The Journal of Supercomputing (2022) 78:17231–17246
https://doi.org/10.1007/s11227-022-04537-w

PARCSIM: a parallel computing simulator for scalable software optimization

Jesús Cámara[1] · José-Carlos Cano[1] · Javier Cuenca[1] · Mariano Saura-Sánchez[2]

Accepted: 13 April 2022 / Published online: 16 May 2022

Abstract
PARCSIM is a parallel software simulator that allows a user to capture, through a graphical interface, matrix algorithm schemes that solve scientific problems. With this tool, the user can analyse the execution times that would be obtained by using different spatio-temporal mapping of computational tasks on available computational units, parallelism parameters and computational libraries. Furthermore, for complex problem models, the self-optimization engine incorporated in this tool analyses the huge tree of possible calculations grouping and mapping strategies in search of the choice that makes the best use of the available hardware resources. This tool also offers polyalgorithmic resolution by making automatically the best decision between different software approaches to solve a given problem on the hardware system available. This work shows the usefulness of this simulator to efficiently solve hierarchical problems constructed from previously modelled subproblems. This task is performed by reusing, in a scalable way, the optimization information of these subproblems to establish the best execution configuration for the composite problem.

Keywords Hierarchical parallel software · Polyalgorithm · Autotuning

Jesús Cámara, José-Carlos Cano, Javier Cuenca and Mariano Saura-Sánchez have contributed equally to this work.

✉ Javier Cuenca
jcuenca@um.es

Jesús Cámara
jcamara@um.es

José-Carlos Cano
josecarlos.canol@um.es

Mariano Saura-Sánchez
msaura.sanchez@upct.es

[1] Department of Engineering and Technology of Computers, University of Murcia, Murcia, Spain

[2] Department of Mechanical Engineering, Technical University of Cartagena, Cartagena, Spain

1 Introduction

A multitude of scientific problems can be found where it is possible to represent in the form of a directed graph the division of a complex (mechanical, physical, algebraic) system into an ordered set of determined (solvable) subsystems. In this graph, the nodes correspond to blocks of instructions that solve each subsystem, and the directed lines between the nodes indicate the order in which blocks have to be calculated. A study of the dependencies reflected in the graph allows us to identify the subsystems that can be solved simultaneously.

In general, in computational implementations of such problems, the algorithm with the highest potential efficiency, in terms of execution speed, on parallel platforms will be the one that identifies the largest number of blocks of computations that can be executed in parallel. And the fastest implementation will be the one that allocates computational resources in such a way that the parallel execution of blocks is the fastest possible. The searching process for the best implementation is a complex task that can be approached from a pseudo-theoretical point of view (estimating theoretical execution times based on real times measured in executions of the basic operations) [1] or from an experimental point of view (where the best allocation of resources is sought by analysing the behaviour observed after several executions exploring different alternatives) [2]. In this way, PARCSIM, a complete simulator of numerical software executions on heterogeneous parallel platforms, has been designed. This simulator aims to facilitate this search task by providing a graphical interface to guide a non-parallelism expert in selecting the best implementation for the code, thereby achieving optimal utilization of a given hardware platform.

A preliminary version of this tool was designed for the kinematic analysis of multibody systems (MBS) in computational systems [3]. MBS are mechanical systems composed of rigid and flexible bodies connected through mechanical joints which determine the dependencies between the individual bodies and their contribution to the movement of the whole system [4, 5]. The kinematic analysis of a MBS consists of the study of the relation of its components' movements.

This work presents a whole version of PARCSIM where its functionality has been extended and generalized to any types of matrix software, which can run on hybrid platforms composed of a multicore CPU together with a set of massively parallel computational accelerators. In this way, parallelism can be exploited at the implicit level (using multithread libraries which are available for the solution of the constraint equations with both dense [6] and sparse matrices [7]) and at explicit level (different subsystems are solved with shared-memory [8] or GPU [9] parallelism). Therefore, this tool becomes a polyalgorithmic resolution platform, as it offers the user the possibility of automatically comparing and making the best decision between different software approaches to solve a given problem on the hardware system available at any given time.

On the other hand, a hierarchical optimization approach [10] has been incorporated to PARCSIM. Traditionally, hierarchical approaches have been applied in the design of general software [11], and, particularly, in the design of parallel

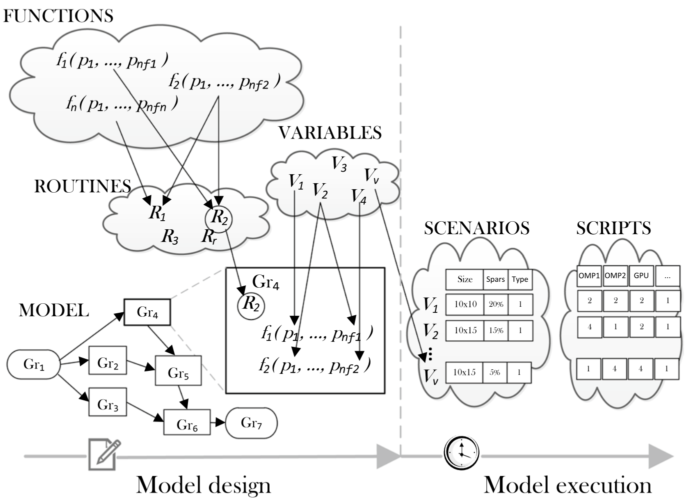

Fig. 1 Schematic view of the concepts used in the simulator and their interdependencies

linear algebra routines [12] and also in the theoretical study of its execution time [13]. In this way, when modelling a high-level routine, this tool allows and facilitates the reuse of models previously made for more basic routines, incorporating them into the nodes of the temporary dependency graph of this high-level routine. Therefore, all the optimization information of the basic routines is automatically incorporated into the model of this higher-level routine, so that, just like the software itself, the global optimization process of this software becomes easily scalable.

Finally, an interactive auto-tuning module has been added to the simulator. This module allows users to test and redesign their software, step by step, on the basis of the feed-back information they receive from this tool at any given moment regarding the performance of their code.

The remainder of the paper is organized as follows. The general structure of PARCSIM is shown in Sect. 2. Section 3 explains how this simulator offers a scalable optimization system for hierarchical organized software and how polyalgorithmic approach is also incorporated in this tool. Finally, Sect. 4 concludes the paper.

2 Description of the simulator

A schematic view of the concepts created and managed in PARCSIM and their interdependencies are shown in Fig. 1.

The linear algebra operations and procedures built into the simulator that can be used by a user are called `functions`. These `functions` can be used by a user to build algorithms for solving certain numerical problems. They can include basic algebraic operations and matrix transformations, such as an addition or a transpose, or higher-level functions, usually imported from external libraries. A

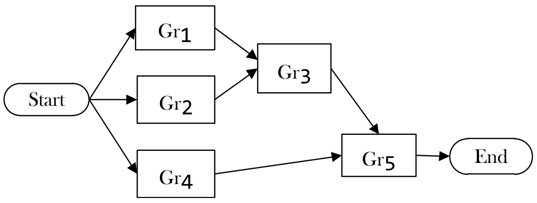

Fig. 2 Graph of solving a numerical problem by decomposition into subproblems

routine represents a sequence of functions, created by a user to solve a given problem or part of a problem. During the simulation, a routine executes in sequence all the functions that compose it. A model is the acyclic graph representation of the algorithm for solving a scientific problem. In this type of graph, the nodes represent the blocks of instructions that execute certain sections of the algorithm, and the directed lines between the nodes indicate the order in which they must be solved. The groups in the simulator are the different subsystems or modules into which an algorithm is divided. The user defines the calculations to be performed to solve each of these groups, which is done by assigning a user routine. A set of variables are used to identify the arrays that will be used as arguments or parameters to the functions used to simulate a given model. The term scenario refers to the nature of the problem to be solved. A scenario, therefore, assigns a number of columns, dispersion factor and type to the variables. A script represents the set of adjustable algorithmic parameters (*AP*) that the software applies during the simulation of a model (number of threads, number of GPUs, computational libraries used,...).

A route specifies a certain way of ordering and grouping the set the computations that solve a model. The set of all valid routes is represented in a tree, where each branch corresponds to a route. Those branches that give rise to computational sequences whose execution is a permutation of another one already generated are eliminated automatically. A graph of this type is shown in Fig. 2, in which seven groups are calculated to solve a complete system. The tree represented in Fig. 3 shows all the possibilities that exist to solve the problem of this example.

Given a particular scientific problem, one of the main objectives of this proposal is to find the order of the calculations that allow the resolution of the model that represents it in the shortest possible time, i.e., to select the best route of the tree. In this way, when the routing tree associated with a given model has been developed, a process of estimating the execution time associated with each branch can be implemented in the simulator to find the one that can theoretically offer the fastest resolution on a given hardware platform. As a starting point for this estimation process, a theoretical–experimental modelling process has been established with the aim of having an agile tool for its approach and management in order to make decisions that are very close to the ideal optimum. In any case, it is important to emphasize that both the simulator and the underlying methodology allow this specific cost estimation process to be modified, extended or replaced as necessary, without affecting the approach as a whole.

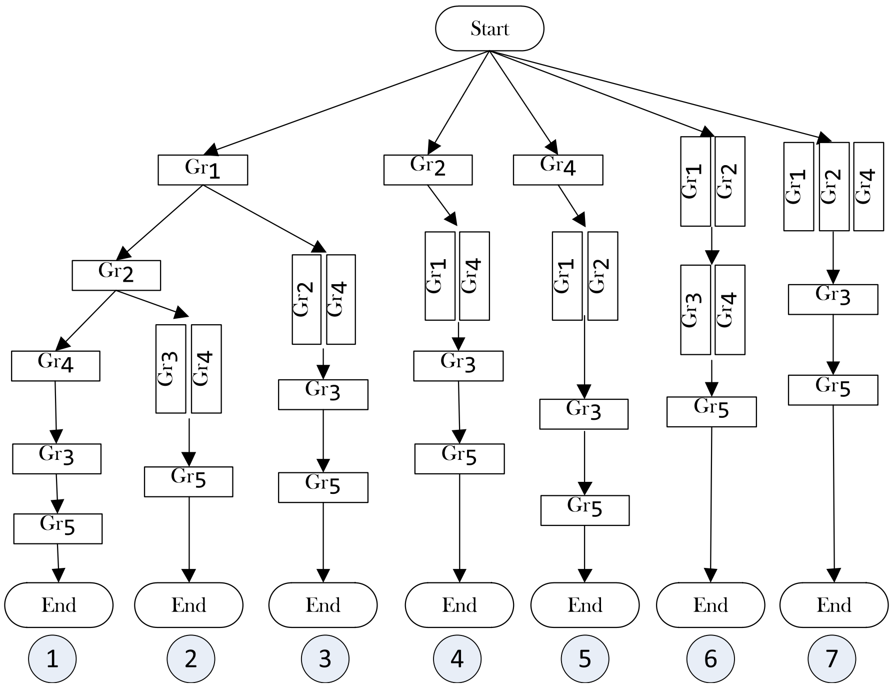

Fig. 3 Tree showing the seven valid computational sequences (`routes`) for Fig. 2. Each branch is a path representing a way of ordering the computations capable of solving the system as a whole. Tree nodes containing more than one group imply the simultaneous resolution of these groups

Thus, this experimental–theoretical model of the execution time has been constructed based on a set of assumptions: as we saw above, the `functions` that are part of a `routine` are executed sequentially, so the resolution time of a `group` *i* containing *k* `functions`, can be modelled as the sum of the execution time of all f_{ik} `functions`, hence $T(Gr_i) = \sum_{j=1}^{k} T(f_{ij})$. On the other hand, since the *n* `groups` contained in a `node` will be executed in parallel, the time needed to perform the computations of an `node` *N* will be equal to the greater of the times needed to solve the `groups` it contains, i.e., $T(N) = \max_{Gr_i \in N} \left\{ T(Gr_i) \right\}$.

With all this, the total time needed to solve the problem following a given `route` *R* will be the sum of the times consumed in the `nodes` that form it can be initially established as shown in (Eq. 1):

$$T(R) = \sum_{i=1}^{\text{nodes}(R)} T(N_i) = \sum_{i=1}^{\text{nodes}(R)} \max_{Gr_j \in N_i} \left\{ \sum_{k=1}^{\text{functions}(Gr_j)} T(f_{jk}) \right\} \quad (1)$$

It can be seen, therefore, that the calculation of the execution time of any branch is based on the known execution times of the `functions` used in that branch. However, in general, these times are different depending on the *AP* and the type of data described in the scenario, *SCN*, on which these `functions` operate. In PARCSIM, the *AP* is formed by the set $\{n_c, n_g, l\}$, where n_c is the number of cores, n_g the number of GPUs and l the type of library. A *SCN* scenario encompasses the size, topology and sparsity factor of the arrays. Therefore, we can rewrite Eq. 1 to reflect the variability of execution times as a function of *AP* and *SCN* when solving a problem along a given `route` *R*, as shown in (Eq. 2):

$$\begin{aligned} T(R, AP, \mathrm{SCN}) &= \sum_{i=1}^{\mathrm{levels}(R)} T(N_i, AP, \mathrm{SCN}) \\ &= \sum_{i=1}^{\mathrm{levels}(R)} \max_{Gr_j \in N_i} \left\{ T(Gr_j, AP, SCN) \right\} \\ &= \sum_{i=1}^{\mathrm{leves}(R)} \max_{Gr_j \in N_i} \left\{ \sum_{k=1}^{\mathrm{functions}(Gr_j)} T(f_{jk}, AP, \mathrm{SCN}) \right\} \end{aligned} \tag{2}$$

where N_i represents the node at level i of the `route` R[1].

After the user has entered in PARCSIM the `model` representing a given algorithm and the data to be handled (Fig. 1: `Model Design`), the simulation process can perform the actual execution of the calculations included in that algorithm (Fig. 1: `Model Execution`). At installation time (`Model Execution: Training Mode`), each `function` is executed for each training `scenario`. The information about the execution times obtained is used to form a `performance database`. After that, taking into account both the experimental–theoretical model of the execution time (Eq. 2) and the information stored in the `performance database`, the best combination of *AP* for each node of the `tree` (Fig. 3) is obtained for each training `scenario`. Then, this information is also stored in the `performance database`. Finally, at execution time (`Model Execution: Autotuning Mode`), the `performance database` is used to find the optimal strategy for calculation ordering and resource allocation to solve a specific user problem.

3 Hierarchical optimization plus polyalgorithm engine

In this section, it is shown how a hierarchical approach can be performed in order to design the algorithm for solving a complex problem composed of a set of subproblems previously modelled in the simulator. Likewise, it will be observed how the optimization and self-optimization process of the complex problem is supported by

[1] In our previous works ([1, 2]), with dense linear algebra routines only, the execution time of each basic atomic routine (basic `function` f_{jk}), which depends on *SCN* and *AP*, ($T(f_{jk}, AP, SCN)$), was called system parameter. Moreover, *SCN* was made up only of the problem size.

the one previously performed for each subproblem, which enhances the process as a whole, leading to a scalable optimization capacity.

In linear algebra, there are routines where the standard approach to solving may not be optimal in terms of computational time. One example is matrix multiplication, which is of particular interest because of its use as basic kernel in solving scientific and engineering problems. In addition to the traditional three-loop solving algorithm, others have been developed, such as block multiplication or Strassen's algorithm. PARCSIM can assist in the optimal development of such higher hierarchical routines that use the basic routines. In addition, this simulator also provides the ability to choose which routine/algorithm is the most appropriate in each case, with a polyalgorithmic resolution approach.

3.1 Experimental platforms

Experiments of this work have been carried out with two multicore CPU+ GPUs configurations:

- **SATURN** is a node with 4 hexa-cores Intel Xeon E7530 with 32 GB of shared-memory at 1.87GHz, and a GPU Tesla K20c (Kepler architecture) with 4800 MBytes in Global Memory and 2496 CUDA cores (13 Streaming Multiprocessors and 192 Streaming Processors per Multiprocessor).
- **JUPITER** is a node with 12 cores and 6 GPUs. The multicore has two hexa-cores Intel Xeon E5-2620 with 32 GB of shared-memory at 2.00GHz. The GPU cards are two Nvidia Fermi Tesla C2075 with 5375 MBytes in Global Memory and 448 cores (14 Streaming Multiprocessors and 32 Streaming Processors per Multiprocessor) and four Nvidia GeForce GTX 590 with 1536 MBytes in Global Memory and 512 CUDA cores (16 Streaming Multiprocessors and 32 Streaming Processors per Multiprocessor).

For both systems, the operating system is Linux (kernel 3.13.0-33-generic #58-Ubuntu), the CUDA version is 7.5, and the compiler used is Intel FORTRAN version 17.0.1, compilation 20161005.

The linear algebra libraries incorporated into the tool in order to select the most appropriate routine for solving each element of the problem to be modelled are MKL [14], PARDISO [15], HSL [16] and MAGMA [17].

3.2 Matrix multiplication: block algorithm

Initially, a basic model, `MATMULT`, for solving a whole matrix multiplication using the basic routine, `RMM`, is create (Fig. 4a). The resolution time of this model with various matrix sizes can serve as a basis for comparison with other possible implementations. In order to have another possible algorithm for solving the matrix multiplication, the matrix B which is part of the product $AB = C$ is going to be divided by columns in a certain number of blocks. This creates the model `MATMULT_COLS_50` shown in Fig. 4b. This model allows us to solve the two multiplications

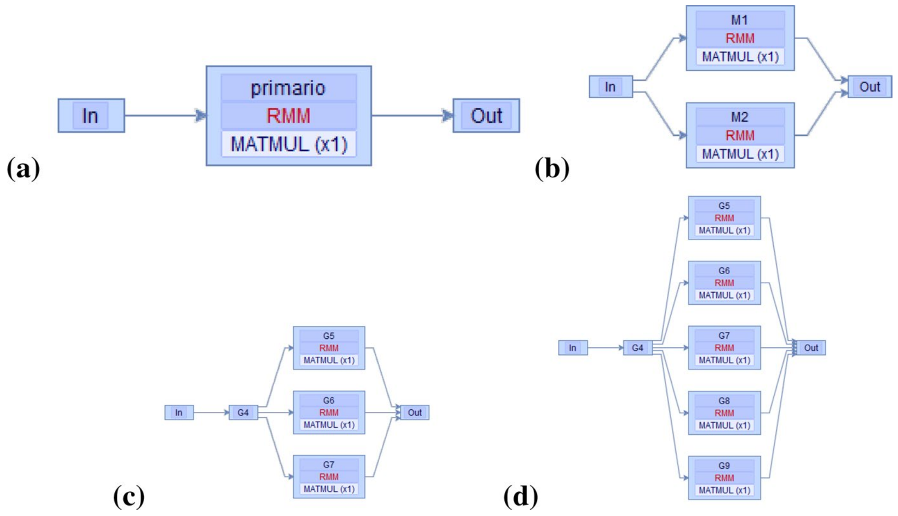

Fig. 4 Simulator representations of the matrix multiplication model (`MATMULT`, `MATMULT_COLS_50`, `MATMULT_COLS_33` and `MATMULT_COLS_20`), $AB = C$, dividing by columns the matrix B (one, two, three and five blocks)

required after dividing the B matrix by columns into two blocks of equal size. Similarly, we create two new models to solve the multiplication when the original matrix B is column-split into three and five blocks, with sizes 33% and 20% of the original, respectively (Fig. 4c, d).

These different models can be solved following as many strategies as their `route trees` indicate. As an example, only those executions that solve simultaneously all their groups (Figs. 5, 6 and 7) are considered in order to compare their performances with the direct multiplication version (Figure 3.2).

The simulator is configured to manage the set of the combinations of models, scenarios and algorithmic parameters. After the execution of each model, log files are generated for each scenario and stored in the database. Table 1 shows the best combinations of first and second level parallelism threads, OpenMP and MKL, respectively, obtained from the simulations on the SATURN platform. It is observed that, with small matrices, direct multiplication is the most efficient using MKL. However, by increasing the size of the matrices, a notable advantage is obtained by splitting the original matrix while introducing parallelism to simultaneously solve groups that handle matrices of sizes smaller than the original matrix. For example, for matrices of dimension 3000, the best strategy is the one applied in `MATMULT_COLS_20`, which block-splits the matrix B into sizes representing 20% of the original size, thus generating five matrices that can be multiplied at the same time by assigning five threads to OpenMP parallelism and four to MKL parallelism. In general, the optimal block size tends to be values that allow the best use of the highest system memory levels. On the other hand, the best number of threads, taking into account both levels of parallelism, tends to be close to the number of CPU cores.

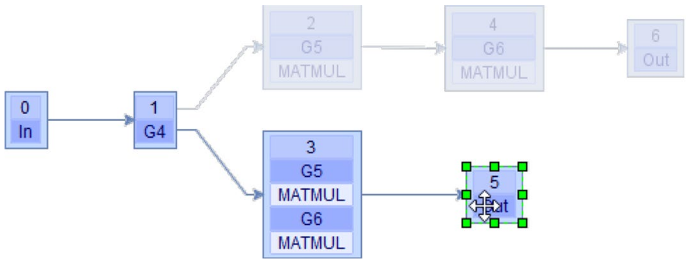

Fig. 5 Selection of the route for applying group parallelism to solve a block matrix multiplication using `MATMULT_COLS_50`

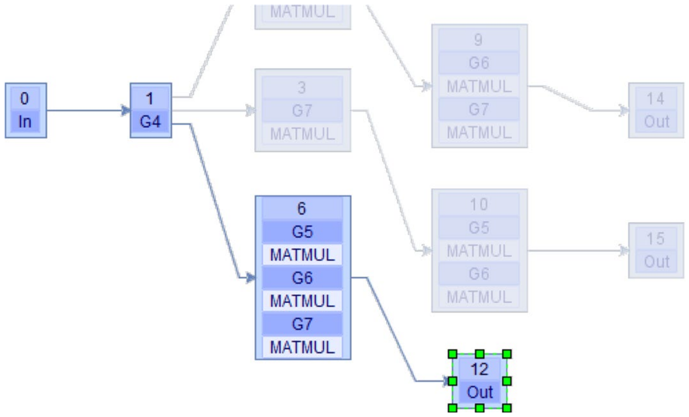

Fig. 6 Selection of the route for applying group parallelism to solve a block matrix multiplication using `MATMULT_COLS_33`

If the hardware now includes the possibility to use multiple GPUs in the resolution of matrix multiplication, the simulator uses the MAGMA library to assign computational tasks to the available GPUs (six GPUs in the case of the JUPITER). To verify this use of multiple GPUs in the simulator, `MATMULT_COLS_33` is used. For this ordering of the calculations, the simulator will be guided by the `route` shown in Fig. 6. Table 2 shows the execution times obtained, where the best performance offered by the GPUs can be observed when the size of the matrices increases.

3.3 Matrix multiplication: strassen algorithm

Strassen algorithm can be represented as a model composed of groups of operations that can be solved in parallel and assigned to the different computational units available. It is stated that, given two square matrices A and B which we consider by simplification of dimensions $A, B \in 2^n \times 2^n$, the multiplication $C = AB$ can be performed by dividing them into blocks of equal size $A_{ij}, B_{ij} \in 2^{n-1} \times 2^{n-1}$:

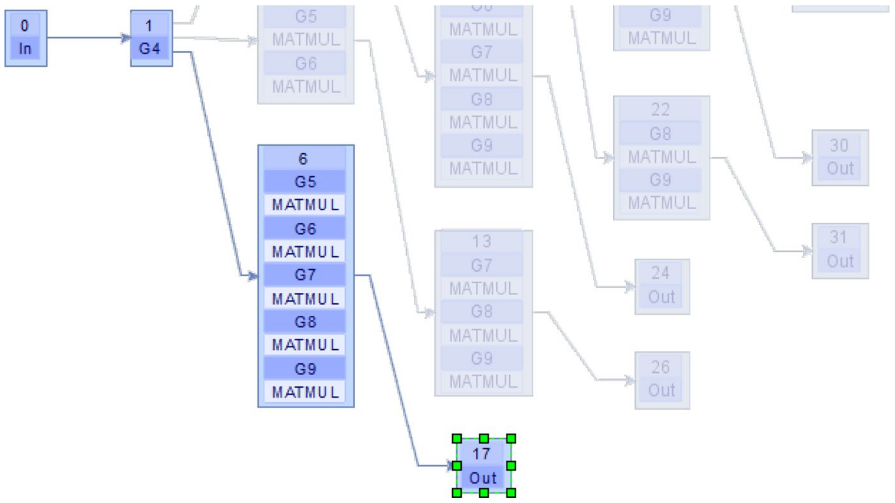

Fig. 7 Selection of the route for applying group parallelism to solve a block matrix multiplication using `MATMULT_COLS_20`

Table 1 Comparison of the execution times (in seconds) obtained in SATURN when simulating traditional matrix multiplication model and with block versions, for different matrix sizes (`nROWS`)

	`MATMUL`		`MAT-MULT_COLS_50`	`MAT-MULT_COLS_33`	`MAT-MULT_COLS_20`
`nROWS`	1×4	1×20	2×12	3×8	5×4
100	**0.00043**	0.00412	0.00065	0.00058	0.00072
500	0.01099	**0.00643**	0.01770	0.01010	0.00702
1000	0.12732	0.06158	**0.03780**	0.04037	0.03883
2000	0.60477	0.27485	0.27300	**0.24414**	0.28912
3000	1.96197	0.78933	0.74721	0.68834	**0.57356**

For each problem size, the minimum execution time is highlighted in bold

Table 2 Comparison of the execution times (in seconds) obtained when simulating block matrix multiplications `MATMULT_COLS_33` with different matrix sizes (`nROWS`), in JUPITER, with 3 OpenMP threads, varying number of MKL threads, or using MAGMA in 3 GPUs

	MKL				MAGMA
`nROWS`	3×1	3×2	3×3	3×4	3 GPUs
500	0.01019	0.00635	**0.00350**	0.00403	0.01055
1000	0.05926	0.02977	0.03592	0.02950	**0.01796**
2000	0.36205	0.16690	0.12079	0.13858	**0.07952**
3000	1.09032	0.86692	0.72478	0.58676	**0.22992**

For each problem size, the minimum execution time is highlighted in bold

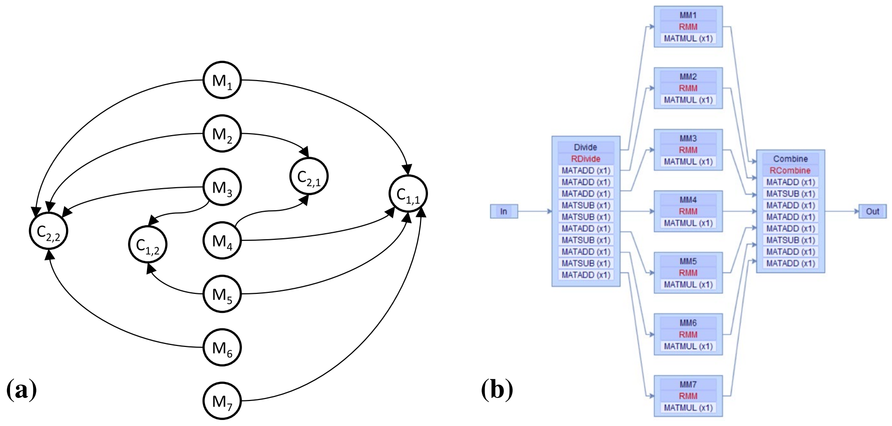

Fig. 8 Graph of the calculations in Strassen algorithm (**a**) and graphical representation in the simulator of the `STRASSEN` model, without recursion, using Strassen algorithm (**b**). In addition to the name of the routines, the functions that compose them are shown

$$A = \left[\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right], B = \left[\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right], C = \left[\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right]$$

This algorithm defines a new set of M_k matrices. These M_k matrices are then used to obtain the final $C_{i,j}$:

$$\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \longrightarrow \quad C_{1,1} = M_1 + M_4 - M_5 + M_7 \\
M_4 &= A_{2,2}(B_{2,1} + B_{1,1}) \longrightarrow \quad C_{1,2} = M_3 + M_5 \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \quad \longrightarrow \quad C_{2,1} = M_2 + M_4 \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \longrightarrow \quad C_{2,2} = M_1 - M_2 + M_3 + M_6 \\
M_7 &= (A_{1,2} + A_{1,1})(B_{1,1} + B_{1,2})
\end{aligned}$$

The M_k matrices can be calculated independently and therefore in parallel. The graph associated with this algorithm calculations depicted in Fig. 8a shows the matrices to be calculated and their dependencies. This makes it possible to determine the order in which each of them must be solved. Three routines are needed to represent this graph in the simulator (Fig. 8b). To begin with, we will instruct the simulator to use a single thread, simulating a single-core hardware system, which allow us to observe the real efficiency of the algorithm without the influence of the improvements that parallelism can bring. Table 3 shows the comparative results obtained on both platforms (JUPITER and SATURN). The reduction in execution times obtained when performing matrix multiplication using Strassen algorithm compared to the conventional method is observed. For matrices of size 6000×6000, the improvement is 8% in SATURN and 6% in the case of JUPITER. For larger matrices, performance is improved by 12% for SATURN and 10% for JUPITER, very close to the theoretical 12.5%.

Table 3 Execution times (in seconds) obtained with the simulator in a multiplication without blocks (MATMUL) and using Strassen algorithm (STRASSEN)

	CPU (with MKL)			
	JUPITER		SATURN	
nROWS×nCOLS	MATMUL	STRASSEN	MATMUL	STRASSEN
6000 × 6000	22.56067	**21.20005**	54.36038	**50.13080**
8000 × 8000	53.34177	**47.80797**	130.18680	**114.81141**
12000 × 12000	179.10999	**160.26036**	428.79211	**379.44492**
	GPU (with MAGMA)			
	JUPITER		SATURN	
nROWS×nCOLS	MATMUL	STRASSEN	MATMUL	STRASSEN
6000 × 6000	**1.82795**	3.36921	**1.48523**	4.14287
8000 × 8000	**4.10491**	5.56739	**3.35258**	6.47299
12000 × 12000	**12.82721**	16.04158	**6.74356**	16.88393

Square matrices with 30% dispersion, in JUPITER and SATURN, using a single core (with MKL) and using GPU (with MAGMA)

For each problem size, the minimum execution time is highlighted in bold

Next, the use of the MAGMA library can be introduced to take advantage of the GPUs installed. The results of the experiments are also listed in Table 3. In view of these results, despite its greater computational capacity with respect to a multicore CPU, the cost of transferring the information from the CPU memory to the GPU memory makes the use of Strassen uncompetitive in terms of execution times when performing sequential executions of the groups that make up its model. This is particularly noticeable in SATURN, despite its faster GPU.

Now the simulator is used to see how the performance is affected by reapplying Strassen algorithm to each of the seven multiplications it performs. We will use the functionality of the simulator that allows us to create groups that solve a complete imported model instead of executing the functions of a routine. As seen in Fig. 9, the seven groups {MM1 … MM7} of the STRASSEN model that contained the RMM routine now include the STRASSEN model itself, representing a first level of recursion. We will call the newly generated model STRASSEN_R1.

Since the embedded model can be solved in turn following several paths, we must tell the simulator which one to run. In the present experiment, with a single-core system, group parallelism cannot be used. For this reason, Fig. 9 shows that the groups {MM1 … MM7} will execute in sequence all groups inherited from the imported Strassen model. In addition, the model STRASSEN_R1 will also be executed according to a sequential path, as we are considering a single-core system at all times (Fig. 10). Table 4 shows that adding a level of recursion to Strassen algorithm does not improve performance for the matrix sizes used in the experiments. This is because the advantage obtained in the multiplications using matrices whose sizes are half the original size does not manage to compensate for the overhead of adding the additional addition and subtraction operations that complete the Strassen algorithm that

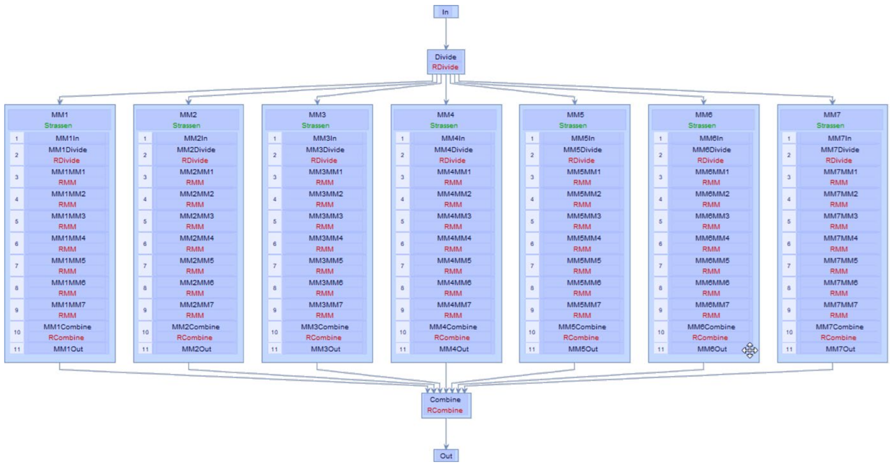

Fig. 9 Representation of `STRASSEN_R1` for matrix multiplication using the Strassen algorithm with one level of recursion

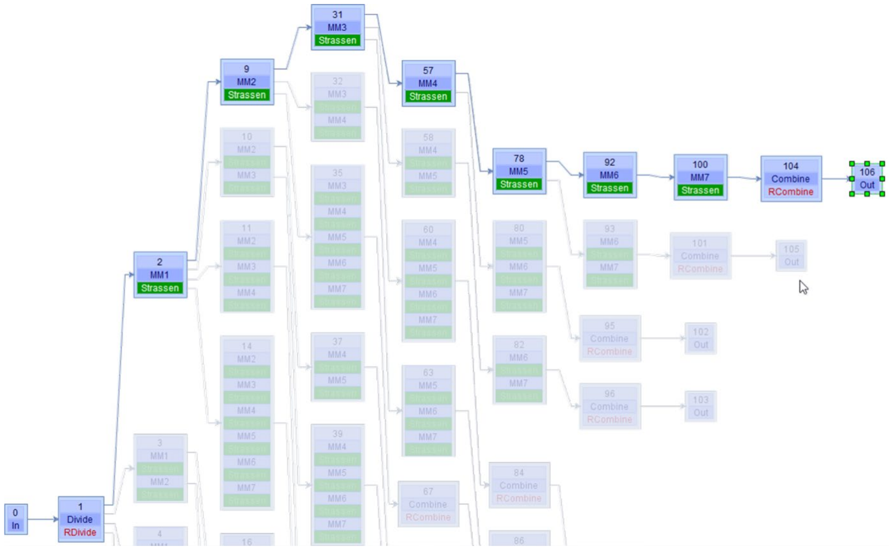

Fig. 10 Representation of the execution path of `STRASSEN_R1`, applicable to single-core systems

solves each group. However, with matrices of size 12000×12000, JUPITER starts to see an improvement in execution times. In this case, and due to the recursion, the matrices are 3000×3000, which is a size with which we have observed in the experiments a performance improvement between 6% and 8%, depending on the platform.

On the other hand, because MKL is a multi-threading library, it is possible to exploit its implicit parallelism and obtain better execution times, as we have seen above. Table 5 shows the results of the experiments with the new algorithmic

Table 4 Execution times (in seconds) obtained with the simulator in a traditional multiplication (MATMUL) and using Strassen algorithm without recursion (R0) and with one level of recursion (R1)

	JUPITER			SATURN		
		STRASSEN			STRASSEN	
nROWS	MATMUL	R0	R1	MATMUL	R0	R1
6000	22.56067	**21.20005**	22.179228	54.36038	**50.13080**	58.838894
8000	53.34177	**47.80797**	48.730221	130.18680	**114.81141**	122.493195
12000	179.10999	160.26036	**157.767395**	428.79211	**379.44492**	410.243134

Square matrices with 30% dispersion in JUPITER and SATURN, using a single core

For each problem size, the minimum execution time is highlighted in bold

Table 5 Execution times (in seconds) obtained with the simulator in a traditional multiplication and using Strassen algorithm without recursion

	SATURN					
	2 threads			3 threads		
nROWS×nCOLS	MATMUL	STRASSEN	%	MATMUL	STRASSEN	%
6000 × 6000	29.24644	**26.59713**	9.1	**19.22396**	19.84433	-3.2
8000 × 8000	70.83429	**66.17148**	6.6	**44.32339**	45.53305	-2.7
12000 × 12000	224.74800	**205.56395**	8.5	147.06793	**143.12148**	2.7
	JUPITER					
	2 threads			3 threads		
nROWS×nCOLS	MATMUL	STRASSEN	%	MATMUL	STRASSEN	%
6000 × 6000	11.31462	**10.67342**	5.7	7.69943	**7.40049**	3.9
8000 × 8000	26.73003	**24.68509**	7.7	18.07492	**17.12385**	5.3
12000 × 12000	89.84947	**81.57596**	9.2	70.71327	**66.61833**	5.8

Square matrices with 30% dispersion in SATURN and in JUPITER, generating 2 and 3 threads

For each problem size, the minimum execution time is highlighted in bold

parameters applied to the execution of the MATMUL and STRASSEN models. We observed that the reductions in execution times obtained with the Strassen algorithm compared to conventional multiplication in single-core systems (Table 3) are smaller as the number of threads is increasing. Assigning 3 threads to Strassen algorithm no longer offers any advantages, except for large sizes.

In any case, it is important to emphasize that the aim of this work is not to achieve an implementation of Strassen that improves the MKL one, but to show how the simulator can be a very useful tool in the analysis of the behaviour of the routines in different execution scenarios. In this way, the simulator can provide the user with an automatic decision framework for the best resolution option in each situation.

4 Conclusions and future work

This paper presents PARCSIM, a simulator for parallel software on heterogeneous platforms. This simulator offers the user the ability to analyse the execution time that their application would obtain depending on the space-time mapping of each of the tasks that make it up on the different computation units of the available hardware platform. An autotuning mode is included to help non-expert users in the selection of the simulation configuration. It has been used to determine satisfactory configurations of the parallelism parameters (number of OpenMP threads and of GPUs, and basic linear algebra library). This tool also offers polyalgorithmic resolution by automatically making the best decision between different software approaches. This work also shows the usefulness of this simulator for efficiently solving hierarchical problems by reusing the optimization information of the different subproblems. Experiments show the usefulness of the simulator with different configurations: computational systems composed of multicore CPU+multi-GPU and several basic linear algebra libraries together with their combinations with OpenMP through two-level parallelism.

The simulator and the autotuning engine can be adapted to other computational systems, for example, nodes including Xeon Phi coprocessors and clusters of multicore CPU+multicoprocessor. The simulator is being applied to other problems whose computation can be decomposed in DAGs. A comparative study of the performance that PARCSIM can offer compared to other similar tools is starting. Preliminary results have been obtained comparing it with Chameleon [?], a task-based dense linear algebra software that internally uses a runtime system to dynamically manage the execution of the different computational kernels on the existing hybrid computing units. Using a Strassen multiplication routine for a set of sizes from 500×500 to 12000×12000, in JUPITER, peak performance with Chameleon reaches about 210 GFlops, whereas with PARCSIM it reaches about 230 GFlops.

Acknowledgements Grant RTI2018-098156-B-C53 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe".

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

References

1. Cámara J, Cuenca J, García L, Giménez D (2014) Auto-tuned nested parallelism: a way to reduce the execution time of scientific software in NUMA systems. Parallel Comput 40(7):309–327
2. Cuenca J, García L, Giménez D, Herrera F (2017) Guided installation of basic linear algebra routines in a cluster with manycore components. Concurr Comput Pract Exp. https://doi.org/10.1002/cpe.4112
3. Cano J-C, Cuenca J, Giménez D, Saura-Sánchez M, Segado-Cabezos P (2018) A parallel simulator for multibody systems based on group equations. J Supercomput 75:1368–1381
4. Saura M, Celdrán AI, Dopico D, Cuadrado J (2014) Computational structural analysis of planar multibody systems with lower and higher kinematic pairs. Mech Mach Theory 71:79–92
5. Saura M, Segado P, Muñoz B, Dopico D (2015) Multibody kinematics. A topological formulation based on structural-group coordinates. In: ECCOMAS Thematic Conference on Multibody Dynamics, pp 88–99
6. Anderson E, Bai Z, Bischof C, Demmel J, Dongarra JJ, Croz JD, Grenbaum A, Hammarling S, McKenney A, Ostrouchov S, Sorensen D (1995) LAPACK user's guide. Society for Industrial and Applied Mathematics, Philadelphia
7. Intel MKL PARDISO (2018) https://software.intel.com/en-us/node/470282
8. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. Comput Sci Eng IEEE 5(1):46–55
9. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. Queue 6(2):40–53
10. Cámara J, Cuenca J, Giménez D (2020) Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. J Supercomput 76:9922–9941
11. Batory D (1992) The design and implementation of hierarchical software systems with reusable components. ACM Trans Softw Eng Methodol 1:355–398
12. Blackford LS, Choi J, Cleary A, D'cAzevedo E, Demmel J, Dhillon I, Dongarra JJ, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK user's guide. Society for Industrial and Applied Mathematics, Philadelphia
13. Dackland K, Kågström B (1996) A hierarchical approach for performance analysis of ScaLAPACK-based routines using the distributed linear algebra machine. In: Applied parallel computing, industrial computation and optimization, third international workshop, PARA96, Lyngby, Denmark, pp 186–195
14. Intel Corporation: Intel Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. Accedido 19 Apr, 2020
15. Intel Corporation: Intel MKL PARDISO-Parallel Direct Sparse Solver Interface. https://software.intel.com/en-us/node/470282. Accedido 16 May, 2020
16. Computational Mathematics Group at the STFC Rutherford Appleton Laboratory: HSL (2013) A collection of Fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk/. Accedido 16 May, 2020
17. Tomov S, Dongarra J, Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 36:232–240