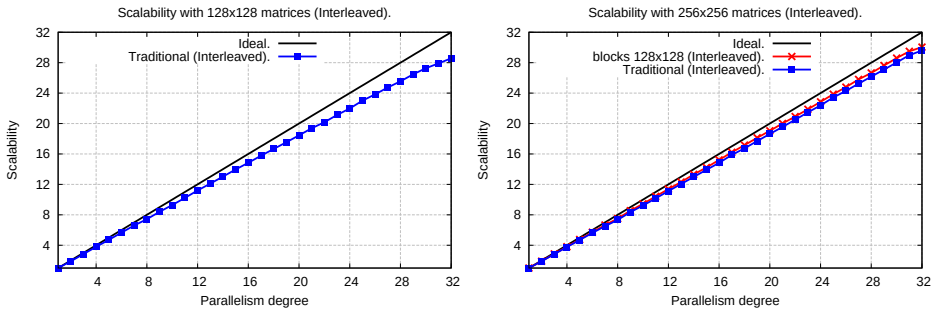
Matrix	**Block**	**Throughput** (matrix per sec.)	**Scalability**	**Required Mem. Bandwidth.**
128×128	128×128	31,515	12.98	3.85 GB/s
256×256	128×128	8,701	21.39	4.25 GB/s
	256×256	8,610	23.23	4.2 GB/s
512×512	128×128	1,729	27.30	3.37 GB/s
	512×512	1,500	27.02	2.93 GB/s

Table 2.: Task-farm memory bandwidth requirement using 32 workers.

The table does not show the result of using 1024×1024 matrices. In this case, due to the matrix size (8 MB per matrix, for a total amount of 256 MB using 32 workers) the set of currently calculated matrices can not be completely maintained in the L3 caches, resulting in a significantly higher number of transfers to the main memory. In this situation the tile version gives a clear advantage by reducing the working set and thus the number of cache misses.

Since the memory bandwidth is the bottleneck to provide scalable implementations, we exploit a clever allocation of the covariance matrices in main memory. Previous results are based on a simple allocation policy: the emitter thread receives matrices from the other STAP phases through high-speed interconnection networks (e.g. Infiniband or Myrinet). Each matrix is copied into a local buffer of the emitter thread, that is allocated on the memory controller nearest to the CPU on which the emitter is executed.

A better utilization of the aggregate bandwidth of the memory controllers can be achieved by allocating the virtual pages in a round-robin fashion among all the nodes on the system, i.e. with an equal distribution of accesses among the four memory controllers of the Intel multi-processor. We call this strategy *Interleaved Allocation Policy*, and it can be easily implemented using the standard `NUMA Library` [55, 56]. Fig. 9 shows the importance of this solution that provides near-optimal results for any matrix size.



(a) Scalability 128×128.

(b) Scalability 256×256.

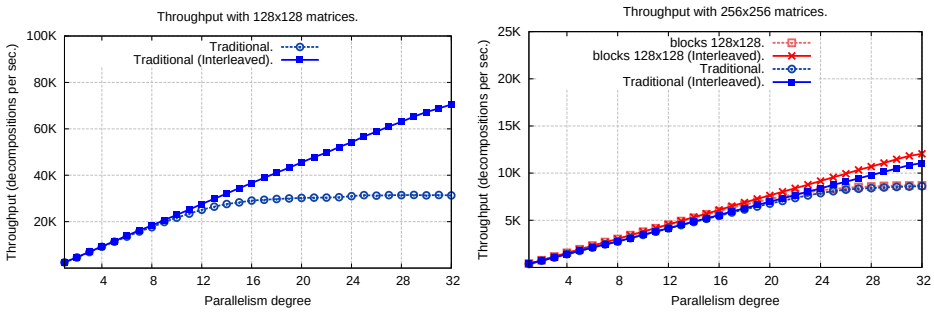
(c) Scalability 512×512.

(d) Scalability 1024×1024.

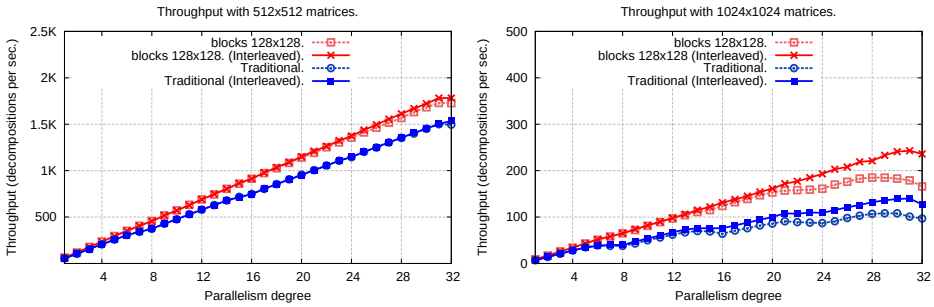
Figure 9.: Scalability of task-farm on the Intel platform using the Interleaved Allocation Policy.

In terms of throughput, the importance of the tile algorithm is much more evident (see Fig. 10). Using the tile algorithm we start from a better sequential version (a smaller T_{fact}) and the interleaved allocation makes it possible to reach near-optimal scalability levels. Therefore, the task-farm performing the tile algorithm outperforms the task-farm using the traditional algorithm, achieving a higher number of completed matrices per second.

For the sake of completeness, Tab. 3 depicts the behavior of the task-farm using different block sizes. The table shows the throughput and the corresponding



(a) Throughput 128×128. (b) Throughput 256×256.



(c) Throughput 512×512. (d) Throughput 1024×1024.

Figure 10.: Throughput of task-farm on the Intel platform.

scalability using 32 workers (the scalability is indicated in parentheses). Using more threads than the number of cores (exploiting multiple thread contexts) is completely useless in this experiment, since the tile Cholesky is a floating-point intensive workload that does not benefit from HyperThreading.

Using the interleaved allocation policy the scalability is acceptable for any matrix and block size. However, the block size of 128×128 is the best one, providing the best throughput.

In conclusion, a proper selection of the best block size and a careful allocation of matrices are decisive aspects in designing scalable task-farm parallelizations of the weight vectors calculation phase.

Block / Matrix	**128×128**	**256×256**	**512×512**	**1024×1024**
8×8	36,941 (29.83)	4,978 (29.94)	565 (29.78)	44 (18.95)
16×16	54,794 (29.12)	8,978 (30.03)	1,270 (29,97)	136 (24.21)
32×32	62,189 (28.7)	11,025 (29.71)	1,653 (29.57)	206 (26.27)
64×64	64,683 (27.99)	11,781 (29.76)	1,759 (28.79)	239 (26.94)
128×128	70,472 (28.58)	12,049 (30.05)	1,782 (28.26)	244 (26.01)
256×256	-	11,072 (29.59)	1,700 (28.03)	242 (26.38)
512×512	-	-	1,535 (27.99)	233 (26.60)
1024×1024	-	-	-	140 (18.37)

Table 3.: Maximum throughput and scalability (in parentheses) on the Intel multi-processor.

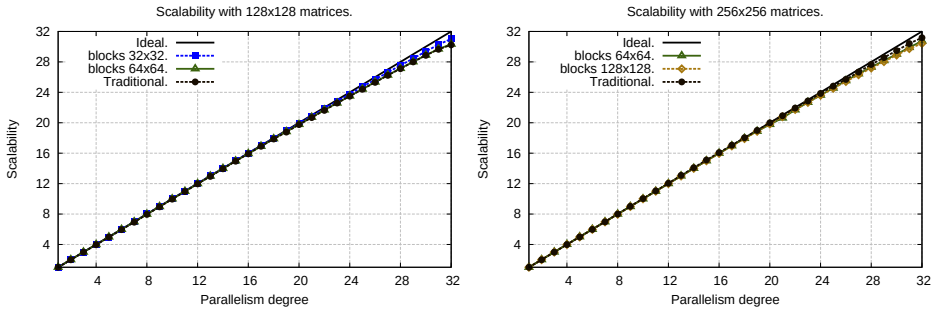
6.1.2 IBM experiments

On the IBM architecture the task-farm code has been compiled by statically linking the `GotoBLAS2` library, a multi-platform `BLAS/LaPack` implementation providing comparable results with `MKL` on Intel. The sequential decomposition times of the tiled Cholesky are shown in Tab. 4.

Block / Matrix	**128 × 128**	**256 × 256**	**512 × 512**	**1024 × 1024**
8 × 8	0.665	4.557	34.05	358.16
16 × 16	0.407	2.499	17.32	148.84
32 × 32	0.326	1.909	12.57	97.81
64 × 64	0.315	1.724	10.91	79.45
128 × 128	0.366	1.777	10.62	73.02
256 × 256	-	2.552	12.27	73.85
512 × 512	-	-	18.70	87.67
1024 × 1024	-	-	-	159.33

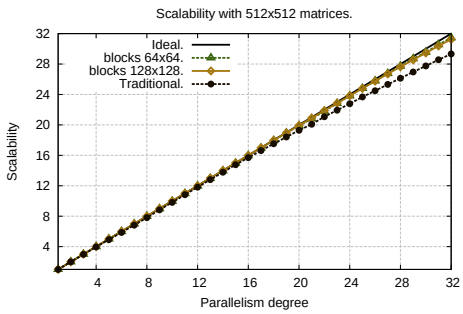
Table 4.: Decomposition time (milliseconds) on the IBM multi-processor.

On the IBM multi-processor the decomposition time is from 15% to 50% smaller than using the `MKL` library. This is due to a more powerful SIMD sub-system compared to the Intel Nehalem. The `MKL` implementation exploits SSE3 instructions enabling 128 bit operands (up to 4 single precision floating-point numbers). On the Power7, the VSX extension features the same operand length, but the architecture is equipped with four distinct floating-point units able to perform up to two vec-

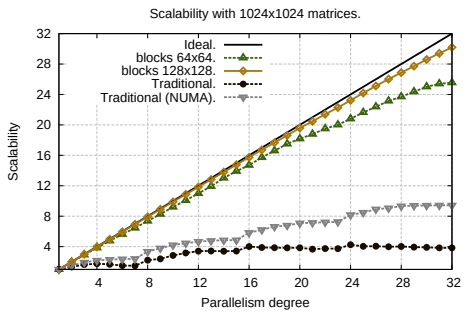


(a) Scalability 128 × 128.

(b) Scalability 256 × 256.



(c) Scalability 512 × 512.



(d) Scalability 1024 × 1024.

Figure 11.: Scalability of the task-farm on the IBM Power7 multi-processor.

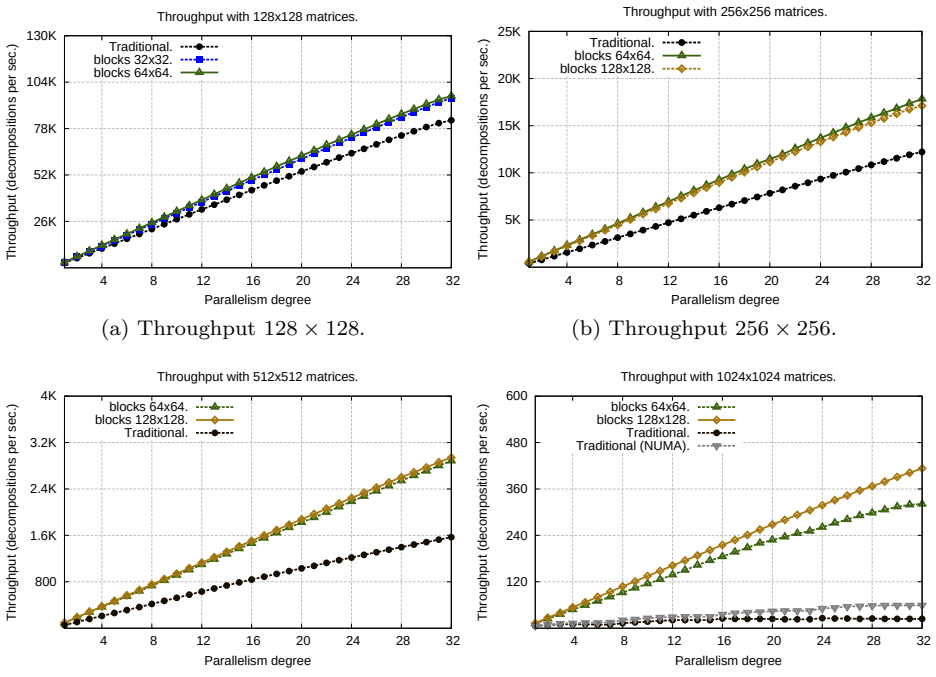
tor operations in parallel. This, coupled with a higher clock frequency, allows the Power7 to outperform the other considered architectures.

From a qualitative viewpoint, the results are similar to the Intel multi-processor. For every matrix size there exists an optimal block size. For 128×128 and 256×256 matrices the best block size is using blocks of 64×64 elements. For larger matrices the best size is the same as the Intel (i.e. 128×128 blocks). This difference is probably due to the different implementation of the `BLAS/LaPack` routines.

Fig. 11 shows the scalability on the IBM multi-processor. Also on this architecture we have no performance improvement by using more thread contexts per core, so we consider 32 workers as the maximum parallelism degree.

With 128×128, 256×256 and 512×512 matrices the scalability is near to the ideal one. Compared to the Intel multi-processor, now satisfactory results can be achieved without using the interleaved allocation. This is due to a more efficient memory sub-system, which uses two faster memory controllers per chip; in this way the memory bandwidth of a single chip is enough to guarantee a near-optimal scalability.

With 1024×1024 matrices the results are extremely interesting (see Fig. 11d). In this case the scalability is, again, limited by the memory bandwidth. The traditional algorithm is not able to provide good scalability neither with the interleaved allocation. On the other hand, the tile algorithm is able to achieve near-optimal results also with the standard allocation policy.



(a) Throughput 128×128.

(b) Throughput 256×256.

(c) Throughput 512×512.

(d) Throughput 1024×1024.

Figure 12.: Task-farm throughput on the IBM Power7 multi-processor.

Fig. 12 and Tab. 5 describe the throughput achieved by the task-farm. The block size achieving the smaller decomposition time also provides the best performance

Block / Matrix	**128 × 128**	**256 × 256**	**512 × 512**	**1024 × 1024**
8 × 8	47,505 (31.59)	7,018 (31.98)	934 (31.81)	29 (10.58)
16 × 16	76,982 (31.37)	12,759 (31.88)	1,802 (31.23)	83 (12.48)
32 × 32	95,057 (31.02)	16,597 (31.68)	2,514 (31.60)	170 (16.72)
64 × 64	96,246 (30.32)	17,834 (30.75)	2,884 (31.46)	321 (25.56)
128 × 128	82,644 (30.24)	17,120 (30.42)	2,942 (31.24)	419 (30.20)
256 × 256	-	12,217 (31.18)	2,444 (29.99)	413 (30.98)
512 × 512	-	-	1,568 (29.33)	344 (30.19)
1024 × 1024	-	-	-	59 (9.42)

Table 5.: Maximum throughput and scalability on the IBM multi-processor.

with the maximum parallelism degree.

6.1.3 AMD experiments

On the AMD architecture the task-farm code can be compiled using both the `MKL` and `GotoBLAS2` library. From our experiments, we achieve slightly better results using the `GotoBLAS2` library, since `MKL` is optimized only for Intel architectures. Tab. 6 shows the T_{fact} using different block and matrix sizes.

As for the IBM multi-processor, the `GotoBLAS2` implementation of the tile Cholesky provides the best results with blocks of 64×64 elements with 128×128 and 256×256 matrices. For larger matrices, the smallest decomposition times are achieved using blocks of 128×128 elements.

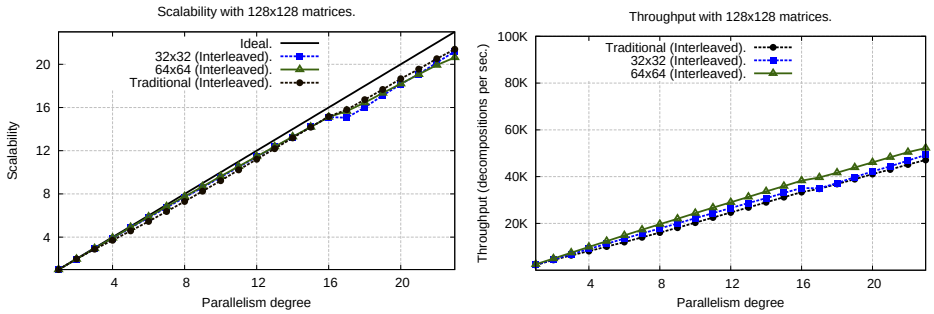
Block / Matrix	**128 × 128**	**256 × 256**	**512 × 512**	**1024 × 1024**
8 × 8	1.022	7.246	55.44	476.99
16 × 16	0.550	3.580	26.85	217.14
32 × 32	0.422	2.617	18.61	140.88
64 × 64	0.377	2.308	15.66	114.70
128 × 128	0.437	2.422	15.18	105.91
256 × 256	-	2.823	17.62	110.14
512 × 512	-	-	28.35	133.84
1024 × 1024	-	-	-	228.57

Table 6.: Decomposition time (milliseconds) on the AMD multi-processor.

Fig. 13 shows the scalability and the throughput with different block and matrix sizes. Similar to the Intel architecture, all the CPUs of the AMD multi-processor are equipped with one memory controller; consequently, the Interleaved Allocation Policy is extremely important. In fact, given the relatively small size of L3 cache, a tile version is required also for matrices of 512×512 to obtain good scalability results. Tab. 7 summarizes the best throughput and scalability results.

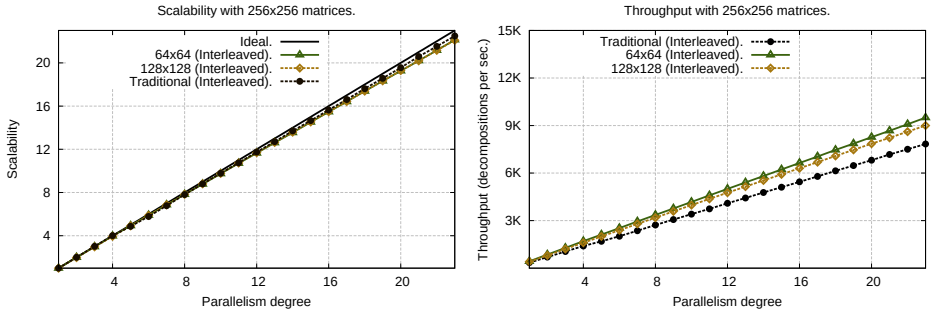
We conclude this section by comparing the results achieved on the three test-bed architectures. Tab. 8 and Tab. 9 summarize the best T_{fact} and the best throughput. As we can observe, the best results are always achieved using the IBM architecture which outperforms the other multi-core platforms.

On the IBM multi-processor our task-farm implementation scales perfectly without using the Interleaved Allocation Policy (the two integrated controllers of a CPU provide sufficient bandwidth for our application). Besides memory bandwidth, the level of throughput achieved on the IBM multi-processor is due to a more powerful



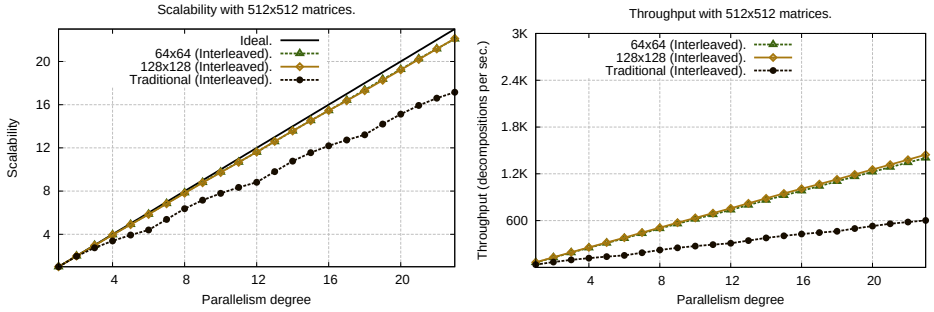
(a) Scalability 128×128.

(b) Throughput 128×128.



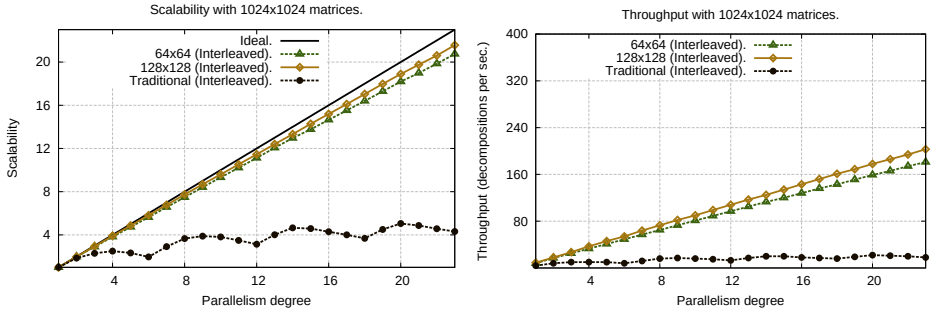
(c) Scalability 256×256.

(d) Throughput 256×256.



(e) Scalability 512×512.

(f) Throughput 512×512.



(g) Scalability 1024×1024.

(h) Throughput 1024×1024.

Figure 13.: Task-Farm results on the AMD multi-processor using the Interleaved Allocation Policy.

cache hierarchy (featuring larger L3 caches w.r.t the Intel and the AMD architectures) and a higher clock frequency.

Block / Matrix	**128 × 128**	**256 × 256**	**512 × 512**	**1024 × 1024**
8 × 8	22,517 (23.31)	3,244 (23.49)	409 (22.93)	38 (18.37)
16 × 16	39,432 (21.99)	6,312 (22.76)	824 (22.34)	87 (18.82)
32 × 32	49,285 (21.19)	8,435 (22.36)	1,191 (22.28)	139 (19.68)
64 × 64	52,246 (20.63)	9,496 (22.13)	1,403 (22.08)	181 (20.74)
128 × 128	47,103 (21.39)	8,998 (22.09)	1,445 (22.13)	203 (21.57)
256 × 256	-	7,836 (22.49)	1,229 (21.79)	199 (21.85)
512 × 512	-	-	602 (17.15)	149 (19.88)
1024 × 1024	-	-	-	22 (5.05)

Table 7.: Maximum throughput and scalability on the AMD multi-processor.

	128 × 128	**256 × 256**	**512 × 512**	**1024 × 1024**
`Intel`	0.411	2.458	15.78	106.55
`AMD`	0.377	2.308	15.18	105.91
`IBM`	0.315	1.724	10.62	73.02

Table 8.: Best decomposition times (milliseconds) on the three multi-processors.

	128 × 128	**256 × 256**	**512 × 512**	**1024 × 1024**
`Intel`	70,472	12,049	1,782	243
`AMD`	52,246	9,496	1,445	203
`IBM`	96,246	17,834	2,942	413

Table 9.: Best throughput on the three multi-processors (decompositions per seconds).

6.1.4 Limitations of the task-farm approach

As previously demonstrated, with a careful implementation the task-farm achieves good performance portability on a broad range of multi-core architectures. However, the task-farm behavior can become inefficient when the number of tasks to compute is not large enough and in general comparable with the available number of cores (which is in line with the current tendency of multi-core CPUs). In this regard, two aspects deserve special consideration:

- the number of covariance matrices of a CPI cube limits the maximum theoretical parallelism degree exploitable by the task-farm;
- the completion time T_{weight} achieved by the task-farm can be greatly influenced by workers that receive additional tasks (e.g. if the number of matrices is not perfectly divisible by the number of workers).

Fig. 14 shows the ideal behavior of a task-farm with 4 workers. Let us suppose that the emitter schedules each matrix according to an on-demand policy with a constant scheduling time T_{sched}, and each worker applies the decomposition with

a calculation time T_{fact}. We can observe that the flow of results is produced in "batches": i.e. when the first worker produces a result, the others finish their current work every T_{sched}. Assuming a scheduling time negligible w.r.t T_{fact}, a more precise estimation of T_{weight} is given by:

$$T_{weight}^{(p)} \simeq \left\lceil \frac{\Gamma}{p} \right\rceil \cdot T_{fact} \tag{10}$$

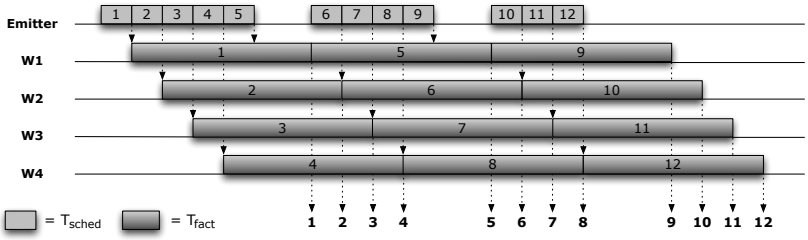


Figure 14.: Ideal behavior of the task-farm with 4 workers.

When the number of matrices Γ is a multiple of the parallelism degree p, the two expressions (8) and (10) give the same result. Otherwise T_{weight} is limited by the workers that receive additional tasks. For instance with 4 workers, computing 5 or 8 matrices requires almost the same time. This behavior is much more important if we consider parallel architectures featuring a large set of cores.

To exemplify this concept, we consider a numerical example using an abstract architecture with 32 cores. Let us suppose that we need to compute $\Gamma = 100$ cells under test per CPI cube. If T_{fact} is equal to 70 ms, the computation latency of a CPI cube using 32 workers can be estimated as follows:

$$T_{weight}^{(32)} \simeq \left\lceil \frac{100}{32} \right\rceil \cdot T_{fact} = 4 \cdot T_{fact} = 4 \cdot 70 \text{ ms} = 280 \text{ ms}$$

Better results can be achieved if we consider a more complex parallelization in which each task-farm worker, instead of being sequential, implements an internal parallelization able to reduce the T_{fact} to complete a single decomposition. This approach exploits one of the most important properties of SPP, i.e. *the composability of parallelism schemes and their capability to be nested in complex and hierarchical structures*. Fig. 15 outlines the nesting between the task-farm scheme and the internal parallelization of each worker (exploiting a *data-parallel paradigm* as will be discussed in Section 6.2).

If we use two cores to halve the calculation time of a single decomposition, we are able to achieve a T_{weight} expressed as follows:

$$T_{weight}^{(16)} \simeq \left\lceil \frac{100}{16} \right\rceil T_{fact}^{(2)} = 7 \cdot T_{fact}^{(2)} = 7 \cdot 35 \text{ ms} = 245 \text{ ms}$$

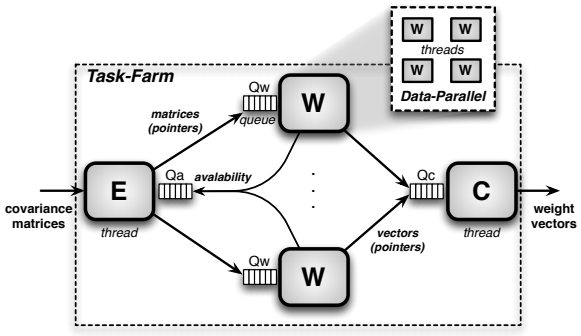


Figure 15.: Nesting of task-farm and data-parallel schemes.

With 16 task-farm workers, each one internally parallelized using two cores, we are able to improve the computation latency of 12.5% w.r.t the solution with a pure task-farm scheme exploiting the same number of cores. Greater benefits can be achieved if we can use a higher parallelism degree inside each decomposition task. For instance with 8 task-farm workers each one using four cores, we obtain the following result:

$$T^{(8)}_{weight} \simeq \left\lceil \frac{100}{8} \right\rceil \cdot T^{(4)}_{fact} = 13 \cdot T^{(4)}_{fact} = 13 \cdot 17.5 \text{ ms} = 227.5 \text{ ms}$$

In this way T_{weight} is reduced by 18.75% compared to the initial result. In conclusion, the more the integer ratio between the number of covariance matrices and the task-farm parallelism degree is smaller, the more the additional work to a set of workers influences the completion time T_{weight}. In this case a nesting between the task-farm and the data-parallel scheme can be useful to improve the computation latency. In the rest of this paper we present our data-parallel parallelization and we propose a nesting of the data-parallel approach inside our task-farm structure.

6.2 Data-parallel parallelization

Parallelizations of linear algebra problems like LU, QR and Cholesky decompositions have been extensively studied over the last years. De-facto standards are represented by optimized libraries providing efficient parallelizations for shared-memory multicores (notably the `PLASMA` library [38]) and distributed-memory platforms (e.g. `ScalaPack` [43, 57]). Such parallel routines exploit tile algorithms to achieve a finer granularity parallelism, and are essentially based on the data-parallel paradigm. On multi-core architectures the `PLASMA` library represents a valuable work. Linear algebra computations are abstracted by directed acyclic graphs, where nodes represent computational kernels (*tasks*) and arcs represent dependencies among them (the so-called *stencil pattern*).

The efficiency of a parallelization is greatly influenced by the way in which tasks are mapped onto real executors. For dense linear algebra problems such as the Cholesky decomposition, the assignment of tasks to parallel workers is extremely

challenging: on one hand the adopted scheduling strategy should achieve a load balanced execution; on the other hand we need to guarantee that parallel workers have some work to do at every execution instant.

To give a better idea, Fig. 16 sketches the dependencies between blocks during the steps of the algorithm. The `PLASMA` library provides two scheduling strategies:

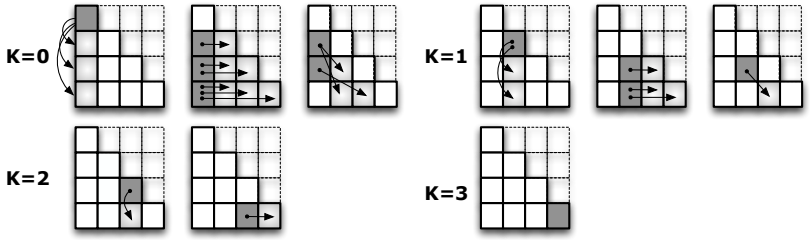


Figure 16.: Data dependencies between blocks at each step k of the tile Cholesky decomposition (matrix composed of 4×4 blocks).

- a *static scheduling*, in which the assignment of tasks to workers is predetermined at compile time. Tasks are grouped into jobs composed of a row of blocks at a specific iteration index of the computation. Jobs are statically assigned to workers in a round-robin fashion (more details are available in [40, 41]). The static scheduling simplifies the run-time support but it is tightly tailored to the specific algorithm. Grouping tasks in rows also affects negatively the total amount of parallelism available;
- a *dynamic scheduling*, in which tasks are scheduled as they become available. The run-time support requires complex data structures to manage the availability of tasks and their data dependencies, and an efficient procedure to explore the graph searching for new tasks to assign to the available workers. This solution allows more tasks to be executed concurrently at the cost of a heavier support. Therefore, it is effective only when the computational cost of a single task is sufficiently greater than the scheduling overhead. Fig. 17 exemplifies the Cholesky graph of a 4×4 blocks matrix.

Also in this case the block size plays an important role. Larger blocks reduce the scheduling complexity (fewer tasks mean also smaller data structures to manage the task dependencies and their availability). However, as the size of the blocks increases, the number of tasks decreases, suggesting that a proper trade-off between task granularity and scheduling efficiency must be made. Moreover, as we have seen in the previous sections, the sequential decomposition time increases using smaller blocks. Consequently, smaller blocks can be used iff the data-parallel parallelization scales sufficiently to outperform a parallel version using larger blocks. An example of the `PLASMA` parallel Cholesky is described in Fig. 18. We show the scalability on the Intel multi-processor using matrices of different sizes. For the sake of readability, the results are shown up to 16 data-parallel workers.

We can notice that the scalability is far from the ideal one and it is extremely poor especially with 256×256 matrices. The best result is a scalability of 3.35 using

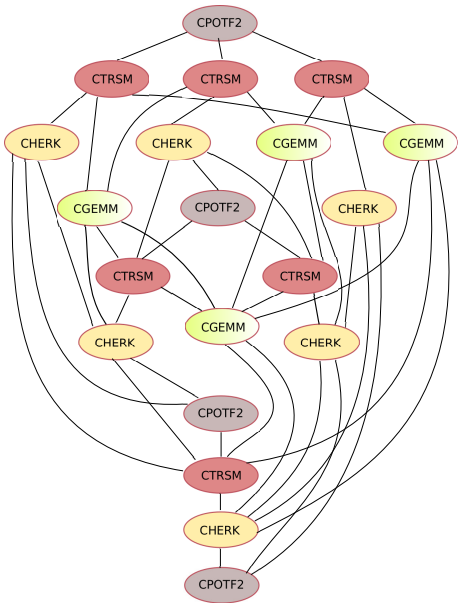


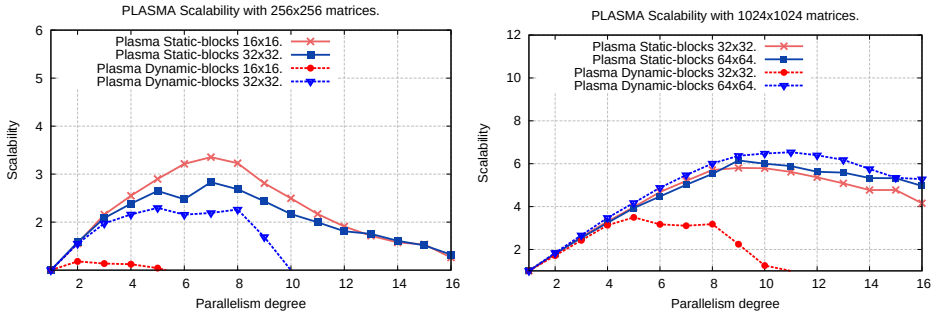
Figure 17.: Cholesky's directed graph of computational kernels and data dependencies (matrix composed of 4×4 blocks).

7 cores, which implies an efficiency of 0.47. Slightly better results can be achieved using 1024×1024 matrices (the best case is a scalability of 6 using 10 cores). We can notice that the dynamic scheduling is effective starting from 64×64 blocks. For smaller blocks the scheduling overhead is too high w.r.t the cost of a single task, and the static scheduling works better.

From this analysis we can make two important observations:

- as stated in the literature [40, 41], `PLASMA` parallelizations provide a satisfactory scalability with relatively large matrices (e.g. matrices of 4096×4096 elements or greater) and using a block size that optimizes the trade-off between task granularity and scheduling complexity. However, in real STAP scenarios the covariances matrices are formed by few hundreds of rows and the case of 1024×1024 matrices is already a limit case;
- as explained in Section 6.1.4, in order to minimize the completion time of a CPI cube processing we make a nesting between the task-farm scheme and the data-parallel pattern. When considering to increase the data-parallel parallelism degree, *we need to account for the fact that fewer cores will be available to instantiate task-farm workers*. Therefore, new data-parallel workers should be added iff the performance gain is well balanced with the reduction of the maximum parallelism degree of the task-farm. Since in our cases the scalability of the `PLASMA` parallel decomposition rarely reaches the ideal one, even with small parallelism degrees, our nested approach will use a very small parallelism degree inside each task-farm worker (up to 4 cores for each task-farm worker).

To address the issues of this particular instantiation of the parallel Cholesky decomposition, in this paper we propose a novel data-parallel implementation. In



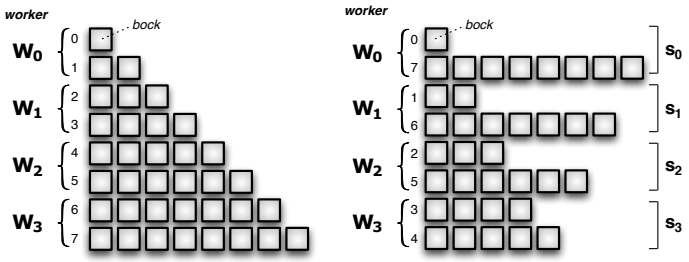
(a) Scalability 256×256. (b) Scalability 1024×1024.

Figure 18.: PLASMA parallel decomposition: scalability on the Intel multiprocessor.

order to scale as much as possible with a limited degree, we use a static assignment of blocks to data-parallel workers which is much simpler compared to the static and the dynamic scheduling strategies adopted by `PLASMA` parallelizations. With high parallelism degrees, our data-parallel version will not scale as the corresponding `PLASMA` implementation. However, it will be effective using relatively small parallelism degrees and, due to a more limited scheduling overhead, it will improve the parallel decomposition time compared to `PLASMA`.

Our implementation adopts a static assignment of blocks to data-parallel workers. Similarly to the `PLASMA` static scheduling, tasks are assigned to workers by grouping them in rows of blocks. We apply a task assignment that improves cache locality at the cost of a lower exploitable parallelism: throughout execution, each worker is responsible for computing the blocks of a specific, fixed set of block rows. This differs from the `PLASMA` static scheduling, where the same block row can be computed by distinct processors at different iterations.

Since the algorithm operates using the lower triangular part of the matrix, each row is composed of a different number of blocks. Therefore a *contiguous assignment*, consisting in assigning to each worker a set of contiguous rows, does not provide a balanced workload. As we can see in Fig. 19a, the first worker operates on a smaller set of blocks; a number that increases as we reach the last row of the matrix.



(a) Contiguous assignment. (b) Interleaved assignment.

Figure 19.: Different assignment strategies of blocks to data-parallel workers.

To partially solve this problem, we apply a static strategy based on an interleaved

mapping. This strategy is aimed at assigning a similar number of blocks to each data-parallel worker over the entire execution. It is worth noting that this approach could be sub-optimal. In fact, data dependencies between block computations may cause non-negligible waiting times in some cores, and an equal load balancing does not guarantee an equal computation time. Nevertheless, as we will show with experimental results, this approach is effective especially for small parallelism degrees without needing more complex and possibly dynamic scheduling strategies such as the ones of the `PLASMA` library.

The strategy identifies a set of pairs of block rows $\mathcal{S}_0, \ldots, \mathcal{S}_{(N_b/2)-1}$ each one defined as follows:

$$\mathcal{S}_i = \left\{A_{(i,*)},\ A_{(N_b-i-1,*)}\right\} \quad \text{for } i = 0, 2, \ldots, (N_b/2) - 1 \tag{11}$$

Pairs designed in such way have a very similar number of blocks. To further guarantee load balancing, the pairs are assigned to workers in an interleaved fashion: pair $\mathcal{S}_i$ is assigned to worker w_j iff $i \bmod p = \mathrm{j}$, where p is the data-parallel parallelism degree. Fig. 19b outlines the interleaved assignment of a matrix composed of 8 rows of blocks to four workers.

To respect the data dependencies, we use an approach similar to the one proposed in [58], in which the worker synchronization is performed by accessing a shared *lock-free* data structure called *precedence table*. This table contains a boolean flag for each block, which indicates if it has been definitely calculated or not. A worker that needs to read a block performs a busy-wait until the corresponding flag becomes true. Algorithm 4 describes the pseudo-code of our data-parallel implementation. At each computation step k, each worker operates on a partition of block rows

Algorithm 4 *Data-Parallel Tile Cholesky.*

```
for k ← 1 to N_b do
    foreach row i in My_Partition[k] do
        if i == k then
            //A(k,k) = Scalar_Fact(A(k,k))
            A(k,k) := cpotf2(A(k,k));
            progress_table[i, i] = true;
        else
            wait_until progress_table[k, k] == true;
            //A(j,k) = A(j,k)/A(k,k)^H
            A(j,k) := cblas_ctrsm(A(j,k), A(k,k))
    foreach row i in My_Partition[k] do
        //A(j,j) = A(j,j)-A(j,k)*A(j,k)^H
        A(j,j) := cblas_cherk(A(j,j), A(j,k))
    for j ← k + 1 to N_b do
        wait_until progress_table[j, k] == true;
        foreach row i in My_Partition[k] - {k} do
            //A(i,j) = A(i,j)-A(i,k)*A(j,k)^H
            A(i,j) := cblas_cgemm(A(i,j), A(i,k), A(j,k))
```

(denoted by $My\_Partition[k]$). The computation correctness is provided by using the shared flags of the precedence table. The `BLAS` calls are the same as the sequential algorithm. Both in the `PLASMA` versions and in our implementation we start

the computation with a parallel copy from the original layout to the block data layout and a copy back at the end of the computation (copies are not shown in the pseudo-code for brevity).

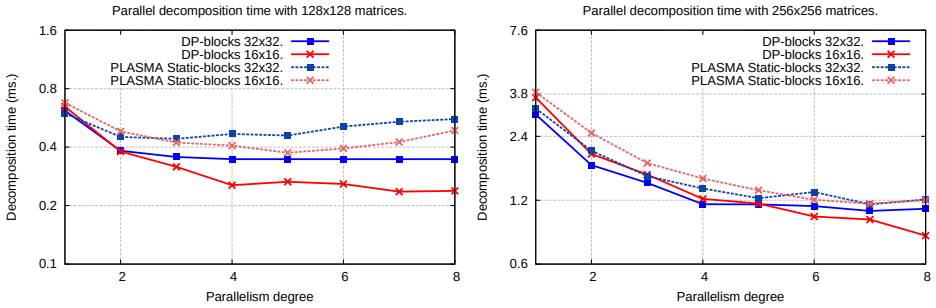
As for the task-farm pattern, our data-parallel implementation is highly portable on different architectures by relying on user-space synchronization mechanisms (i.e. by performing spin-loops on shared boolean flags) [59, 60]. Furthermore, the core functions invoked by data-parallel workers are `BLAS` and `LaPack` routines that can be linked to the most efficient implementation available for the underlying architecture, i.e. without modifying our source code during the porting phase.

In the following section we will describe the experimental results of this parallelization and we will compare it with the `PLASMA` versions.

6.2.1 Data-parallel experiments

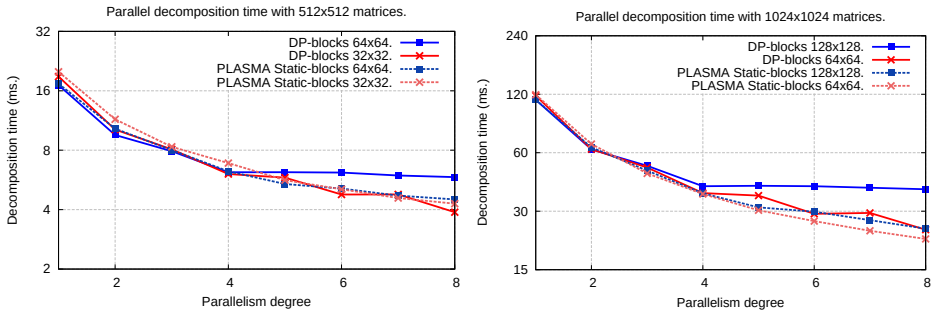
In this section we will present the data-parallel results on the Intel multi-processor (the other architectures are omitted for brevity, but they exhibit the same qualitative behavior). For the matrix sizes used in typical STAP applications, the `PLASMA` dynamic scheduling is not useful since it starts to be effective with larger matrices. Therefore, we will compare our implementation only with the `PLASMA` static scheduling.

Fig. 20 shows the parallel decomposition time using from 1 to 8 cores. As we can observe, the parallel efficiency stops being ideal very early, justifying the fact that the data-parallel parallelization can be used only with small parallelism degrees. We denote with "`DP-blocks 128x128`" our parallelization using blocks of 128×128



(a) $T_{fact}^{(p)}$ 128×128.

(b) $T_{fact}^{(p)}$ 256×256.



(c) $T_{fact}^{(p)}$ 512×512.

(d) $T_{fact}^{(p)}$ 1024×1024.

Figure 20.: Parallel decomposition times on the Intel multi-processor.

elements (and similarly for the other block sizes).

We can appreciate some important features of the data-parallel behavior. Firstly, the selection of the block size is crucial to extract sufficient parallelism, especially in our implementation since each core is assigned to a specific set of block rows. For fixed matrix, larger blocks lead to a smaller set of rows. With 128×128 matrices and 32×32 blocks only 4 block rows are available and parallelism degrees greater than 2 are completely useless. A higher number of cores can be used with smaller blocks: e.g. with blocks of 16×16 elements we double the available number of rows but we start from a slightly higher sequential decomposition time (from 0.461 ms to 0.530 ms). The same behavior can be observed with the other matrices. In conclusion the best results can be achieved using a small parallelism degree and the best block size for the given matrix size.

In terms of comparison between our implementation and the static `PLASMA` version, we can notice that our implementation provides better results with small parallelism degrees (2 and 4 cores). With small matrices (128×128 and 256×256) the performance improvement is evident also with greater parallelism degrees. As explained, this is due to a more simplified assignment of tasks to workers and a better exploitation of cache locality. On the other hand, the `PLASMA` scheduling starts to be more effective using larger matrices (Fig. 20d), since more powerful and complex scheduling strategies are able to extract more parallelism.

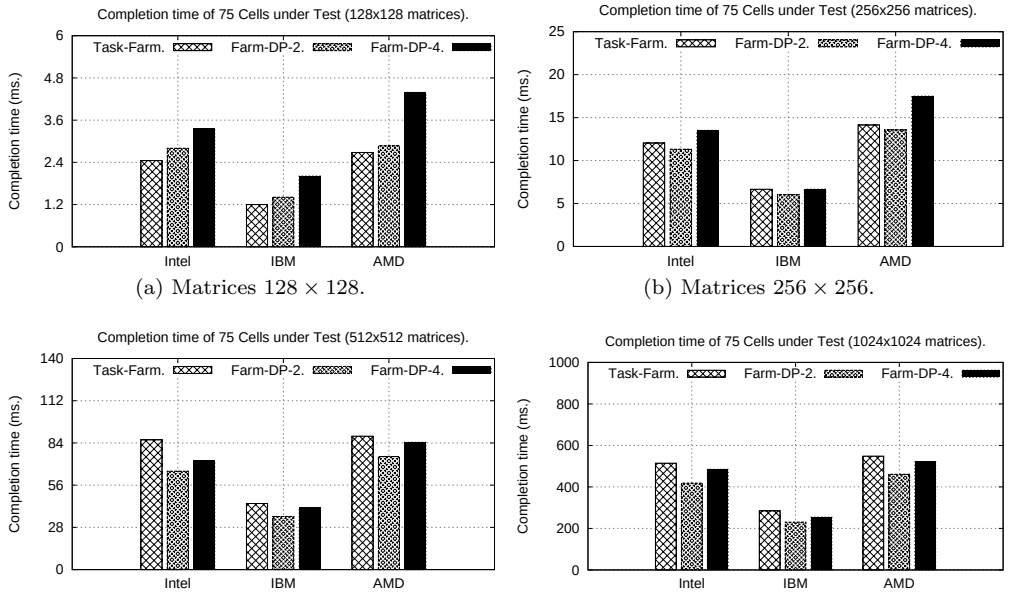
7. Performance evaluation of STAP benchmarks

In this section our goal is to analyze the completion time of the STAP computational kernel, i.e. the latency T_{weight} needed to complete the decomposition of a sequence of Γ covariance matrices (one for each cell under test) and calculate the corresponding weight vectors applying the forward and back substitutions. We compare three different parallelizations:

- **Pure Task-Farm**: the task-farm parallelization is configured to exploit the maximum number of cores of the underlying architecture. Each worker executes the fastest version of the tile Cholesky decomposition, according to the previously discussed results;
- **Nested Farm-DP-2**: a nested approach in which each task-farm worker is implemented by a data-parallel program using 2 cores. Thus, we have halved the task-farm parallelism degree (16 workers on the Intel and IBM multi-processors and 12 on the AMD multi-processor). The parallel Cholesky decomposition uses different block sizes in order to provide the best trade-off between the number of blocks and sequential calculation time. We use 32×32 blocks with matrices of size 128×128 and 256×256, 64×64 blocks with 512×512 matrices and 128×128 blocks with matrices of 1024×1024 elements;
- **Nested Farm-DP-4**: a nested approach in which each task-farm worker is internally parallelized using 4 cores. Again, we use the block size that provides the best parallel decomposition time with 4 cores. We use 16×16 and 32×32 blocks with matrices of size 128×128 and 256×256 elements. 64×64 blocks are used with larger matrices, of size 512×512 and 1024×1024.

As stated in Section 2, the number of cells under test depends on the radar characteristics and the number of desired targets to be tracked. We consider three different scenarios inspired by the examples described in [61]. In the first benchmark we consider $\Gamma = 75$ cells under test, in the other two benchmarks we consider a greater number equal to $\Gamma = 100$ and $\Gamma = 200$ cells. For each configuration (determined by the size of the covariance matrices and the number of cells under test) we discuss the results achieved by the three parallel versions.

Fig. 21 shows the results of 75 cells under test per CPI data-cube. In order to understand the rationale of our approach, we recall the concepts exposed in Section 6.1.4. On the IBM and Intel platforms, the integer ratio (rounded to the nearest upper integer) between the number of cells under test and the parallelism degree of the pure task-farm version is equal to $\lceil 75/32 \rceil = 3$. This means that some workers perform the computation on three cells under test. With the **Farm-DP-2** approach we have an integer ratio equal to 5. In this case, some task-farm workers receive five tasks, but each of them is solved faster since each worker is internally parallelized. We can notice that we have halved the task-farm parallelism degree but the number of tasks assigned to each worker is less than double that of the pure task-farm solution. If the data-parallel parallelization is sufficiently efficient, the result is an improvement in the computation latency T_{weight}.



(a) Matrices 128×128.

(b) Matrices 256×256.

(c) Matrices 512×512.

(d) Matrices 1024×1024.

Figure 21.: Completion time of 75 Cells under Tests.

Fig. 21a underlines that the nested approach is not satisfactory with 128×128 matrices. In this case the pure task-farm version always provides the best computation latency. To understand the reason, we can measure the *speed-up* of the

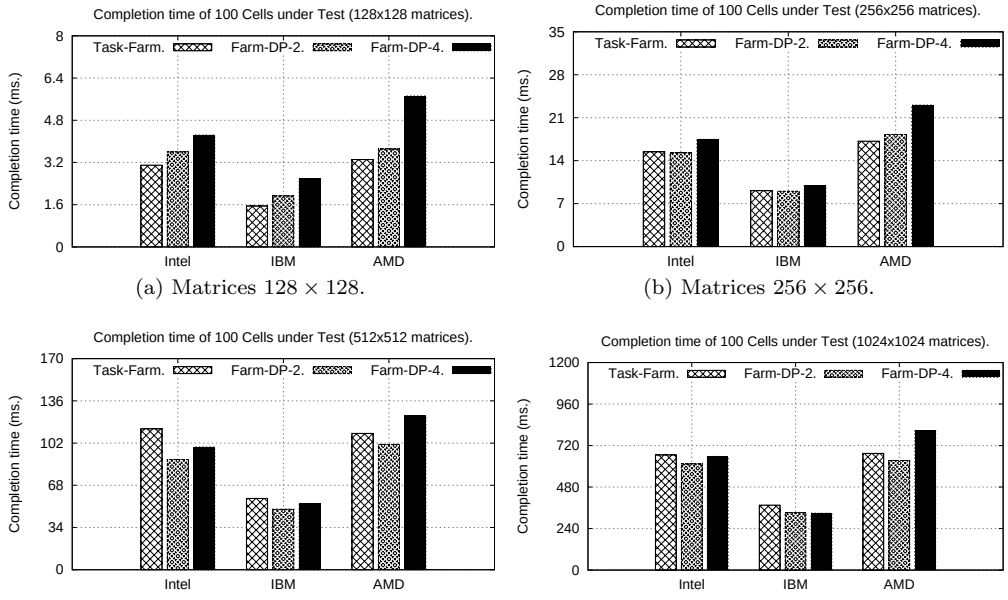
data-parallel version using p cores:

$$speed\text{-}up(p) = \frac{T_{fact}^{(best)}}{T_{fact}^{(p)}} \tag{12}$$

the term speedup is in this case justified, since we compare the parallel decomposition time with the best sequential algorithm (which can use a different block size). With 128×128 matrices, the speed-up is extremely low, equal to 1.15 on the Intel platform (and similarly on the other architectures). Under this condition, halving the number of task-farm workers results in a worse completion time.

With bigger matrices we experience a decisive change of course. The data-parallel parallelization obtains better speed-up: on the Intel architecture the speed-up using two cores is 1.35, 1.65 and 1.80 with 256×256, 512×512 and 1024×1024 matrices. As Figs. 21b, 21c and 21d highlight, the **Farm-DP-2** version achieves the best completion time. The average improvement (on the three architectures) compared with the pure task-farm version is of 6.56% with 256×256 matrices, 19.66% with 512×512 matrices and 17.92% with 1024×1024 matrices.

The **Farm-DP-4** version exhibits similar behavior. The speed-up of a parallel worker using 4 cores is extremely poor with small matrices (1.64 with matrices of 128×128 elements), and it gets better with larger matrices (we obtain speed-ups of 2.15, 2.55, 2.88 with 256×256, 512×512 and 1024×1024 matrices, respectively). This justifies the fact that the **Farm-DP-4** version becomes more effective than the pure task-farm solution using larger matrices. Nevertheless, the maximum speed-up is far from being optimal, and the **Farm-DP-2** version is the most effective combination to reach the best completion time.



(a) Matrices 128×128.

(b) Matrices 256×256.

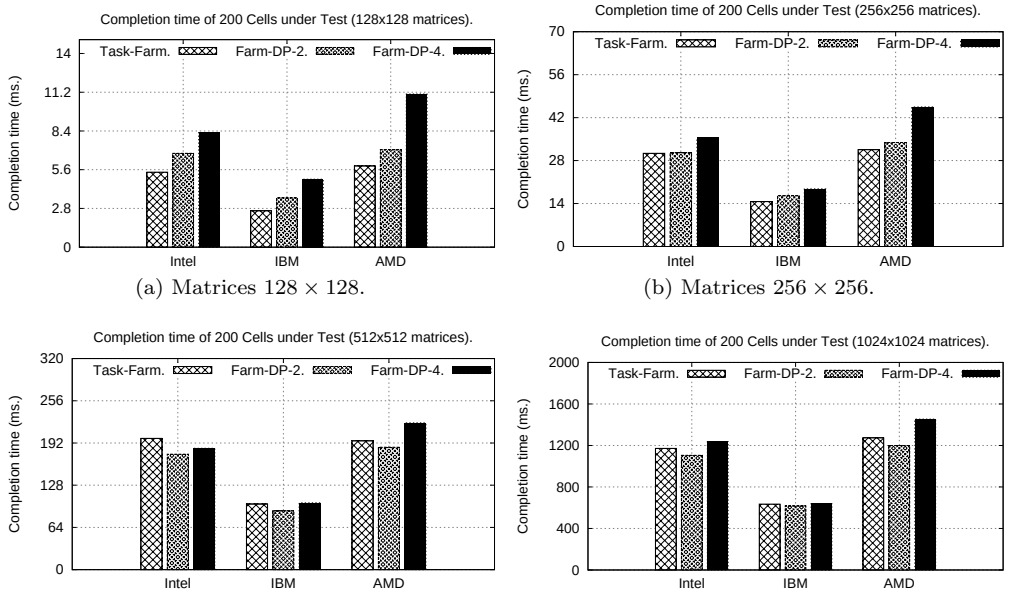
(c) Matrices 512×512.

(d) Matrices 1024×1024.

Figure 22.: Completion time of 100 Cells under Tests.

Fig. 22 shows the results with $\Gamma = 100$ cells under test. Qualitatively the behavior is similar to the previous benchmark. Due to a greater number of cells under test, the integer ratio between the task-farm parallelism degree and the number of cells under test is higher. Therefore, the contribution of the number of additional tasks received by some workers on the completion time is lower. The consequence is a smaller advantage of the nested versions w.r.t the task-farm solution. Similarly to the previous case, for 128×128 matrices the task-farm is still the best solution. This is also true for matrices of 256×256 elements, in which the nested versions perform similarly and in some cases worse than the task-farm scheme. With larger matrices, for which the data-parallel speed-up is better, the **Farm-DP-2** offers a non-negligible improvement. The average improvement on the three architectures is of 15% and 8.5% with matrices of 512×512 and 1024×1024 elements.

Finally, in the last benchmark (Fig. 23) we show the results of 200 cells under test. As for the previous benchmark, with 128×128 and 256×256 matrices the task-farm parallelization provides the best completion time. With larger matrices, the nested approach **Farm-DP-2** is still able to provide a slight advantage (of 9% and 4.86% with 512×512 and 1024×1024 matrices).



(a) Matrices 128×128.

(b) Matrices 256×256.

(c) Matrices 512×512.

(d) Matrices 1024×1024.

Figure 23.: Completion time of 200 Cells under Tests.

8. Conclusions

STAP is an effective and computationally expensive radar filtering technique. Due to its computational load, this algorithm is very difficult to be implemented in real-world apparatus. Important studies on special-purpose architectures such as FPGAs and GPUs have been conducted over the last years. However, a similar in-

vestigation on modern general-purpose architectures is not sufficiently undertaken. For this reason, this paper provided a detailed analysis of different parallelizations of the STAP computation kernel on modern multi-processors composed of several multi-core chips. We focused on the importance of the sequential algorithm to perform efficient and cache-aware covariance matrix decompositions. Tile algorithms deserved special attention: firstly they are able to exploit the potential of the hierarchical memory of modern architectures, and secondly they provide a decisive simplification to extract parallelism at the block level.

In this paper we investigated the application of parallelism paradigms on three modern architectures covering a broad range of existing multicores. The portability of our parallelizations, in terms of source code and achievable performance, was enabled by exploiting the `BLAS` and `LaPack` libraries and through a careful design of the run-time support of our parallel programs. For the task-farm scheme, we studied its throughput and scalability. The results evidenced a positive correlation between achieved performance and the adopted block size and the data-structure layout in main memory. For the data-parallel implementation we discussed the implementation of the `PLASMA` library. We experienced a poor scalability with the matrix sizes considered in this paper. For this reason we developed a customized data-parallel implementation able to provide better results in our test cases.

In the last part of the paper we evaluated our parallelizations using some STAP benchmarks, measuring their completion time. We presented some major problems of a pure task-farm approach when working with a small number of tasks, and we discussed the possibility of exploiting a data-parallel pattern to solve the problem. We proposed a mixed approach able to overcome the limitations of the task-farm approach without suffering from the limited data-parallel scalability. This was possible by exploiting a fundamental concept of Structured Parallel Programming, i.e. the composability of parallelism schemes. The nested approach provided a variable performance gain, reaching peaks of 20% compared with the pure task-farm parallelization.

In conclusion the paper provided a guideline to parallelize the STAP computational kernel on general-purpose multi-core architectures. We highlighted the benefits of using well-known parallelism paradigms, that allowed us to compare different implementations and compose them to leverage their strengths and weaknesses.

Acknowledgments

Many thanks to Dr. Achille Peternier and Prof. Walter Binder of the University of Lugano (USI), Switzerland, for granting us access to the IBM POWER7 server for conducting our experiments.

References

[1] J. Ward, *Space-time adaptive processing for airborne radar*, in *Space-Time Adaptive Processing (Ref. No. 1998/241), IEEE Colloquium on*, Apr., 1998,

pp. 2/1 –2/6.

[2] J.R. Guerci, *Space-Time Adaptive Processing for Radar*, Artech House, Norwood, MA, 2003.

[3] M. Wicks, M. Rangaswamy, R. Adve, and T. Hale, *Space-time adaptive processing: a knowledge-based perspective for airborne radar*, Signal Processing Magazine, IEEE 23 (2006), pp. 51 – 65.

[4] R. Klemm, *Applications of Space-Time Adaptive Processing*, IEEE Publishing, USA, 2004.

[5] J. Lebak and A. Bojanczyk, *Design and performance evaluation of a portable parallel library for space-time adaptive processing*, Parallel and Distributed Systems, IEEE Transactions on 11 (2000), pp. 287 –298.

[6] K. Hwang and Z. Xu, *Scalable parallel computers for real-time signal processing*, Signal Processing Magazine, IEEE 13 (1996), pp. 50 –66.

[7] C. DeLuca, C.W. Heisey, R.A. Bond, and J.M. Daly, *A Portable, Object-Based Parallel Library and Layered Framework for Real-Time Radar Signal Processing*, in *Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, ISCOPE '97, Springer-Verlag, London, UK, 1997, pp. 241–248.

[8] D. Weiner, *Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers*, in *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, IPPS '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 220–238.

[9] M. Lee, , M. Lee, and V.K. Prasanna, *High Throughput-Rate Parallel Algorithms for Space Time Adaptive Processing*, in *Proc. 2nd International Workshop on Embedded HPC Systems and Applications (EHPC '97)*, 1997.

[10] D. Wu, Y.H. Li, J. Eilert, and D. Liu, *Real-Time Space-Time Adaptive Processing on the STI CELL Multiprocessor*, in *Radar Conference, 2007. EuRAD 2007. European*, October, 2007, pp. 71 –74.

[11] M. Aldinucci, L. Anardu, M. Danelutto, M. Torquati, and P. Kilpatrick, *Parallel patterns + Macro Data Flow for multi-core programming*, in *Proc. of Intl. Euromicro PDP 2012: Parallel Distributed and network-based Processing*, Feb., IEEE, Garching, Germany, 2012, pp. 27–36.

[12] G. Mencagli and M. Vanneschi, *QoS-control of Structured Parallel Computations: A Predictive Control Approach*, in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, 2011, pp. 296–303.

[13] G. Mencagli, M. Vanneschi, and E. Vespa, *Reconfiguration stability of adaptive distributed parallel applications through a cooperative predictive control approach*, in *Proceedings of the 19th international conference on Parallel Processing*, Euro-Par'13, Aachen, Germany, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 329–340.

[14] I. Reed, J. Mallett, and L. Brennan, *Rapid convergence rate in adaptive arrays*, Aerospace and Electronic Systems, IEEE Transactions on AES-10 (1974), pp. 853 –863.

[15] S. De Greve, F. Lapierre, and J. Verly, *Canonical framework for describing suboptimum radar space-time adaptive processing (STAP) techniques*, in *Radar Conference, 2004. Proceedings of the IEEE*, April, 2004, pp. 474 – 479.

[16] J.O. McMahon, *Space-Time Adaptive Processing on the Mesh Synchronous Processor*, in *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, 1996.

[17] M. Cole, *Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming*, Parallel Comput. 30 (2004), pp. 389–406.

[18] H. González-Vélez and M. Leyton, *A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers*, Software–Practice & Experience 40 (2010), pp. 1135–1160.

[19] G. Yaikhom, M. Cole, S. Gilmore, and J. Hillston, *A structural approach for modelling performance of systems using skeletons*, Electronic Notes in Theoretical Computer Science 190 (2007), pp. 167 – 183, Proceedings of the Fifth Workshop on Quantitative Aspects of Programming Languages (QAPL 2007).

[20] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi, *A heterogeneous environment for hpc applications*, Parallel Computing 25 (1999), pp. 1827–1852.

[21] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, *P3l: A structured high-level parallel language, and its structured support*, Concurrency: Practice and Experience 7 (1995), pp. 225–255.

[22] J. Darlington, Y.k. Guo, H. To, and J. Yang, *Functional skeletons for parallel coordination*, in *EURO-PAR '95 Parallel Processing*, S. Haridi, K. Ali, and P. Magnusson, eds., Lecture Notes in Computer Science, Vol. 966, Springer Berlin Heidelberg, 1995, pp. 55–66.

[23] M. Vanneschi, *The programming model of assist, an environment for parallel and distributed portable applications*, Parallel Comput. 28 (2002), pp. 1709–1732.

[24] D. Caromel and M. Leyton, *Fine tuning algorithmic skeletons*, in *Euro-Par 2007 Parallel Processing*, A.M. Kermarrec, L. Bouge, and T. Priol, eds., Lecture Notes in Computer Science, Vol. 4641, Springer Berlin Heidelberg, 2007, pp. 72–81.

[25] M. Aldinucci, M. Danelutto, and P. Teti, *An advanced environment supporting structured parallel programming in java*, Future Generation Computer Systems 19 (2003), pp. 611 – 626, ¡ce:title¿Tools for Program Development and Analysis. Best papers from two Technical Sessions, at ICCS2001, San Francisco, CA, USA, and ICCS2002, Amsterdam, The Netherlands¡/ce:title¿.

[26] J.F. Ferreira, J. Sobral, and A. Proenca, *JaSkel: a Java skeleton-based framework for structured cluster and grid computing*, in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, Vol. 1, 2006, pp. 4 pp.–304.

[27] M. Leyton and J. Piquer, *Skandium: Multi-core Programming with Algorithmic Skeletons*, in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 2010, pp. 289–296.

[28] M. Dieterle, J. Berthold, and R. Loogen, *A skeleton for distributed work pools in eden*, in *Proceedings of the 10th international conference on Functional and Logic Programming*, FLOPS'10, Sendai, Japan, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 337–353.

[29] H. Kuchen and S. Ernsting, *Data parallel skeletons in java*, Procedia Computer Science 9 (2012), pp. 1817 – 1826, ¡ce:title¿Proceedings of the International

Conference on Computational Science, {ICCS} 2012¡/ce:title¿.

[30] M. Danelutto and M. Stigliani, *SKElib: Parallel Programming with Skeletons in C*, in *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, Springer-Verlag, London, UK, UK, 2000, pp. 1175–1184.

[31] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, *Accelerating code on multi-cores with fastflow*, in *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, Bordeaux, France, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 170–181.

[32] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, *A library of constructive skeletons for sequential style of parallel programming*, in *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, Hong Kong, ACM, New York, NY, USA, 2006.

[33] J.J. Dongarra, P. Luszczek, and A. Petitet, *The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience*, Concurrency and Computation: Practice and Experience 15 (2003), p. 2003.

[34] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, *Lapack: a portable linear algebra library for high-performance computers*, (1990), pp. 2–11.

[35] Intel, *Math kernel library* .

[36] T.A.C. Center, *Goto blas library 2* .

[37] R.C. Whaley and J.J. Dongarra, *Automatically tuned linear algebra software*, in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, San Jose, CA, IEEE Computer Society, Washington, DC, USA, 1998, pp. 1–27.

[38] T.A.C. Center, *Plasma users guide, parallel linear algebra software for multicore architectures, version 2.3* (2012).

[39] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Comput. 35 (2009), pp. 38–53.

[40] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra, *Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures*, Concurr. Comput. : Pract. Exper. 24 (2011), pp. 305–321.

[41] J. Kurzak, H. Ltaief, J. Dongarra, and R.M. Badia, *Scheduling dense linear algebra operations on multicore processors*, Concurreny and Computation: Practice and Experience (2009), pp. 15–44.

[42] S. Tomov, J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid gpu accelerated manycore systems*, Parallel Comput. 36 (2010), pp. 232–240.

[43] L.S. Blackford, J. Choi, A. Cleary, A. Petitet, R.C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker, *ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance*, in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Pittsburgh, Pennsylvania, United States, IEEE Computer Society, Washington, DC, USA, 1996.

[44] M. Mccool, *Signal processing and general-purpose computing and gpus [exploratory dsp]*, Signal Processing Magazine, IEEE 24 (2007), pp. 109 –114.

[45] D. Yang, G.D. Peterson, and H. Li, *Compressed sensing and cholesky decomposition on fpgas and gpus*, Parallel Comput. 38 (2012), pp. 421–437.

[46] I. Reed, J. Mallett, and L. Brennan, *Rapid convergence rate in adaptive arrays*, Aerospace and Electronic Systems, IEEE Transactions on AES-10 (1974), pp. 853 –863.

[47] D.W. Barron and H.P.F. Swinnerton-Dyer, *Solution of simultaneous linear equations using a magnetic-tape store*, The Computer Journal 3 (1960), pp. 28–33.

[48] O.E. Brnlund and T.L. Johnsen, *Qr-factorization of partitioned matrices : Solution of large systems of linear equations with non-definite coefficient matrices*, Computer Methods in Applied Mechanics and Engineering 3 (1974), pp. 153 – 172.

[49] A.H. Sameh, *Parallel algorithms on the CEDAR system*, in *Proc. of the conference on algorithms and hardware for parallel processing on CONPAR 86*, Aachen, Germany, Springer-Verlag New York, Inc., New York, NY, USA, 1986, pp. 25–39.

[50] C. Bischof and C.v. Loan, *The WY representation for products of householder matrices*, in *Selected Papers from the Second Conference on Parallel Processing for Scientific Computing*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1987, pp. 2–13.

[51] J. Shirako, K. Sharma, N. Fauzia, L.N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, *Analytical bounds for optimal tile size selection*, in *Proceedings of the 21st international conference on Compiler Construction*, CC'12, Tallinn, Estonia, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 101–121.

[52] E. Park, S. Kulkarni, and J. Cavazos, *An evaluation of different modeling techniques for iterative compilation*, in *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, Taipei, Taiwan, ACM, New York, NY, USA, 2011, pp. 65–74.

[53] D. Gibson and D.A. Wood, *Forwardflow: a scalable core for power-constrained CMPs*, in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, Saint-Malo, France, ACM, New York, NY, USA, 2010, pp. 14–25.

[54] M. Kudlur and S. Mahlke, *Orchestrating the execution of stream programs on multicore platforms*, SIGPLAN Not. 43 (2008), pp. 114–124.

[55] W.J. Bolosky, M.L. Scott, R.P. Fitzgerald, R.J. Fowler, and A.L. Cox, *Numa policies and their relation to memory architecture*, SIGPLAN Not. 26 (1991), pp. 212–221.

[56] R.P. Larowe Jr. and C. Schlatter Ellis, *Experimental comparison of memory management policies for numa multiprocessors*, ACM Trans. Comput. Syst. 9 (1991), pp. 319–363.

[57] L.S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, *ScaLAPACK user's guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[58] J. Kurzak, A. Buttari, and J. Dongarra, *Solving systems of linear equations on the cell processor using cholesky factorization*, IEEE Trans. Parallel Distrib. Syst. 19 (2008), pp. 1175–1186.

[59] T. De Matteis, F. Luporini, G. Mencagli, and M. Vanneschi, *Evaluation of Architectural Supports for Fine-Grained Synchronization Mechanisms*, in *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, 2013.

[60] D. Buono, M. Danelutto, S. Lametti, and M. Torquati, *Parallel Patterns for General Purpose Many-Core*, in *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 131–139.

[61] K. Cain, J. Torres, and R. Williams, *Rt-stap: Real-time space-time adaptive processing benchmark*, Tech. rep., MITRE Corporation, 1997.