IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 13, 2018, accepted February 7, 2019, date of publication February 19, 2019, date of current version March 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2900122

A Fast Solver for Large Tridiagonal Systems on Multi-Core Processors (Lass Library)

PEDRO VALERO-LARA[1], DIEGO ANDRADE[2], RAÜL SIRVENT[1], JESÚS LABARTA[3], BASILIO B. FRAGUELA[2], AND RAMÓN DOALLO[2]

[1]Barcelona Supercomuting Center, 08034 Barcelona, Spain
[2]Department of Computer Engineering, Universidade da Coruña, 15001 A Coruña, Spain
[3]Department of Computer Architecture, Politècnica de Catalunya, 08034 Barcelona, Spain

Corresponding author: Diego Andrade (diego.andrade@udc.es)

This work was supported in part by the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Human Brain Project SGA1 and Human Brain Project SGA2 under Grant 720270 and Grant 785907, in part by the Spanish Ministry of Economy and Competitiveness under the Project Computación de Altas Prestaciones VII under Grant TIN2015-65316-P, in part by the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Paralůlels under Grant 2014-SGR-1051, in part by the Juan de la Cierva under Grant IJCI-2017-33511, in part by the Fujitsu under the Barcelona Supercomputing Center-Fujitsu Joint Project: Math Libraries Migration and Optimization, in part by the Ministerio de Economía, Industria y Competitividad of Spain, in part by the Fondo Europeo de Desarrollo Regional Funds of the European Union under Grant TIN2016-75845-P, and in part by the Xunta de Galicia co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups under Grant ED431C 2017/04, and in part by the Centro Singular de Investigación de Galicia accreditatión 2016-2019 under Grant ED431G/01.

ABSTRACT Many problems of industrial and scientific interest require the solving of tridiagonal linear systems. This paper presents several implementations for the parallel solving of large tridiagonal systems on multi-core architectures, using the OmpSs programming model. The strategy used for the parallelization is based on the combination of two different existing algorithms, PCR and Thomas. The Thomas algorithm, which cannot be parallelized, requires the fewest number of floating point operations. The PCR algorithm is the most popular parallel method, but it is more computationally expensive than Thomas. The method proposed in this paper starts applying the PCR algorithm to break down one large tridiagonal system into a set of smaller and independent ones. In a second step, these independent systems are concurrently solved using Thomas. The paper also contains an analytical study of which is the best point to switch from PCR to Thomas. Also, the paper addresses the main performance issues of combining PCR and Thomas proposing a set of alternative implementations, some of them even imply algorithmic changes. The performance evaluation shows that the best implementation achieves a peak speedup of 4 with respect to the Intel MKL counterpart routine and 2.5 with respect to a single-threaded Thomas.

INDEX TERMS Tridiagonal solve, multi-core, auto-tuning, OmpSs.

I. INTRODUCTION

The resolution of tridiagonal linear systems is required in many problems of industrial and scientific interest. Examples are: alternating direction implicit methods [1], Poisson solvers [2], [3], cubic spline approximations [4], numerical ocean models [5], preconditioners for iterative linear solvers [6] or the simulation of the human brain [7], [8]. Usually, the solving of tridiagonal systems takes most of the computation time of these applications.

Most of the references of the related work (see Section VII) are focused on solving multiple tridiagonal systems in parallel. However, a major problem not addressed in previous works is the efficient resolution of large tridiagonal systems on multi-core architectures. This work focuses on this latter problem for several reasons [9]: i) A set of independent systems can always be expressed as a single large system by joining matrices into a large one. ii) A fast parallel solver for a single system is the most generally applicable, being even able to deal with irregularly sized systems with no additional complexity. iii) A single system is the most difficult case to parallelize, because the problem has no inherent independence to exploit from disjoint systems. iv) To minimize the memory transfer overhead, in those applications which involve large enough systems and require partitioning. The amount of communication and transfer overhead will grow with the number of times each system is partitioned. We should then prefer to send large integrated systems for

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

minimal transfer overhead by facilitating parallel executions, which does not need external communication.

Although the use of pivoting for the solving of linear systems of equations is commonly accepted, we can find multiples problems where the matrices to be solved are well-conditioned, and so expensive computationally operations like pivoting are not necessary. Due to this, it is possible to find multiples implementations in reference libraries, which do not make use of such technique. Examples of this are: MAGMA library,[1] Intel MKL,[2] NVIDIA cuSolver,[3] NVIDIA cuSparse,[4] just to mention a few. Regarding the solving of tridiagonal systems, it is also easy to find multiples implementations, which do not make use of pivoting. In the cuSparse library, the reference library for sparse operations on NVIDIA GPUs, we find five different routines, which implement no-pivoting algorithms (*g*tsv_nopivot, *g*tsv2_nopivot, *g*tsvStridedBatch, *g*tsv2StridedBatch, and *g*tsvInterleavedBatch). Also, the BLKTRI routine of the open-source FISHPACK package,[5] makes use of no-pivoting Thomas algorithm to compute tridiagonal systems [2], [3]. Other examples of implementations of no-pivoting solvers for the computation of tridiagonal systems are introduced in Section VII.

In this work, the authors propose a novel implementation based on the use of two **no-pivoting** existing methods, Parallel Cyclic Reduction (PCR) and Thomas. The Thomas algorithm is the optimal one in terms of number of floating point operations, but it cannot be parallelized. The PCR one requires many more floating point operations but it can be aggressively parallelized. In fact, in many-core architectures, the parallel execution can compensate the high computational cost. However, the differences found among both architectures, many-core (high-throughput oriented architectures) and multi-core (low-latency oriented architectures), make this last unsuitable to execute PCR. Still, in the context of multi-core architectures, PCR can be used to break down a large problem into a set of smaller independent ones. These problems can then be solved using Thomas. As the systems generated by PCR are independent, they can be solved in parallel using different cores.

We make use of the open-source OmpSs programing model [10] instead of others for the following reasons: i) This model presents an efficient management of the threads based on the use of queues, without the need of dealing with the overhead found in others models, such as the fork-join model used in OpenMP. ii) It allows us to have a deeper control and analysis to evaluate the threads scheduling and the potential benefits of the optimizations proposed. iii) OmpSs is specially well integrated with the tools used for the performance evaluation Extrae and Paraver.

This work contains several implementations of the PCR+Thomas (P+T) algorithm that try to address several problems. One of them is the poor data locality. The reasons for this are that: the PCR algorithm generates strided memory accesses, and the switch from PCR to Thomas (and viceversa) requires data movements between strided memory positions and consecutive ones. Another problem is that in multi-core platforms, the parallel execution of PCR does not compensate its higher computational cost. As a consequence, we must pay a special attention at the PCR part of the algorithm.

One of the key challenges in making the aforementioned strategy efficient consists in selecting the appropriate switch point (SP) between both algorithms, PCR and Thomas. The paper describes a general strategy that allows to select automatically the best SP for a given platform and input size. Unlike the time-consuming autotuning-based approaches [11], [12], the proposed strategy is purely analytical and it requires just a few short calculations. This analysis reveals that under certain conditions, the P+T algorithm achieves the best performance when PCR generates less independent systems than cores. This means that during the Thomas part of the algorithm, the processor is underutilized, but it also implies an important reduction of the power consumption.

In summary, the contribution of this work is two-fold. First, we develop different variants of a multi-stage and hybrid tridiagonal solving which is able to efficiently compute large tridiagonal systems on current multi-core architectures. Our second contribution consists of a self-tuning implementation that attempts not only to reduce the execution time but also to minimize the use of computational resources, achieving an energy-efficient implementation. The work described in this paper is part of a novel open-source library for linear algebra operations called LASs (Linear Algebra routines on OmpSs).[6]

The rest of this paper is organized as follows. Section II briefly introduces the problem at hand and the different methodologies to deal with it. Section III explains the P+T algorithm and describes the analytical method proposed to select the best SP. Section IV explains in detail the alternative implementations of the P+T algorithm proposed in this paper, as well as the optimizations explored and the memory space occupied by each of the implementations involved in this paper. A study about the stability of the proposed methodology is performed in Section V. Section VI shows the performance of these implementations and analyzes their behavior using their respective traces. Section VII discusses the related work and Section VIII concludes.

II. TRIDIAGONAL SYSTEMS

The best method to solve tridiagonal systems sequentially is the Thomas algorithm [2], [3]. This method is a specialized

[1] http://icl.cs.utk.edu/projectsfiles/magma/doxygen/group_group_gesv_nopiv.html
[2] https://software.intel.com/en-us/mkl-developer-reference-c-mkl-getrfnpi
[3] https://docs.nvidia.com/cuda/cusolver/index.html
[4] https://docs.nvidia.com/cuda/cusparse/
[5] https://www.netlib.org/fishpack/
[6] https://pm.bsc.es/mathlibs/lass

application of the Gaussian elimination that takes advantage of the the tridiagonal structure of the system. It consists of two stages, commonly denoted as forward elimination and backward substitution.

The algorithm solves a linear $Au = y$ system, where A is a tridiagonal matrix:

$$A = \begin{bmatrix} b_1 & c_1 & & & & 0 \\ a_2 & b_2 & c_2 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}$$

Let us notice that the data structures required by this algorithm are: three arrays (a, b and c) of size n representing the three diagonals of the input matrix; and two additional vectors of the same size, u and y, that store the unknowns of the equation (to be calculated) and the right hand terms of the equation, respectively. The implementation of this algorithm do not really require the u array, as the result is overwritten in array y, we have included u for the sake of clarity. The algorithm starts with the forward stage that eliminates the lower diagonal as follows:

$$c'_1 = \frac{c_1}{b_1}, \quad c'_i = \frac{c_i}{b_i - c'_{i-1}a_i} \quad \text{for} \quad i = 2, 3, \ldots, n-1$$

$$y'_1 = \frac{y_1}{b_1}, \quad y'_i = \frac{y_i - y'_{i-1}a_i}{b_i - c'_{i-1}a_i} \quad \text{for} \quad i = 2, 3, \ldots, n-1$$

and then the backward stage recursively solves each row in reverse order:

$$u_n = y'_n, \quad u_i = y'_i - c'_i u_{i+1} \quad \text{for} \quad i = n-1, n-2, \ldots, 1$$

Overall, the algorithm requires $8n$ operations in $2n - 1$ steps, its main limitation being that it is not possible to parallelize it.

Cyclic Reduction (CR) [2], [3], [13], [14] is a parallel alternative to Thomas algorithm. It also consists of two phases (reduction and substitution). In each intermediate step of the reduction phase, all even-indexed (i) equations $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ are reduced. The values of a_i, b_i, c_i and d_i are updated in each step according to:

$$\begin{aligned} a'_i &= -a_{i-1}k_1, \quad & b'_i &= b_i - c_{i-1}k_1 - a_{i+1}k_2 \\ c'_i &= -c_{i+1}k_2, \quad & y'_i &= y_i - y_{i-1}k_1 - y_{i+1}k_2 \\ k_1 &= \frac{a_i}{b_{i-1}}, & k_2 &= \frac{c_i}{b_{i+1}} \end{aligned}$$

After $\log_2 n$ steps, the system is reduced to a single equation that is solved directly. All odd-indexed unknowns x_i are then solved in the substitution phase by introducing the already computed u_{i-1} and u_{i+1} values:

$$u_i = \frac{y'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}$$

Overall, the CR algorithm needs $17n$ operations and $2\log_2 n - 1$ steps. Figure 1-left graphically illustrates its access pattern.

The most popular parallelizable method to solve tridiagonal system is the Parallel Cyclic Reduction (PCR) [2], [3], [13], [14] algorithm, which only has a substitution phase. For convenience, we consider cases where $n = 2^s$, that involve $s = \log_2 n$ steps. This assumption is common to most implementations, and it can easily overcome using padding and other well-known techniques. The coefficients a, b, c and y are updated as follows, for $j = 1, 2, \ldots, s$ and $k = 2^{j-1}$:

$$\begin{aligned} \alpha_i &= -a_i/b_{i-k}, \quad & \beta_i &= -c_i/b_{i+k} \\ a'_i &= \alpha_i a_{i-k}, & b'_i &= b_i + \alpha_i c_{i-k} + \beta_i a_{i+k} \\ c'_i &= \beta_i c_{i+k}, & y'_i &= y_i + \alpha_i y_{i-k} + \beta_i y_{i+k} \end{aligned}$$

finally the solution is achieved as:

$$u_i = \frac{y'_i}{b'_i}$$

Essentially, at each reduction stage, the current system is transformed into two smaller systems and after $\log_2 n$ steps the original system is reduced to n independent equations of 1 element, which can be solved trivially. Overall, the operation count of PCR is $12n \log_2 n$, which is much larger that the operation count of Thomas. The main advantage of PCR over Thomas is that it can be parallelized. Actually, its massively parallel execution in many-core platforms compensate its greater operation count. A very important problem of this algorithm is the poor locality of its access pattern, which is sketched in Figure 1-right. The first step of the algorithm uses stride one, but in each step the stride is multiplied by 2, which affects negatively to the cache performance. This is one of the issues that we have to address in this work, as PCR is one of the building blocks of our proposal.

There are more algorithms apart from the ones mentioned above to solve tridiagonal systems, such as those based on Recursive Doubling [13], among others. Although, we also explore the use of *CR*, since it is one of the state-of-the-art methodologies for the parallel solving of tridiagonal systems, we mainly focus on PCR and Thomas, as a partial execution of PCR allows to break down one big problem into smaller ones, which can be solved concurrently in different cores, and Thomas is the most efficient algorithm to be used within a core.

In fact, hybrid combinations that try to exploit the best of each algorithm have been explored [2], [3], [13]–[16] by combining PCR + CR. However, as it is shown in this paper and unlike the Thomas algorithm, the CR algorithm is in need of many more arithmetic and memory operations, being not as efficient within a single core as the Thomas method.

III. THE PCR + THOMAS METHOD

The method proposed in this paper, exemplified in Figure 2, consists of combining the PCR and the Thomas algorithms. We first process a prefixed number of PCR steps (or levels). We will call this number the switch point (SP) from PCR to Thomas. PCR is using here as a mean to break down a

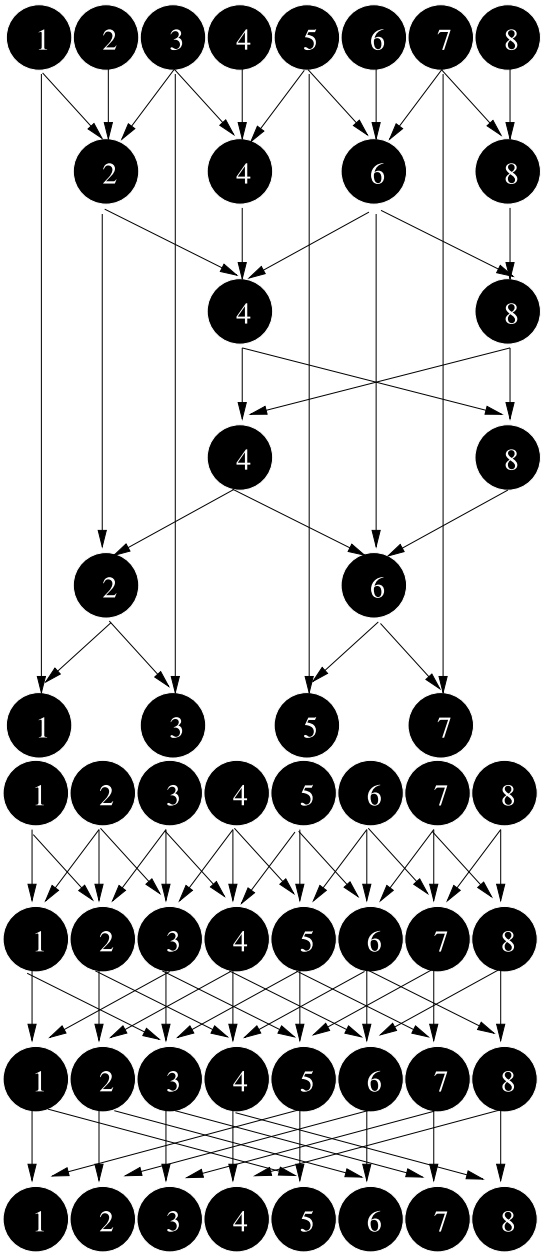

FIGURE 1. Access pattern of the CR (left) and the PCR (right) algorithm.

large problem into smaller ones. Formally, at the end of each PCR step i, the algorithm generates 2^i systems of $\frac{n}{2^i}$ equations each. Each system s is defined by the elements of the a, b, c and y arrays starting at position s and with a stride of 2^i, that is, $[s + j \times 2^i]$, $\forall j = 0 \ldots \frac{n}{2^i}$. In the example of Figure 2, the size of the diagonal $n = 8$, the SP is 2, and after that step of PCR, the algorithm has generated 4 systems (represented in black, grey,green and red colors, respectively) of 2 elements separated by a stride of 4.

The systems generated by PCR are fully independent and although the Thomas algorithm is not parallel, it can be applied concurrently, without any synchronization, to solve each one of these systems. But before the independent

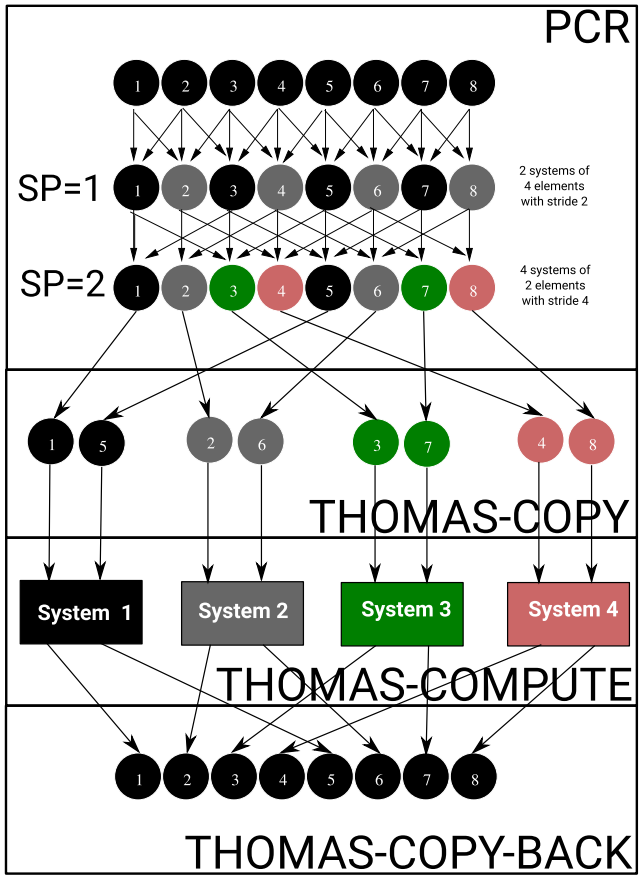

FIGURE 2. Example of the PCR+Thomas algorithm with $n = 8$ and $SP = 2$.

systems are concurrently solved using Thomas, the positions of all the elements of a system has to be placed in consecutive positions of 4 extra buffers, one per array. Once each system is solved using Thomas, the results are copied back (overwritten) to its corresponding positions in the y array. Let us recall, that the result is really overwritten in this array.

In summary, our PCR+Thomas method for multi-core CPUs is composed of the following stages (shown in the example of Figure 2):

1) **PCR**: SP steps of PCR are done
2) **Thomas-Copy**: The information of each system, which is in strided positions, is placed in consecutive memory positions of buffers, as a preparation to be processed by the Thomas algorithm.
3) **Thomas-Compute**: Each system is solved using Thomas. All the systems can be solved concurrently.
4) **Thomas-Copy-Back**: The results of each system, which are in consecutive positions of the buffers, are copied back to the appropriate strided positions of the y array, which is overwritten with the result.

A. SWITCH POINT

The idea of combining PCR and Thomas has been already explored in [16] for GPU-based architectures without any benefits with respect to the sequential CPU code when

solving one large tridiagonal system. The implementation introduced in [16] is divided into four stages. In the first three ones the problem is increasingly subdivided using PCR, until the sub-systems created by PCR fit in the GPU shared memory. In the fourth stage, each thread solves sequentially a system using Thomas. The authors of this implementation prove that the selection of the switch points between the different stages of this method is key for its performance, and they propose several auto-tuning strategies to do it appropriately. These strategies first split systems until there is enough parallel work to do and the it keeps subdividing the problem in a subsequent phase until each sub-system can fit the shared memory of the GPU. It is important to note that although the pure PCR algorithm is specially suitable for GPU-based architectures, where a massive parallel execution can compensate its larger operation count with respect to the Thomas algorithm, the differences among both architectures, many-core GPUs and multi-core CPUs, implies that the PCR computing must be much shorter in CPUs.

As a consequence, one of the key challenges in the implementation of our hybrid algorithm is to select the appropriate switch point. A preliminary idea would be to use PCR to break down the input problem until we have generated at least one system per core. But, a deeper study of the particular characteristics of our problem reveals that it is not the best strategy.

The PCR algorithm requires $12 \times n$ operations in each level, n being the size of the problem to be computed. The Thomas algorithm requires $8 \times n$ operations to solve the whole system. Thus, we need less operations to solve the system using Thomas than to process just one PCR level. This means that PCR is only worth when it is executed in parallel, because its complexity is going to be divided by the number of cores available (for instance, 48 in our target platform). Each PCR level doubles the number of independent tridiagonal systems generated, which is initially 1, and the size of each system is divided by 2. This means that we have 2^i independent tridiagonal systems of size $n/2^i$ after computing the i^{th} PCR level. To compute the number of operations of our hybrid method depending on the switch point, we follow the next equation:

$$((12 \times n)/\#cores) \times SP + (8 \times (n/2^{SP}))$$

where SP is the switch point, i.e., the number of PCR levels computed. $((12 \times n)/\#cores) \times SP$ being the cost of the PCR part, and $(8 \times (n/2^{SP}))$ the cost of the Thomas one.

Figure 3 graphically illustrates the total number of operations depending on the size of the tridiagonal system and the switch point for our test platforms (48 cores). As shown, the total number of operations decreases until the 4^{th} PCR level independently of the system size. From this switch point (PCR level) the number of operations is the same (in the 5^{th} level) or bigger (from the 6^{th} PCR level). Notice that in order to compute the number of operations for the case of computing 6 PCR levels, it is necessary to multiply the right side of the equation (Thomas operations) by 2. The reason is

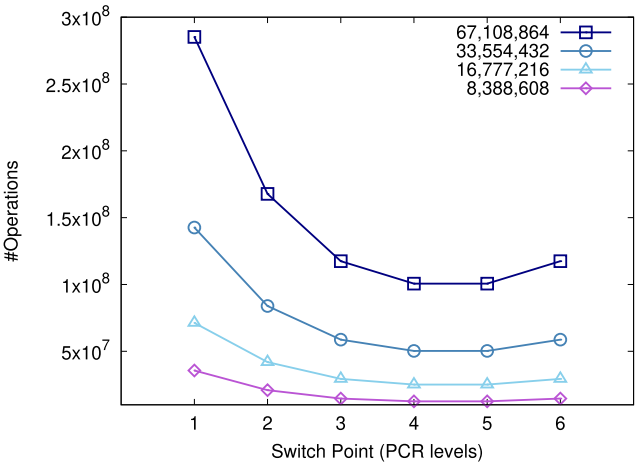

FIGURE 3. PCR + Thomas algorithm.

that some cores have to process two systems using Thomas. Specifically, 64 independent tridiagonal systems are created after computing 6 PCR levels and our test platform has only 48 cores. Given this analysis, the peak performance should be achieved by computing 3 or 4 PCR levels, using all the cores available for computing the PCR step, and 8 (2^3) or 16 (2^4) cores (switch point) for computing the Thomas step. This analysis can be used to select the appropriate switch point in a self-tuning implementation. Its effectiveness will be evaluated in Section VI.

For the sake of clarity and help to understand our work, we include in Algorithm 1 a simple pseudocode of the implementation above described.

IV. IMPLEMENTATION OF THE PCR+THOMAS METHOD

In PCR, every equation is independent with respect to the rest, and so each of them can be totally computed in parallel. However, the computation must be synchronized in every level, as there is a data-dependency between consecutive PCR levels. The baseline parallel implementation of the PCR+Thomas algorithm uses an OmpSs *loop pragma* directive in the inner loop of PCR. This OmpSs compiler directive is similar to the OpenMP parallel loop pragma, and it is used to distribute the iterations of a loop among several threads. In the Thomas stages, a loop is needed to iterate on the systems generated in the PCR stage. This loop goes through the three last stages: **Copy**, **Compute** and **Copy-Back**. The baseline implementation uses also a *loop pragma* to parallelize this loop. We call this variant the PCR+Thomas (P+T) implementation. In the following sections, Sections IV-A and IV-B, we describe the strategies proposed to improve performance on the mayor stages.

A. IMPROVING THOMAS

The main hotspot among the Thomas stages is the **Thomas-Copy** stage, where the elements of the input arrays corresponding to each generated system are copied to consecutive memory positions of several buffers, one per system, in order

Algorithm 1 PCR-Thomas pseudocode.

```
1: //PCR
2: //SWITCH_POINT has been previously computed as
   described in Section III
3: for i = 0 → SWITCH_POINT do
4:    //N is the size of the tridiagonal system
5:    #pragma oss loop
6:    for j = 0 → N do
7:       ComputePCR_Level(i)
8:    end for
9:    #pragma oss taskwait
10: end for
11: //Thomas-Copy
12: #pragma oss loop
13: for j = 0 → N do
14:    PCR_Outputs → Thomas_Inputs
15: end for
16: #pragma oss taskwait
17: //Thomas-Compute & Thomas-Copy-Back
18: //#Thomas is the independent number of tridiagonal
   systems uncoupled in the PCR-step
19: #pragma oss loop
20: for j = 0 → #Thomas do
21:    ComputeThomas(j)
22:    Thomas_Output → RHS
23: end for
24: #pragma oss taskwait
```

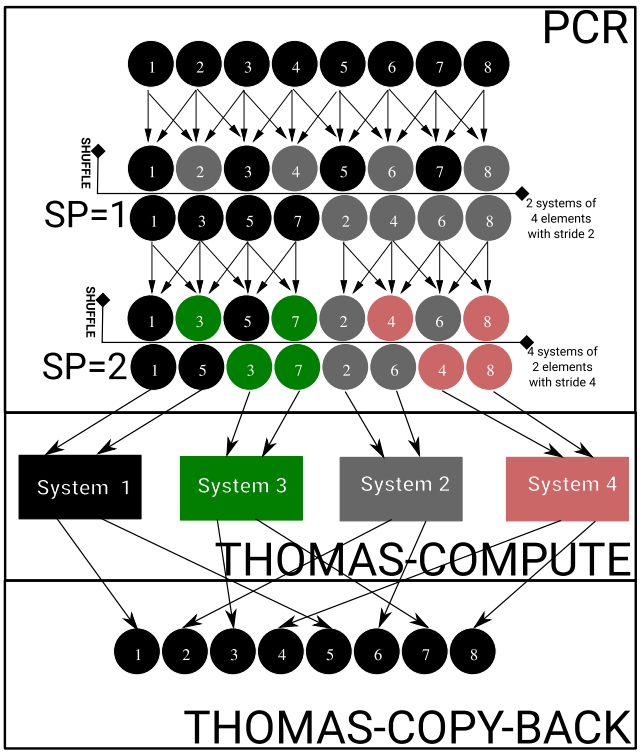

FIGURE 4. Example of the Shuffled PCR+Thomas implementation with $n = 8$ and $SP = 2$.

to be computed using Thomas. This involves the copy of all the elements of the 4 arrays of size n from strided positions to consecutive ones. Besides, the later the method switches from Thomas to PCR, the more strided the access pattern is. This has a negative impact on the cache performance. An alternative, would be to implement Thomas on strided position, but we found this approach to be more efficient because it improves the locality within each core. In summary, the main reasons for the poor performance of the **Thomas-Copy** stage are that: i) it involves the full copy of the four arrays of size n and that ii) these copy operations present poor locality.

We evaluated several strategies to improve the performance, but none of them was so successful as merging this copy with the last step of PCR. This means that in the last iteration of PCR the data is copied in the appropriate positions of the Thomas buffers instead of in the strided positions of the original arrays. In the next section (Section VI) we show the performance gains associated to this optimization. In the rest of the paper, we call this variant the PCR+Thomas Overlapped (P+TO) version.

a: USING TASK GRAPH PARALLELISM

The OmpSs programming model allows us to generate finer grain tasks specifying the *in* and *out* dependences between the tasks. This feature allows us to generate a different parallelization strategy for the Thomas part of the method. On top of the P+T implementation, the code uses now one task pragma for each one of the three stages (Copy, Compute and Copy-back). The dependences of these tasks are specified carefully in order to allow as much overlapping as possible between the tasks. We call this variant the PCR+Thomas tasked (P+TT) implementation.

B. IMPROVING PCR

PCR consumes a big percentage of the total execution time of the P+T method. In low-latency oriented architectures (big caches memories) it is not efficient to solve one large tridiagonal system just using PCR. The role of PCR in the P+T method is just to break down one large system into a certain amount of independent smaller systems. However, this stage still consumes a significant percentage of the execution time, and thus it is worth to optimize it as much as we can.

One of the main problems of PCR is its poor data locality because the computation is done using strided positions. The stride is initially one and it is multiplied by 2 in each subsequent PCR level (see Figure 1). In order to alleviate this problem, we propose a variant of the PCR method, called the Shuffled PCR algorithm. Figure 4 shows an example of this algorithm for $n = 8$ and $SP = 2$. It is actually the same example used in Figure 2, which can eases the comparison of both implementations. The system has initially one system of n equations, n being the size of the diagonal. After computing the first PCR level, two systems are generated, one with the values of the odd positions and other one with the even

positions. Shuffling the results of this level, the odd positions are consecutively placed in the first half of the array and the even ones in the second half. In the next PCR levels, the same procedure is applied recursively on each system created.

The advantages of this approach are twofold. First, the calculations are always done with stride one, which should have a positive impact on the cache performance. Second, the **Thomas-Copy stage** is not required, as, after each PCR level, the information of each system is already placed in consecutive positions (see example in Figure 4). Nevertheless, the **Copy-Back** stage finds the result shuffled and it needs to move the information back to the appropriate positions in the *y* resulting array.

C. LOW LEVEL OPTIMIZATIONS

The implementations described so far have been improved using a set of low-level optimizations. Some of them have been successful, being able to improve the performance. For instance, the PCR phase uses a double buffer for all the data structures involved in the computation, as in each step of the algorithm the output of the previous step is used as an input. Also, PCR has to calculate long mathematical expressions, and the code has been written trying to maximize the usage of processor registers to store intermediate and replicated results. Finally, the compiler reported the automatic vectorization of some blocks of the code.

Other optimizations were tried but they had a negative impact or not impact at all on the performance. For instance, in order to improve the vectorization, we made two separate attempts. On one side, the PCR phase of the P+TO implementation was reimplemented to remove all the conditional statements inside the loops and to avoid the usage of intermediate registers in the calculations. Both things were apparently preventing the compiler to automatically vectorize the PCR phase. With these modifications, the automatic vectorization worked on one of the loop copies of this phase, but it was not good enough to improve the performance of the P+TO original version. The second attempt was to hand-vectorize using intrinsics in the inner PCR loop of the P+TO version, but this vectorization did not improve significatively the original P+TO, and it was outperformed in some cases by the SP+T version. It is also worth to mention that the PCR phase of SP+T is not a good candidate to be hand-vectorized because, the shuffled approach followed in this phase generates different access patterns in loads and stores, which makes hand-vectorization extremely difficult (if not impossible) and probably highly inefficient. Also, the tiling technique was applied in several points of the code to improve the cache performance but with no success. Finally, different policies and chunk sizes were tried in the scheduling of the loop pragmas.

D. MEMORY REQUIREMENTS

In Table 1 we can see the memory occupancy of each of the implementations evaluated and proposed in this paper. In all the implementations, we do not modify the input matrix

TABLE 1. Memory space consumed by each of the *gtsv* implementations.

GB	*gtsv* implementations					
N	CR	Pure-PCR	P+T	P+TO	P+TT	SP+T
8,388,608	0.5	0.75	1	1	1	0.75
16,777,216	1	1.5	2	1	2	1.5
33,554,432	2	3	4	4	4	3
67,108,864	4	6	8	8	8	6

(*a, b and c* vector in section II). However, on exit, the right-hand side (*y* vector in section II) is overwritten by the vector solution *u*. This is in the line of the vendors libraries as the *gtsv* routine of the MKL Intel library. As shown, and to avoid overwriting the input matrix, we are in need of extra memory. *CR* is in need of less extra memory than the other implementations, using one extra buffer per vector. *PCR*, due to its higher parallelism and memory access pattern (see Figure 1), is in need of two extra buffers per vector to avoid race conditions. In the *P+T*, *P+TO* and *P+TT* implementations is required more memory because of the switch between *PCR* and *Thomas*. After computing *PCR*, we need extra memory to store the output of *PCR* in other vectors using the corresponding data-layout (stride 1) to compute the *Thomas* phase. Unlike the previous implementations, the *SP+T* does not require the extra memory to compute Thomas by shuffling the output in every *PCR* level, adapting the data-layout on the fly.

V. ERROR ANALYSIS

In this section, we perform the error analysis of the methodology proposed in this paper, that is PCR+Thomas, and the pure-PCR algorithm, w.r.t. two reference codes. One is the sequential implementation of the state-of-the-art Thomas method, and one is the MKL gtsv routine. We assume that the matrices are well-conditioned (diagonal dominant). We initialize the matrices randomly. Unlike the other methods, the MKL routine computes pivoting. We graphically illustrate the maximum error, the maximum difference found between PCR+Thomas and pure-PCR w.r.t the sequential Thomas implementation and the MKL routine, in Figure 6-left. In a similar way, Figure 6-right illustrates the average error ($\sum_{i=0}^{n} |RHS_{ref}[i] - RHS[i]|/n$). As shown, although the pure-PCR implementation presents a worse accuracy than PCR+Thomas, this is in the line with the results shown in [17], both implementations present a good enough numerical precision compared to the reference implementations. In fact, the error is less when comparing with the MKL routine. The error for the PCR+Thomas algorithm depends on the switch point. In the cases shown in the graphs (Figure 6) we have used a switch point equal to 3. The smaller the switch point, the error is more similar to the pure-PCR's error, or in other words, the bigger the switch point, the error is closer to the sequential Thomas's error.

VI. PERFORMANCE EVALUATION

In this section we evaluate the following Tridiagonal resolution versions:

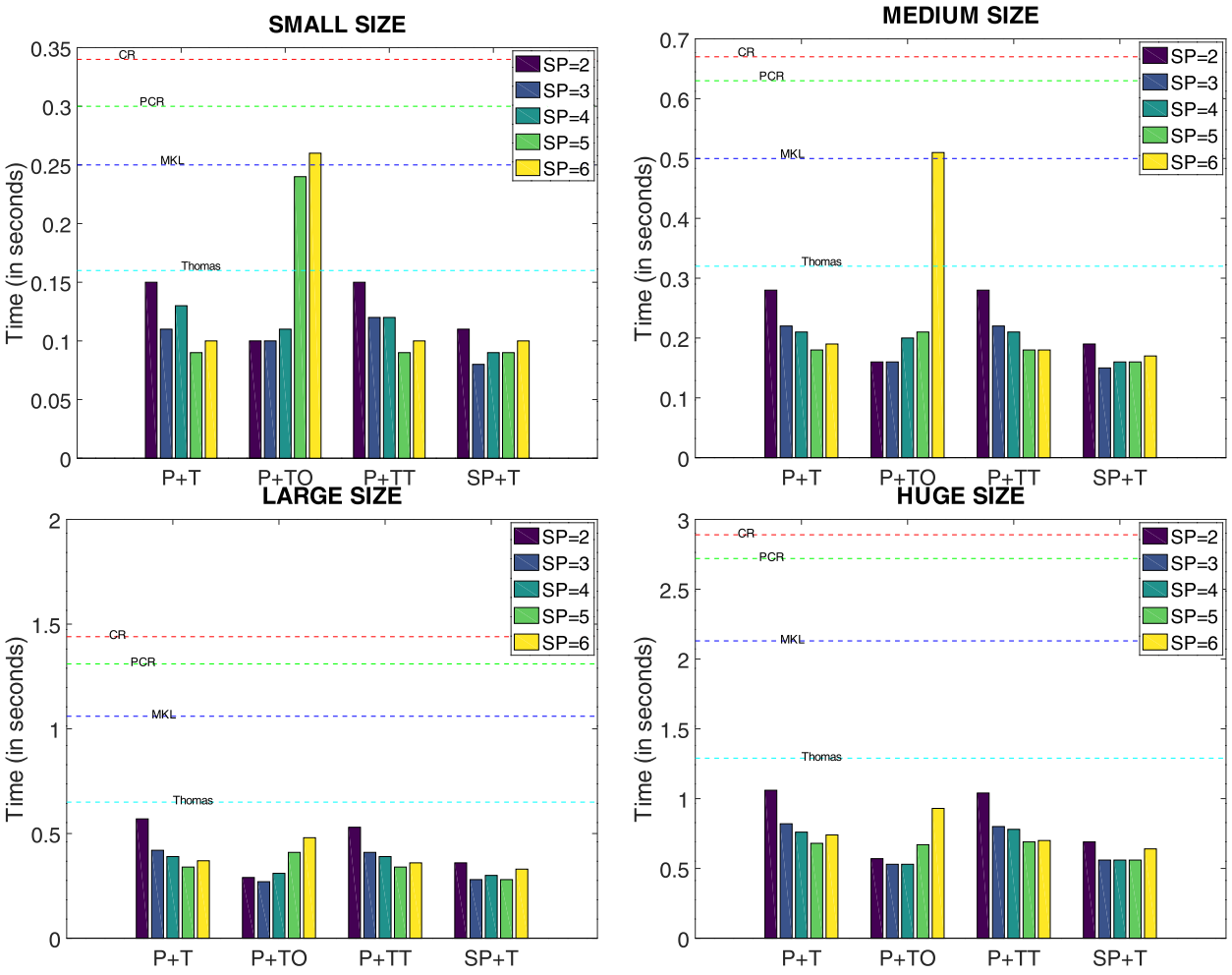

FIGURE 5. Execution times of MKL, pure PCR, CR, and the optimized P+TO, P+TT and SP+T using different problems sizes and switch points.

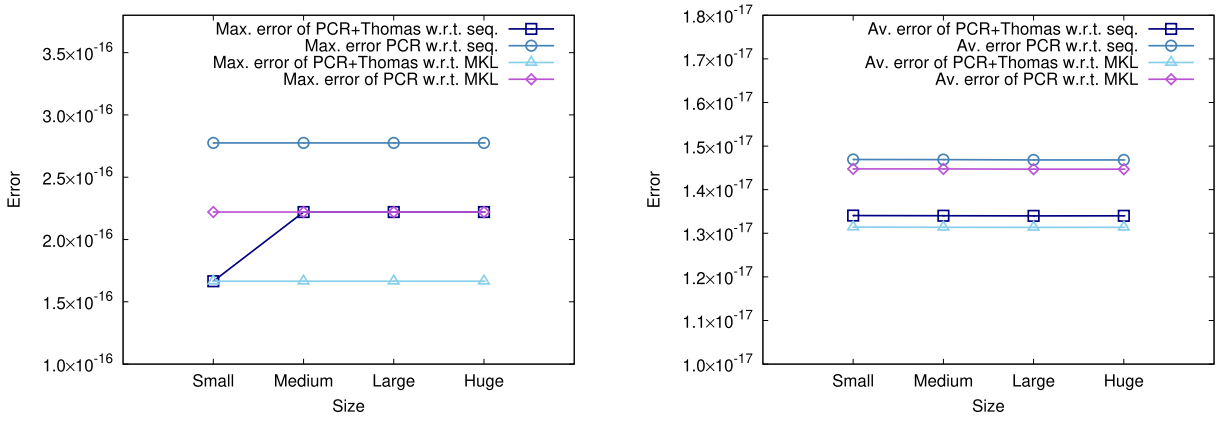

FIGURE 6. Maximum error (left) and average error (right) of the methodology proposed in this paper (PCR+Thomas), with a switch point equal to 3, and pure-PCR algorithm w.r.t the sequential Thomas and the *gtsv* routine of MKL.

- **Thomas**: It is a sequential implementation of the Thomas algorithm. Let us recall that this algorithm cannot be parallelized.
- **MKL**: It uses the MKL implementation of the dgtsv routine. This routine uses pivoting and is not parallelized.
- **CR**: It is an implementation of the pure CR algorithm. The parallelization is made using OmpSs *loop pragmas*.
- **PCR**: It is an implementation of the pure PCR algorithm. The parallelization is made using OmpSs *loop pragmas*.
- PCR+Thomas (**P+T**): It is the baseline implementation of the idea of combining PCR and Thomas.

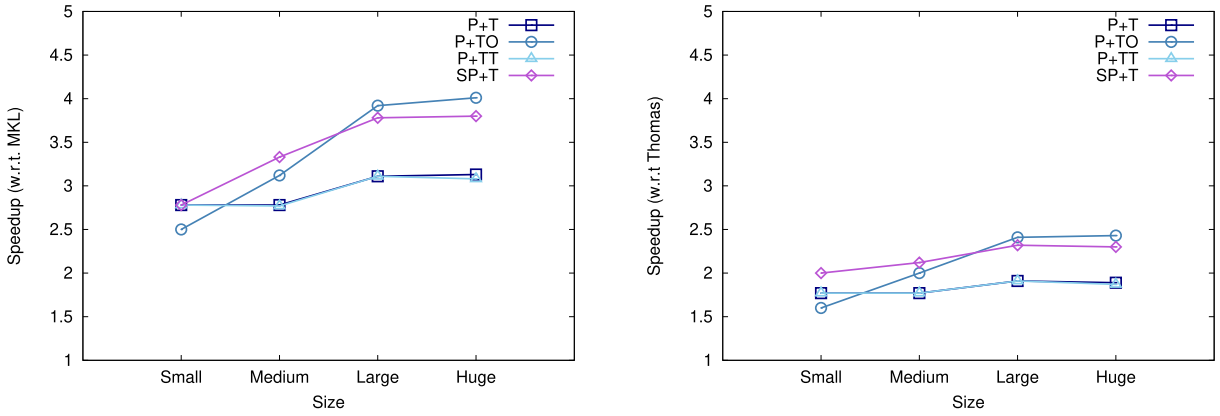

FIGURE 7. Speedup (using the best switch point) of the optimized variant with respect to MKL (left) and Thomas (right) using different problem sizes.

The parallelization is made using OmpSs *loop pragmas*. The switch point (SP) is configured manually. This number indicates how many systems are created using PCR before computing Thomas.

- P+T Overlapped (**P+TO**): This version joins the computation of the last PCR level with the Thomas-Copy stage. It is parallelized using OmpSs *loop pragmas*.
- P+T Tasked (**P+TT**): In this variant, we make use of the OmpSs tasks to parallelize the Thomas stages of the algorithm.
- Shuffled P+T (**SP+T**): It improves the data locality of PCR using the shuffled variation of that algorithm introduced in this paper. It is parallelized using OmpSs *loop pragmas*.

The experiments have been performed on one node of the Mare Nostrum supercomputer with 2x Intel Xeon Platinum 8160 with 24 cores each running at 2.1 GHz. The RAM memory is composed of 12 DDR4-2667 modules of 32GB. The version of MKL is 2018.1 and the one of GCC, 7.2.0. We have used the latest version of the OmpSs programing model (mercurium compiler + nanos 6 runtime), OmpSs-2. In the following we will call this platform M48. All the computations were carried out using double precision. Note that throughout all of our experiments we ensure the cache of each processor is flushed before every invocation of each of the implementations to be tested in order to avoid obtaining misleading performance results. By neglecting this step we can obtain performance results up to 4 times faster than those reported here in the cases where the data fits in the cache memory. This is consistent with observations described in [18]. The problem sizes used in this evaluation are: **Small** ($n = 8388608$), **Medium** ($n = 16777216$), **Large** ($n = 33554432$) and **Huge** ($n = 67108864$). Also, for each variant, different switch points are explored (2, 3, 4, 5 and 6).

Figure 5 shows the execution times of the 6 optimized variants using 4 problem sizes and 5 switch points. As shown, the P+TO and SP+T are the best variants. All the variants are faster than MKL, single-threaded Thomas and pure-PCR, except P+TO with a switch point (SP) equal to 6. As shown in Fig. 5, the best switch point depends on the variant used. For instance, for the P+TO and SP+T, the best switch point is 3, creating 2^3 sub-systems to be computed by Thomas. However, for the remaining variants, P+T and P+TT, the best switch point is 5, creating 2^5 sub-systems to be computed by the Thomas stages. This means that the first approaches, not only need a lower switch point, but also a lower number of cores. In fact, P+TO and SP+T are able to outperform the other variants using a smaller number of cores.

It is important to note that the SP+T approach, unlike the P+TO, presents a more stable behavior. We do not find important variations in time using a different switch point, probably because these variations are usually due to the PCR stage, and this stage is specially optimized in SP+T.

The speedup with respect to MKL and to Thomas is graphically illustrated in Figure 7. We see that, for any problem size, the speedup of the SP+T version is always greater than 2.7 w.r.t. MKL and greater than 1.5 w.r.t. single-threaded Thomas, being the fastest approach for the small and medium systems. Although the speedup of the P+TO variant is worse for these sizes, it turns out to be the best for large and huge systems. Unlike the previous implementations, the other two approaches, P+T and P+TT, present a similar performance. This proves that the optimization carried out in the P+TT approach is not as effective as expected. It is important to note that the bigger the system, the higher speedup, achieving a peak speedup about 4 and 2.5 w.r.t. MKL and single-threaded Thomas respectively, using the P+TO version.

Figure 8 graphically illustrates the execution time consumed by all P+T variants using different switch points for the huge problem size. The best switch point for the two fastest versions, P+TO and SP+T, is 3, thus creating 8 tridiagonal systems to be computed in the Thomas stage. Unlike the previous variants, the approaches based on P+T and P+TT present a very similar trend. In both cases, the best switch point is 5, creating 32 triangular systems. The techniques that improve the integration and swapping between both methods, PCR and Thomas, have important consequences on the performance, as shown by the performance achieved by the P+TO and SP+T approaches. Given these results, P+TO and SP+T are proven to be, not only the

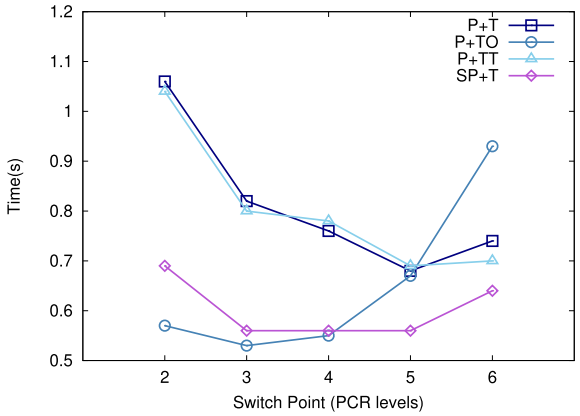

FIGURE 8. Execution time of the optimized variants using different switch points.

fastest approaches, but also the more energy efficient implementations, as these, unlike the other variants, require fewer cores in the Thomas stage to achieve the peak performance.

Comparing the analysis carried out in Section II (see Fig. 3) and the results obtained and graphically illustrated in Fig. 8, we can confirm the effectiveness of this analysis. In fact, for the best approaches, P+TO and SP+T, this analysis is able to predict the best switch point selection, using a simple computation, which only requires the number of cores available and the size of the system to be solved.

In order to carry out a deeper analysis on the strategies presented, we have used the packages Extrae + Paraver [19]. Extrae is a dynamic instrumentation package to trace programs compiled and run using one of the next programing models, OpenMP, OmpSs, pthreads, the message passing (MPI) programming model or a combination of the previous programming models (different MPI processes using OpenSs threads within each MPI process). Extrae generates trace files that can be later visualized with Paraver.

Figure 9 illustrates the traces of the four optimized versions, using the best switch point for each version observed in Figure 5. In all of them, the first part (red) corresponds to the PCR phase. As commented before, it is necessary to synchronize all the threads at the end of each PCR level. This is clearly observed in the traces. Also, we can see the number of PCR levels computed, 5 in *P+T*, 3 in *P+TO*, 5 in *P+TT* and 3 in *SP+T*. In *P+TO* the last *PCR* level (dark-red) computes not only the last *PCR* level but also the Thomas-Copy step (see Figure 2). Also, in the *SP+T* implementation, the *PCR* computation is different w.r.t. the other implementations due to the shuffling performed in this phase. The number of threads in the *Thomas* phase (violet) depends on the number of steps performed during the *PCR* phase. In this sense, in *P+T* and *P+TT* we see 32 threads (2^5, 5 PCR levels), and 8 threads in *P+TO* and *SP+T*. In *P+TT* we make use of tasks to compute the Thomas-Copy (violet), Thomas-Compute (dartk-red) and Thomas-Copy-Back (red). In this implementation, although there is synchronization between these steps, we make use of data-dependences between them to guarantee the correct execution.

The P+T version consumes about half of the time in computing PCR and the other half in computing Thomas. As expected, P+TO is in need of a larger last PCR level (dark red), where the result of the last PCR level computed is already copied to the appropriate positions of the buffers to be computed by Thomas. This extends the PCR stage but it reduces the overall execution time. The time consumed using a switch point of 3 and 4 are quite similar. This is in agreement with the analysis shown in Fig. 3.

P+TT makes use of data-dependent tasks for Thomas-Copy (violet), Thomas-Compute (dark red) and Thomas-Copy-back (green). The performance of this approach is not very competitive. The Thomas part of the method requires more time than in the other approaches, despite using finer grain tasks. The reason is that the data dependences of the tasks serialize the Thomas-Copy, Thomas-Compute and Thomas-Copy-back stages, which should result into a task footprint similar to that of the P+T version. Nevertheless, it is slightly different due to the scheduler, which sometimes migrates the resolution of one system from one core to another one.

Finally, SP+T reduces the time consumed by PCR (red), which it is the main motivation behind this optimization. The reason of this is twofold: i) A better data locality, and ii) A smaller switch point. Also the Thomas-Copy stage is not required, as the information is already placed in consecutive position before it is processed using Thomas. Finally, the Thomas-Compute and the Thomas-Copy-back stages are combined into one single tasked loop (violet). These two last stages are larger than in P+TO due to the need to unshuffle the result at the end.

In order to visualize the benefit achieved using the proposed strategies against the use of CR, and pure-PCR, Figure 10 compares the traces of these two last versions with the best optimized one for the biggest size evaluated, which is the P+TO approach, using a switch point of 3. As commented, the CR algorithm is composed by two phases, the reduction phase, where the parallelism (number of independent elements) is reduced step by step, and the substitution, with the opposite behave, the parallelism is bigger step by step. It is necessary to synchronize the execution at the end of each of the steps to avoid race conditions. This can be seen in the Figure 10. The pure PCR approach performs PCR until the end, where the characteristics of the multi-core architectures are not able to compensate the large number of extra operations required by PCR with respect to other implementations. On the other hand, our best optimization is much faster, as it switches from PCR to Thomas.

We examined the (instruction per cycles) IPC statistics in order to find out how much our best implementation improves the efficiency of the algorithm. While pure PCR, achieves an average IPC of 0.59 with peaks of 0.94, the SP+T version achieves an average value of 0.87 with peaks of 1.95. These values of the IPC confirm that this is a memory bound problem, and that the SP+T version makes a very good job alleviating this problem.

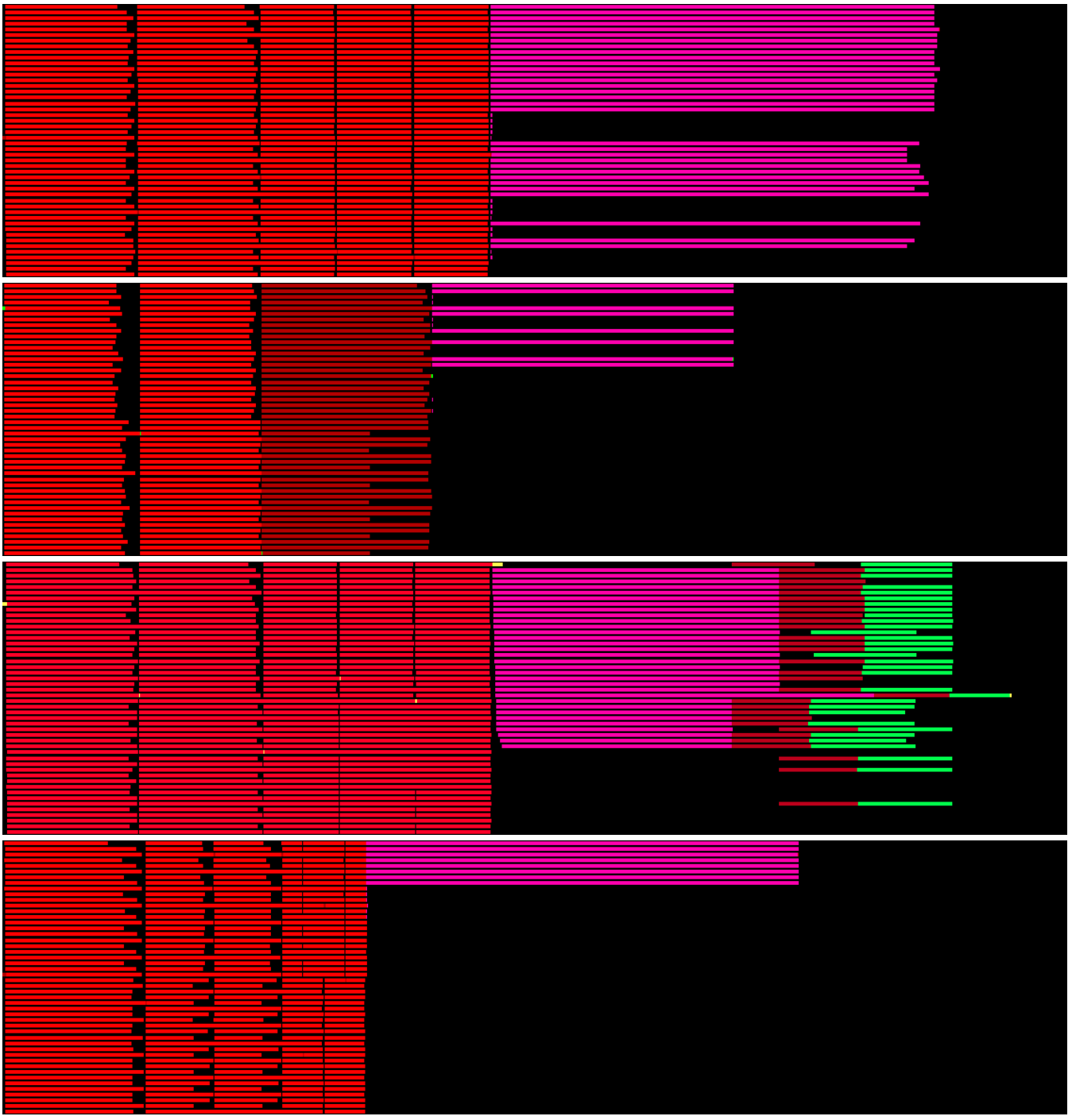

FIGURE 9. Traces of the three optimized versions (from top to bottom: P+T, P+TO, P+TT, SP+T) using the best switch point selection, 5, 3, 5 and 3, respectively.

A. SCALABILITY

Finally, the ability of the proposed implementation to take advantage of additional computational resources has to be evaluated. Figure 11 shows a comparison of the execution time of our two best implementations (P+TO and SP+T) for different switch points (2, 3, 4, 5 and 6) using two different machines: the machine used so far in the experiments (M48) and one with less computational resources (M16). This latter machine is equipped with 2x Intel Xeon E5 2670 with 8 cores each running at 2.6 GHz, and 128GB of RAM memory. The figure shows that both implementations performs clearly better in the M48 machine than in the M16 one, which proves that both implementations can take advantage of the additional computational resources available in M48. Let us

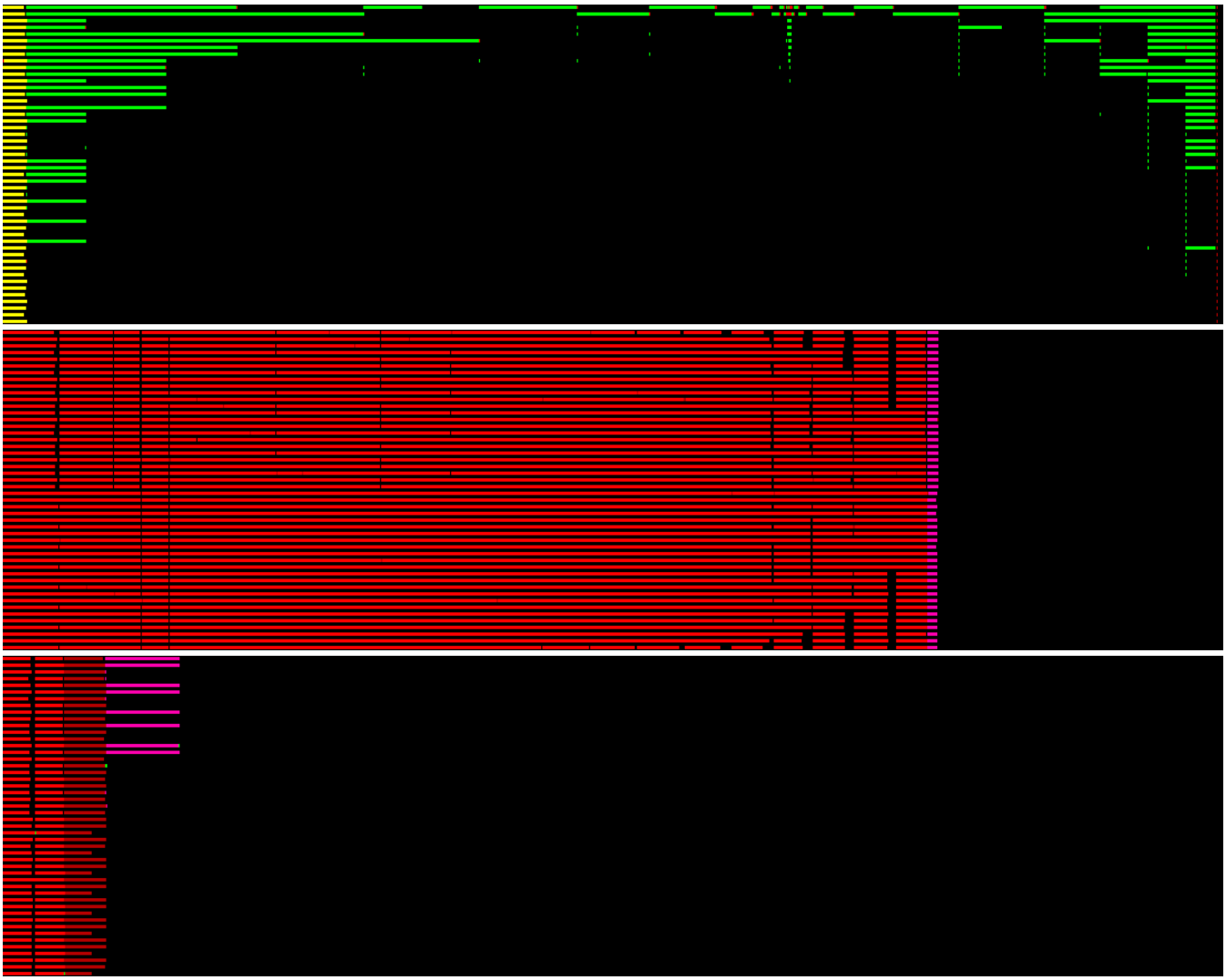

FIGURE 10. Trace of the CR (top), pure PCR (middle) and the best version with the best switch point selection (bottom). The best version is P+TO with SP=3. All the traces have the same time-scale.

recall that one of the bottlenecks of our implementations is the PCR phase, where we can benefit of the availability of an increasing number of cores. That is the reason why in the M16 machine the best switch point is smaller than in the M48 machine for the P+TO algorithm, as in this latter machine the PCR phase bottleneck is in part alleviated by the availability of 32 additional cores. This tendency is less clear in the SP+T version where we have used an improved implementation of the PCR phase, and we can run PCR longer, which allows to generate more independent systems.

VII. RELATED WORK

We can find multiple works that attempt to parallelize the tridiagonal solve on parallel computers. Probably, the most important reference of the present work is the work of [16]. In this work, the authors combine both no-pivoting methods, PCR and Thomas, but on GPUs, using a completely different set of optimizations adapted to the GPU-based architectures. Although the optimizations presented are effective when computing more than one tridiagonal systems in parallel, the implementation turns out to be inefficient when computing one single large tridiagonal system, being about a 30% slower than the CPU sequential counterpart (MKL). Other works are focused on solving multiple and small tridiagonal systems by using no-pivoting solvers on NVIDIA GPUs, as the works presented by [13] and by [20]. Recently, cuSparse library, the reference library for sparse computation on NVIDIA GPUs, included a new routine (*gtsvInterleavedBatch*) based on no-pivoting Thomas [21]. Other interesting reference is the work of [9], in this case the authors implement a stable numerical solver on GPUs for large tridiagonal systems (from 4 Millions to 128 Millions of equations) based on Spike (tridiagonal solve with pivoting), which is able to achieve good numerical results even on ill-conditioned matrices, paying an extra computational cost due to the overhead of pivoting. Reference [2], [3] proposed

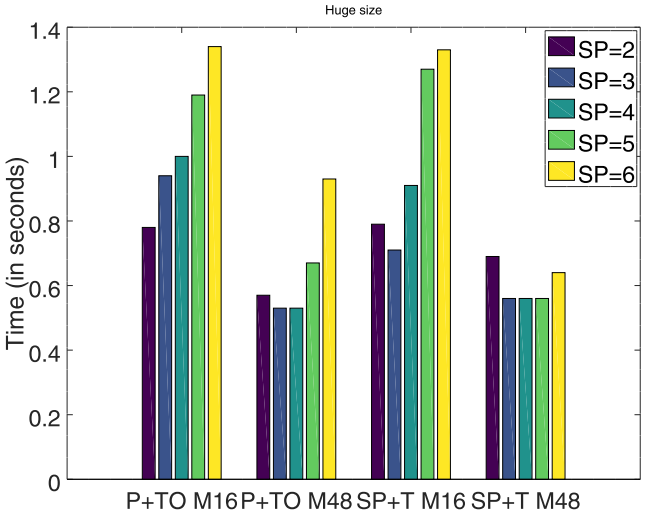

FIGURE 11. Execution times of the P+TO and the SP+T versions in the M16 machine and the M48 machine, using switch points of 2,3,4,5 and 6 respectively.

an heterogeneous (CPU + GPU) implementation to accelerate the solving of Block-Tridiagonal systems, where multiple tridiagonal systems are solved on GPU, making use of the PCR-CR algorithm, while other tridiagonal systems are computed simultaneously on CPU, using the Thomas algorithm.

VIII. CONCLUSION

The parallel implementations presented in this paper focus on the solving of large tridiagonal systems on multi-core architectures. Our implementations are based on combining the parallel PCR and the Thomas algorithms. Several variants of this idea have been proposed, trying to squeeze the maximum performance of modern multi-core CPUs addressing the main problems of this approach. On one side, the PCR+Thomas algorithm requires memory accesses and data movements with poor locality. The SP+T version is actually our best attempt to address this problem, although the P+TO version also tries to alleviate this problem overlapping the Thomas-Copy stage with the last PCR iteration. On the other side, the selection of the switch point between PCR and Thomas is essential to achieve a high performance. In this line, the paper proposes an effective analysis to select the switch point for a given platform (number of cores) and problem size.

The comparison of the execution time of our versions of PCR+Thomas and the MKL library shows that our best implementations can be up to 4 times faster than the MKL implementation and up to 2.5 times faster than the single-threaded Thomas. A study of the influence of the switch point in the performance shows that the performance analysis proposed in this paper is really effective to select the switch point between PCR and Thomas. An insightful analysis of the traces of the different versions of the algorithm reveals that the P+TO and SP+T versions of the code accomplish their objectives. This way, P+TO removes the Thomas-Copy stage, which is overlapped with the last PCR iteration. This has a positive effect on the performance. The SP+T version shortens the PCR part of the algorithm, although it increases slightly the Thomas part due to the need to unshuffle the result in the Thomas-Copy-Back stage. Finally, the trace of the P+TT version reveals that its attempt to extract finer-grain parallelism does not have a positive effect on the performance, due to the serialization of the three Thomas tasks for each processed system.

As future work, we plan to extend this work to make use of the Spike algorithm for more numerically stable results. We also plan to compute large systems with multiple right-hand sides. Furthermore, we want to implement the same idea using different features and programming models, such as OpenMP or StarPU, among others.

REFERENCES

[1] C. T. Ho and L. Johnsson, "Optimizing tridiagonal solvers for alternating direction methods on Boolean cube multiprocessors," *SIAM J. Sci. Stat. Comput.*, vol. 11, no. 3, pp. 563–592, May 1990.

[2] P. Valero-Lara, A. Pinelli, J. Favier, and M. P. Matias, "Block tridiagonal solvers on heterogeneous architectures," in *Proc. IEEE 10th Int. Symp. Parallel Distrib. Process. Appl.*, Jul. 2012, pp. 609–616.

[3] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias, "Fast finite difference poisson solvers on heterogeneous architectures," *Comput. Phys. Commun.*, vol. 185, no. 4, pp. 1265–1272, Apr. 2014.

[4] L.-W. Chang, M.-T. Lo, N. Anssari, K.-H. Hsu, N. E. Huang, and W.-M. W. Hwu, "Parallel implementation of multi-dimensional ensemble empirical mode decomposition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2011, pp. 1621–1624.

[5] G. R. Halliwell, "Evaluation of vertical coordinate and vertical mixing algorithms in the HYbrid-coordinate ocean model (HYCOM)," *Ocean Model.*, vol. 7, nos. 3–4, pp. 285–322, 2004.

[6] A. Greenbaum, *Iterative Methods for Solving Linear Systems*. Philadelphia, PA, USA, SIAM, 1997.

[7] P. Valero-Lara and I. Martínez-Perez, A. J. Peña, X. Martorell, R. Sirvent, and J. Labarta, "cuHinesBatch Solving multiple hines systems on GPUs human brain project," in *Proc. Int. Conf. Comput. Sci. (ICCS)*, Zurich, Switzerland, Jun. 2017, pp. 566–575.

[8] P. Valero-Lara, I. Martínez-Pérez, R. Sirvent, A. J. Peña, X. Martorell, and J. Labarta, "Simulating the behavior of the human brain on GPUs," *Oil Gas Sci. Technol.-Rev. IFP Energies Nouvelles*, vol. 73, p. 63, Nov. 2018. doi: 10.2516/ogst/2018061.

[9] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W. M. W. Hwu, "A scalable, numerically stable, high-performance tridiagonal solver using GPUs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Los Alamitos, CA, USA, 2012, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389033

[10] A. Duran *et al.*, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 173–193, Jun. 2011. doi: 10.1142/S0129626411000151.

[11] J. Ansel *et al.*, "PetaBricks: A language and compiler for algorithmic choice," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, New York, NY, USA , Jun. 2009, pp. 38–49. doi: 10.1145/1542476.1542481.

[12] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 115–126, Jan. 2010. doi: 10.1145/1837853.1693471.

[13] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 127–136, May 2010. doi: 10.1145/1837853.1693472.

[14] H.-S. Kim, S. Wu, L.-W. Chang, and W.-M. W. Hwu, "A scalable tridiagonal solver for GPUs," in *Proc. Int. Conf. Parallel Process.*, Sep. 2011, pp. 444–453.

[15] N. Sakharnykh, "Efficient tridiagonal solvers for ADI methods and fluid simulation," in *Proc. NVIDIA GPU Technol. Conf.*, Sep. 2010, pp. 1–51.

[16] A. Davidson, Y. Zhang, and J. D. Owens, "An auto-tuned method for solving large tridiagonal systems on the GPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2011, pp. 956–965.
[17] P. Valero-Lara, I. Martínez-Pérez, R. Sirvent, X. Martorell, and A. J. Peña, "NVIDIA gpus scalability to solve multiple (batch) tridiagonal systems implementation of cuthomasbatch," in *Proc. Int. Conf. Parallel Process. Appl. Math. (PPAM)*, Lublin, Poland, Sep. 2018, pp. 243–253.
[18] R. C. Whaley and A. M. Castaldo, "Achieving accurate and context-sensitive timing for code optimization," *Softw. Pract. Exper.*, vol. 38, no. 15, pp. 1621–1642, Dec. 2008.
[19] G. Llort, H. Servat, J. Gonzalez, J. Giménez, and J. Labarta, "On the usefulness of object tracking techniques in performance analysis," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2013, pp. 1–11. doi: 10.1145/2503210.2503267.
[20] P. Quesada-Barriuso, J. Lamas-Rodríguez, D. B. Heras, M. Bóo, and F. Argüello, "Selecting the best tridiagonal system solver projected on multi-core CPU and GPU platforms," in *Proc. 17th Int. Conf. Parallel Distrib. Process. Techn. Appl.*, H. R. Arabnia, Ed. Providence, RI, USA: CSREA Press, Jul. 2011, pp. 839–845.
[21] P. Valero-Lara, I. Martínez-Pérez, R. Sirvent, X. Martorell, and A. J. Peña, "cuThomasBatch and cuThomasVBatch, CUDA Routines to compute batch of tridiagonal systems on NVIDIA GPUs," *Concurrency Comput., Pract. Exper.*, vol. 30, no. 24, p. e4909, Dec. 2018.

PEDRO VALERO-LARA received the M.S. degree in computer science from the Universidad of Castilla-La Mancha, in 2010, and the Ph.D. degree in computer science from the Complutense University of Madrid, Spain, in 2015. He has been a Researcher with the Barcelona Supercomputing Center, since 2016. His current research interest includes high performance and scientific computing.

DIEGO ANDRADE received the M.S. and Ph.D. degrees in computer science from the Universidade da Coruña, A Coruña, Spain, in 2002 and 2007, respectively, where he has been a Lecturer with the Departamento de Electrónica e Sistemas since 2006. His research interests include performance evaluation and prediction, analytical modeling, and compiler transformations.

RAÜL SIRVENT received the Ph.D. degree in computer science with the Computer Architecture Department, UPC, in 2009. He has been involved in research activities at the European Center of Parallelism of Barcelona, from 2002 to 2005. Since 2005, he has been holding a permanent position with the Barcelona Supercomputing Center, Computer Sciences Department as a Senior Researcher. His current research interests include high performance computing programming models, deployment, dynamic resource management and scheduling, and grid and cloud programming models and tools, automatic workflow generation, and fault tolerance mechanisms.

JESÚS LABARTA received the B.S. and Ph.D. degrees in telecommunication technologies engineering from the Technical University of Catalonia, Spain, in 1981 and 1983, respectively. He is currently a Full Professor with the Politècnica de Catalunya and also the Head of the Computer Science Department, Barcelona Supercomputing Center. His research interests include parallel programming models, performance evaluation, and high performance computing.

BASILIO B. FRAGUELA received the M.S. and Ph.D. degrees in computer science from the Universidade da Coruna, Spain, in 1994 and 1999, respectively. He has been an Associate Professor with the Departamento de Enxenarıa de Computadores, Universidade da Coruña, since 2001. His current research interests include programmability, high performance computing, heterogeneous systems, and code optimization.

RAMÓN DOALLO received the Ph.D. degree in physics from the Universidade de Santiago de Compostela, Spain. He has been a Full Professor of the Universidade da Coruña, Spain, since 1999, where he is currently the Head of Computer Architecture Research Group. He has 22 years of experience in research and development in high-performance computing (HPC), covering a wide range of topics such as compilers and programming languages for HPC, parallel and distributed algorithms and applications, management of HPC infrastructures, cluster and grid computing, processor architecture, computer graphics, and distributed geographic information systems. He has published more than 120 technical papers on these topics.