Received: 13 March 2023 | Revised: 19 October 2023 | Accepted: 2 November 2023

DOI: 10.1002/cpe.7962

SPECIAL ISSUE PAPER

WILEY

Toward a new linpack-like benchmark for heterogeneous computing resources

Luisa Carracciuolo[1] | Valeria Mele[2] | Gianluca Sabella[3]

[1]Institute of Polymers, Composites and Biomaterials of the Italian National Research Council (IPCB-CNR), Pozzuoli, Napoli, Italy

[2]Department of Mathematics and Applications "R. Caccioppoli", University of Naples "Federico II", Napoli, Italy

[3]Department of Electrical Engineering and Information Technology, University of Naples "Federico II", Napoli, Italy

Correspondence
Luisa Carracciuolo, Institute of Polymers, Composites and Biomaterials of the Italian National Research Council (IPCB-CNR), Pozzuoli, Napoli, Italy.
Email: luisa.carracciuolo@cnr.it

Funding information
Università degli Studi di Napoli Federico II - CN_00000013 - CUP UNINA: E63C22000980007"

Summary

This work describes some first efforts to design a new Linpack-like benchmark useful to evaluate the performance of Heterogeneous Computing Resources. The benchmark is based on the Schur Complement reformulation of the solution of a linear equation system. Details about its implementation and evaluation, mainly in terms of performance scalability, are presented for a computing environment based on multi NVIDIA GP-GPUs nodes connected by an Infiniband network.

KEYWORDS

heterogeneous computing, HPC benchmarking, linear equation systems, schur complement

1 | INTRODUCTION

High-performance computers continue to increase in speed and capacity. Alongside these developments, architectures are becoming progressively more complex, with multi-socket, multi-core central processing units (CPUs), multiple graphics processing unit (GPUs) accelerators, and multiple network interfaces per node. This new complexity leaves existing software unable to use efficiently the increased processing power.[1]

In such a context, computer performance remains a complicated issue since it depends on many interrelated elements. These elements include the application, the algorithm, the size of the problem, the high-level language, the implementation, the human level of effort used to optimize the program, the compiler's ability to optimize, the age of the compiler, the operating system, the architecture of the computer, and the hardware characteristics.[2]

Since the '80s, the Linpack Benchmark and its evolution[3-5] was used by scientists worldwide to evaluate computer performance, particularly for innovative advanced-architecture machines. By performance, the Linpack Benchmark intends the number of billions of floating point operations per second, often measured in terms of gigaflops (Gflop/$s^{-}1$), executed during the solution of linear systems of the form

$$Ax = y, \text{ where } A \text{ is a dense } n \times n \text{ matrix.} \tag{1}$$

Large-scale problems in engineering and science often require the solution of the problem (1) where matrix A is sparse and not dense. The Krylov subspace iteration methods (KM) have led to a major change in how users deal with problems based on large, sparse, and nonsymmetric matrices. Krylov methods can solve problems too big for algorithms like factorization, as well as problems where the coefficient matrix A is only available as a function performing the matrix by vector multiplication operation. For all these reasons, the Krylov methods can be listed among the top 10 algorithms for Computing in Science and Engineering.[6] The performance of the Krylov methods is often dominated by communication,

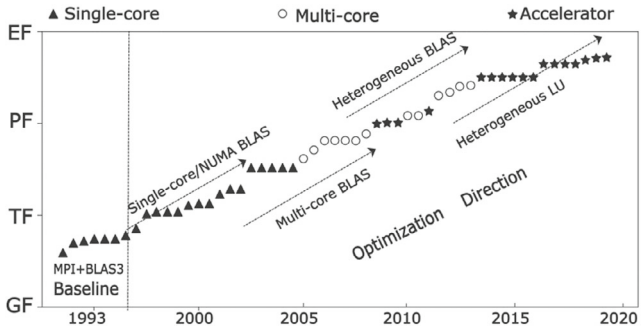


FIGURE 1 The evolution of the Linpack benchmark from 1993 to now. Figure Credits: Tan et al.[22]

which could become much more expensive than computation in terms of both throughput and energy consumption. As in Yamazaki et al.,[7] the term communication is used to include both horizontal data movement between parallel processing units and vertical data movement between memory hierarchy levels. In fact, in their original formulation, these methods are based on level 1 BLAS operations (i.e., vector products, products of a scalar by a vector, etc.)[1]. These operations have a low granularity and they fail to guarantee good performance, especially in high-performance computing contexts. In parallel computing, the term granularity of a task is a metric of the amount of work (or computation) performed,[8] and it refers to the ratio of computation time to communication time.

Considering the above, the Linpack Benchmark (based on level 3 BLAS operations) seems to have lost its relevance in guiding the community toward the development of benchmarks for HPC systems. Indeed, new tools such as the High Performance Conjugate Gradient (HPCG) benchmark,[9-11] which is based on implementations of KM, are now more representative of the computing patterns of real applications.

Nevertheless, paraphrasing the authors of the new benchmarks cited above,[9] we state *"Presently Linpack-like benchmarks remains tremendously valuable as a measure of historical trends, and as a stress test … Furthermore, it provides the HPC community with a valuable outreach tool, understandable to the outside world"*.

Furthermore, we cannot overlook the fact that for the Krylov methods (KM) solvers to efficiently utilize extreme-scale hardware, a lot of work has been dedicated to redesign both the Krylov methods algorithms and their implementations over the last three decades (e.g., see Yamazaki et al.,[7] Bai et al.,[12] Hoemmen,[13] Carracciuolo et al.,[14] Laccetti at al.[15]) to address challenges like extreme concurrency, complex memory hierarchies, costly data movement, and heterogeneous node architectures. All the redesign approaches base the algorithms on BLAS 2 and 3 operations and computational patterns more similar to the direct solvers of the problem (1).

That being said, in alignment with the authors of HPCG,[16] we believe that benchmarks for new computing patterns should be seen as a complement to the Linpack benchmark rather than a replacement.

The structure of this work is as follows: Section 2 provides a "State of the Art" regarding Linpack-like benchmarks for HPC and heterogeneous computing systems; Section 3 offers details about the usage of the Schur Complement in solving linear systems (1); Section 4 describes issues related to the initial implementation and evaluation, mainly in terms of performance scalability, of the benchmark based on the Schur reformulation; Section 5 summarizes the content of the present work and outlines some ideas for our future endeavors.

2 | RELATED WORKS AND MOTIVATION

In the field of Scientific Computing (SC), high-level benchmarks are designed to test the overall system performance, including the utilization of the CPU, memory, and hard drive, in conjunction with all available computing devices (i.e., GPUs). These tools should be capable of evaluating how well the computing environment can deliver the high performance required by SC applications.[17-20]

The tests performed by these tools are often used for both assessing the overall system performance and comparing the performance of different systems. For instance, the Linpack benchmark is a high-level benchmark used to evaluate the performance of computing systems in terms of their ability to process large-scale problems. The Linpack benchmark was originally developed in the 1970s[2,3] and is based on an algorithm for solving linear systems that use direct methods and whose computational complexity is the order of n^3 (where n is the dimension of the matrices involved).

Linpack has played a crucial role in the analysis of computing systems' performance for SC because it provides a way to compare systems of different architectures and sizes on the operation (that is the solution of linear systems with dense matrices) which is at the base of many algorithms of interest of SC. That allows scientists to evaluate, quite accurately, the efficiency of the use of computing systems, as evidenced by the use of the benchmark in the Top 500 ranking.[21] Indeed, the use of Linpack, and its evolutions, is linked to the creation of the Top 500 ranking, which lists the most powerful supercomputers in the world and uses the Linpack score as one of the main classification criteria. Figure 1 shows the

[1]The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. Some vendors make available their optimized implementation of the BLAS.

evolution of Linpack direction since the first stable version was released in 1993: hardware architecture evolution is the main driving source of the benchmark optimization.[22]

Over the years, Linpack has evolved into new versions such as the HPL (High Performance Linpack) benchmark.[4] HPL uses an algorithm to solve dense linear systems enabling more efficient and effective use of distributed memory and network connectivity on modern computing systems. The HPL algorithm was designed to be scalable and usable on a wide range of HPC systems. The algorithm implemented by the Linpack Benchmark is based on a block organization of the linear system matrix, where each block is processed separately, using level 3 BLAS operations. The HPL algorithm (like any Linpack-like benchmark) employs a technique called *"LU decomposition"* to solve linear systems.[23] This technique consists of factoring the matrix A as follows (by algorithms whose computational complexity is the order of n^3)

$$A = LU, \tag{2}$$

where L and U, respectively, are lower triangular and upper triangular matrices. This decomposition allows us to efficiently solve (1), especially if several systems with the same matrix A should be solved. Indeed, if the LU decomposition of a matrix A is already available, the linear system 1 can be solved by 1) the solution of $Lz = y$, followed by 2) the solution of $Ux = z$ (with a total computational complexity $O(2n^2)$).

HPL-MxP (High Performance Linpack Mixed-Precision benchmark, formerly known as HPL-AI) implements a new version of the HPL algorithm based on a mixed-precision approach. HPL-MxP is designed to test the performance of heterogeneous computing systems that use a combination of CPU and GPU.[5,24] The basic algorithm of HPL-MxP is similar to that of HPL and makes use of mixed-precision techniques to better exploit the characteristic of GPU resources. Mixed precision consists of using a lower floating-point precision for calculations on the GPU, being faster but less precise devices, than those based on CPUs. HPL-MxP is designed to be more efficient and scalable than the HPL algorithm. Indeed, in the intentions of its authors, it should more effectively use the system's memory and bandwidth and could be easily adapted to a wider range of system architectures. To maximize GPU resource usage, HPL-MxP uses a series of GPU-specific optimizations, such as the use of CUDA-version of BLAS library[25] or other numerical libraries such as `cuSOLVER`.[26] It also uses several efficient communication methods between the CPU and GPU to minimize data transfer time between the two processing units. The algorithm of HPL-MxP was formerly designed to test the performance of systems specifically planned for Artificial Intelligence and Machine Learning workloads, which often require a combination of high-performance CPU and GPU resources. HPL-MxP seeks to underline the link between the computational paradigms related to both the HPC and AI workloads based on machine learning (ML) and deep learning (DL): while traditional HPC focuses on simulation runs for modeling phenomena in a variety of scientific disciplines mostly requiring 64-bit precision, the ML/DL methods, that are at the basis of advances in AI, achieve the desired results at 32-bit or even lower ones. Performance of the HPL-MxP benchmark on the supercomputer Fugaku was the world's first achievement to exceed the wall of Exascale in a floating-point arithmetic benchmark.[27]

The CUDA-Aware HPL benchmark[28] implemented the first version of CUDA-based HPL for NVIDIA GPU clusters, it uses CUDA libraries to accelerate the HPL benchmark on heterogeneous clusters, where both CPUs and GPUs are used with minor or no modifications to the source code of HPL. A host library intercepts the calls to the most computationally intensive BLAS operations (i.e., the `DGEMM` and `DTRSM` procedures) that form the basis of the LU decomposition and executes each of them while distributing computation on both GPUs and CPU cores. In the CUDA-Aware HPL benchmark, computational load distribution of BLAS operations between CPU and GPU is automatically determined by the benchmark thanks to some metrics related to 1) the bandwidth for data transfer on PCI-e bus from/to host to/from the device and 2) the sustained performance of BLAS operations on the GPU/CPU. Since the sustainable bandwidth from the host to the device (and vice versa) plays a key role in the acceleration of a single DGEMM or DTRSM calls, in the CUDA-Aware HPL benchmark, the CUDA tool related to a fast transfer mode is exploited. Such a tool is enabled when page-locked memory (sometimes called pinned memory[29]) is used.

Since the publication date of Fatica,[28] a lot of work has been done for the Linpack optimization on the CPU-GPU heterogeneous systems for older systems technologies (see for example the "Related works" sections of Kim et al.[30] and Tan et al.[22]). New technologies present new challenges for programming heterogeneous systems. Indeed, the most important challenge is the widening gap between GPU computing speed, CPU computing speed, and data transfer speed (PCIe and inter-node network). From 2010 to 2019, one CPU's double-precision floating-point calculation speed increased from 30GFlops to 1TFlops while one GPU's speed from 250GFlops to 7TFlops,[22] forcing the software developer to deal with the new need to implement strategies that favor an appropriate load balancing to prevent the resources from being idle. Regarding the work done with the same goal in more recent technological contexts, the following two papers are worth mentioning.

In Kim et al.,[30] SnuHPL is described. It is an optimized HPL-based benchmark for modern clusters with inter-node heterogeneity (different GPUs on different nodes). A performance model is used to optimize SnuHPL execution generating information about the best data distribution for a given cluster configuration by considering computing power, memory capacity, and network performance. In the first intentions of the authors, SnuHPL should have been only an open-source HPL implementation optimized enough even for a cluster of modern homogeneous GPUs, just later did they make it a tool capable of adapting its executions, in terms of data and load distribution of work, to distributed memory systems with non-homogeneous GPUs. SnuHPL takes the data distribution generated by its data distribution generation framework that consists of a performance profiler, SnuHPL simulator, and a greedy heuristic generation algorithm. The SnuHPL Performance profiler samples various performance

parameters of the cluster from which the SnuHPL simulator determines the information about data distribution. However, if SnuHPL can adapt its execution on non-heterogeneous systems, it does not seem to be able to use all the system computational resources (CPUs and GPUs).

In Guangming Tan et al.,[22] a reformulation of the *LU* decomposition algorithm is described which better implements strategies for the overlapping of computations and communications phases where CPU-GPU data transfers use a PCIe-based bus. In particular, considering that the major part of such an algorithm proceeds multiple iterations of four consecutive steps—panel factorization (PF), panel broadcast (PB), row swapping (RS), trailing-matrix updating (TU)—on the matrix A in a blocking-by-blocking way, the authors implement and evaluate four different heterogeneous algorithms that organize such steps in diverse pipelines that try to optimize overlapping actions.

In addition to the two above-listed papers, it is also worth mentioning the recent effort spent on deploying rocHPL,[31] AMD's open-source implementation of the HPL benchmark targeting accelerated node architectures designed for exascale systems such as the Frontier supercomputer.[21] That implementation of the original HPL benchmark leverages the AMD GPU accelerators on the node via AMD's ROCm platform, runtime, and toolchains. The rocHPL code is written using the HIP programming language and are based on linear algebra routines highly optimized for AMD's latest discrete GPUs available from the rocBLAS math library.[32]

Together with Tan et al.,[22] we advocate that it is time for the community to release a new version of the Linpack benchmark for the CPU-GPU heterogeneous architecture. Therefore, in a context confirming the community's interest in the Linpack benchmark and its evolution, this work aims to lay the groundwork for a Linpack-like benchmark that can make the most of the heterogeneity of the computing systems with solutions that:

1. use both the CPUs and GPUs present on the individual nodes,
2. exploit all the most performing communications channels available,
3. employ CUDA-Aware (or more generally GPU-aware) messages passing library and innovative BLAS implementations (for example, the Software for Linear Algebra Targeting Exascale (SLATE) library[33]) or innovative approaches that can use a reformulation of the problem (1) no longer relying just on the LU decomposition (in the example, the HPL-MxP Mixed-Precision Benchmark[5]).

3 | THE *"SCHUR COMPLEMENT"*-BASED REFORMULATION OF A LINEAR EQUATION SYSTEM

The Schur complement is a fundamental and versatile tool in mathematical research and applications.[34] It can be considered a fundamental tool for the analysis and solution of the so-called *"Saddle Point Problem"* which arises in a wide variety of technical and scientific applications. For example, the ever-increasing popularity of mixed finite element methods in engineering fields such as fluid and solid mechanics has been a major source of saddle point systems. Another reason for this surge in interest is the extraordinary success of interior point algorithms in both linear and nonlinear optimization, which require at their core the solution of a sequence of systems in saddle point form.[35]

Suppose n_1, n_2 are non-negative integers, and suppose $A_{11}, A_{12}, A_{21}, A_{22}$ are respectively $n_1 \times n_1$, $n_1 \times n_2$, $n_2 \times n_1$, and $n_2 \times n_2$ matrices. Let

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \tag{3}$$

so that A is a $n \times n$ matrix, $n_1 \leq n$ and $n_2 = n - n_1$.

If A_{11} is invertible, the Schur complement of the block A_{11} of the matrix A is the $n_2 \times n_2$ matrix S defined by

$$S = A/A_{11} := A_{22} - A_{21}A_{11}^{-1}A_{12}. \tag{4}$$

The Schur complement arises naturally in solving a system of linear equations such as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}. \tag{5}$$

Assuming that the sub-matrix A_{11} is invertible, we can eliminate x_1 from the equations, as follows.

$$x_1 = A_{11}^{-1}(y_1 - A_{12}x_2). \tag{6}$$

Substituting this expression into the second equation yields

$$(A_{22} - A_{21}A_{11}^{-1}A_{21})x_2 = y_2 - A_{21}A_{11}^{-1}y_1. \tag{7}$$

Algorithm 1. The *"Schur complement"*-based algorithm for the linear system $Ax = y$ solution

procedure SCHURCOMPLEMENTSOLUTION(A, y, x)
  Input: A, y
  Output: x
  $L_{A_{11}}, U_{A_{11}} \leftarrow$ Compute LU factorization of A_{11} ▷ **Task n. 1**. $O\left(\frac{2}{3}n_1^3 - \frac{1}{2}n_1^2\right)$
  $z \leftarrow$ Solve $A_{11}z = y_1$ by mean of $L_{A_{11}}$ and $U_{A_{11}}$ ▷ **Task n. 2**. $O\left(2n_1^2\right)$. See Equation (8)
  $E \leftarrow$ Solve $A_{11}E = A_{12}$ by mean of $L_{A_{11}}$ and $U_{A_{11}}$ ▷ **Task n. 3**. $O\left(2n_1^2 n_2\right)$. See Equation (4)
  $w \leftarrow$ Compute the BLAS2 product $w = y_2 - A_{21}z$ ▷ **Task n. 4**. $O\left(n_2\left(n_1 + 1\right)\right)$. See Equation (8)
  $S \leftarrow$ Compute the BLAS3 product $S = A_{22} - A_{21}E$ ▷ **Task n. 5**. $O\left(n_2^2\left(n_1 + 1\right)\right)$. See Equation (4)
  $L_S, U_S \leftarrow$ Compute LU factorization of S ▷ **Task n. 6**. $O\left(\frac{2}{3}n_2^3 - \frac{1}{2}n_2^2\right)$
  $x_2 \leftarrow$ Solve $Sx_2 = w$ by mean of L_S and U_S ▷ **Task n. 7**. $O\left(2n_2^2\right)$. See Equation (8)
  $u \leftarrow$ Compute the BLAS2 product $u = y_1 - A_{12}x_2$ ▷ **Task n. 8**. $O\left(n_1\left(n_2 + 1\right)\right)$. See Equation (9)
  $x_1 \leftarrow$ Solve $A_{11}x_1 = u$ by mean of $L_{A_{11}}$ and $U_{A_{11}}$ ▷ **Task n. 9**. $O\left(2n_1^2\right)$. See Equation (9)
end procedure

We refer to this as the reduced equation obtained by eliminating x_1 from the original equation. The matrix appearing in the reduced equation is the Schur complement S of the block A_{11}:

Solving the reduced equation, we obtain

$$x_2 = S^{-1}\left(y_2 - A_{21}A_{11}^{-1}y_1\right). \tag{8}$$

Substituting this into the first equation yields

$$\begin{aligned} x_1 &= \left(A_{11}^{-1} + A_{11}^{-1}A_{12}S^{-1}A_{21}A_{11}^{-1}\right)y_1 - A_{11}^{-1}A_{12}S^{-1}y_2 \\ &= A_{11}^{-1}\left[\left(y_1 + A_{12}S^{-1}A_{21}A_{11}^{-1}y_1\right) - A_{12}S^{-1}y_2\right] \\ &= A_{11}^{-1}\left[y_1 - A_{12}S^{-1}\left(y_2 - A_{21}A_{11}^{-1}y_1\right)\right] \\ &= A_{11}^{-1}(y_1 - A_{12}x_2). \end{aligned} \tag{9}$$

From Equations (8) and (9) and from definition (4) follows that the solution of linear systems in Equation (1) can be solved by the procedure described in Algorithm 1 where A, x, and y are defined as in (5).

Concerning the Algorithm 1 we can observe that:

1. if A and A_{11} are nonsingular, then also $S = A/A_{11}$ is nonsingular (see Theorem 1.2 in Zhang[34]). Then, Algorithm 1 is applicable if A_{11} is nonsingular;
2. the Schur complement S is explicitly formed. Numerical instabilities may be a concern when forming S, especially when A_{11} is ill-conditioned (see section 5 in Benzi et al.[35]);
3. the approach used seems to be particularly attractive because of its block-based formulation because each operation can be computed on different computational devices (CPUs and GPUs) depending on the computational cost and also considering that some operations (differently from what happens in the case of algorithms based on the LU decomposition which is strongly recursive) are independent of the others;

where $n_2 = n - n_1$ and the $O(\cdot)$ represents the computational cost (i.e., the order of magnitude of the number of floating point operations).

4 | THE BENCHMARK IMPLEMENTATION AND EVALUATION

4.1 | The Benchmark implementation details

This section describes the strategies used in the implementation of Algorithm 1. These strategies, which are at the basis of the choice of which computing device (CPUs or GPUs) to use in the allocation of each task, must take into account:

1. the dependency between tasks,
2. the computational cost of each task,

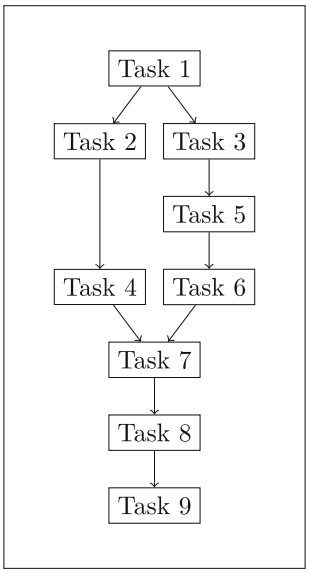


FIGURE 2 Tasks dependency diagram.

TABLE 1 The mapping of tasks in the Algorithm 1.

	T1	T2	T3	T4	T5	T6	T7	T8	T9
CPU									
GPU									

3. the balancing of the total computational load on computing devices,
4. the numbers and dimensions of communications needed to guarantee the availability of the task input data on the allocated computational devices,
5. to preserve, as far as possible, the locality of the data, that is, data retention on allocation devices.

Regarding point 1 of the above list, Figure 2 depicts the Dependency Diagram of the tasks listed in Algorithm 1 from which we can observe that some tasks can be considered independent from the others and suggesting what of them can be executed, eventually in a concurrent way, on the different computational resources (CPUs and GPUs) of the systems.

Under the assumption that $n_1 < n_2$, and to account for their dependencies, computational costs, and data locality (see points 1, 2, and 5), the mapping of the tasks described in Table 1 is considered: tasks with higher computational costs are mapped on GPU also in consideration of the required data exchange between CPUs and GPUs (see point 4).

If the mapping of the tasks described in Table 1 is adopted, the two computing parts of the systems (CPUs and GPUs) share the total computational cost of the Algorithm 1 $\boldsymbol{CompCost}^{Schur} = \boldsymbol{CompCost}^{Schur}{}_{CPU} + \boldsymbol{CompCost}^{Schur}{}_{GPU}$ where:

$$\boldsymbol{CompCost}^{Schur}{}_{CPU} = O\left(\underbrace{\frac{2}{3}n_1^3 - \frac{1}{2}n_1^2}_{T1} + \underbrace{2n_1^2}_{T2} + \underbrace{n_2(n_1+1)}_{T4} + \underbrace{2n_1^2}_{T9}\right) = O\left(n_1^3\right), \tag{10}$$

$$\boldsymbol{CompCost}^{Schur}{}_{GPU} = O\left(\underbrace{2n_1^2 n_2}_{T3} + \underbrace{n_2^2(n_1+1)}_{T5} + \underbrace{\frac{2}{3}n_2^3 - \frac{1}{2}n_2^2}_{T6} + \underbrace{2n_2^2}_{T7} + \underbrace{n_1(n_2+1)}_{T8}\right) = O\left(n_2\left(n_2^2 + n_1^2\right)\right). \tag{11}$$

To evaluate how the values of n_1 and n_2 could condition the balance of the computational load between CPU and GPU (see point 3), we have to consider that balance can be expressed as

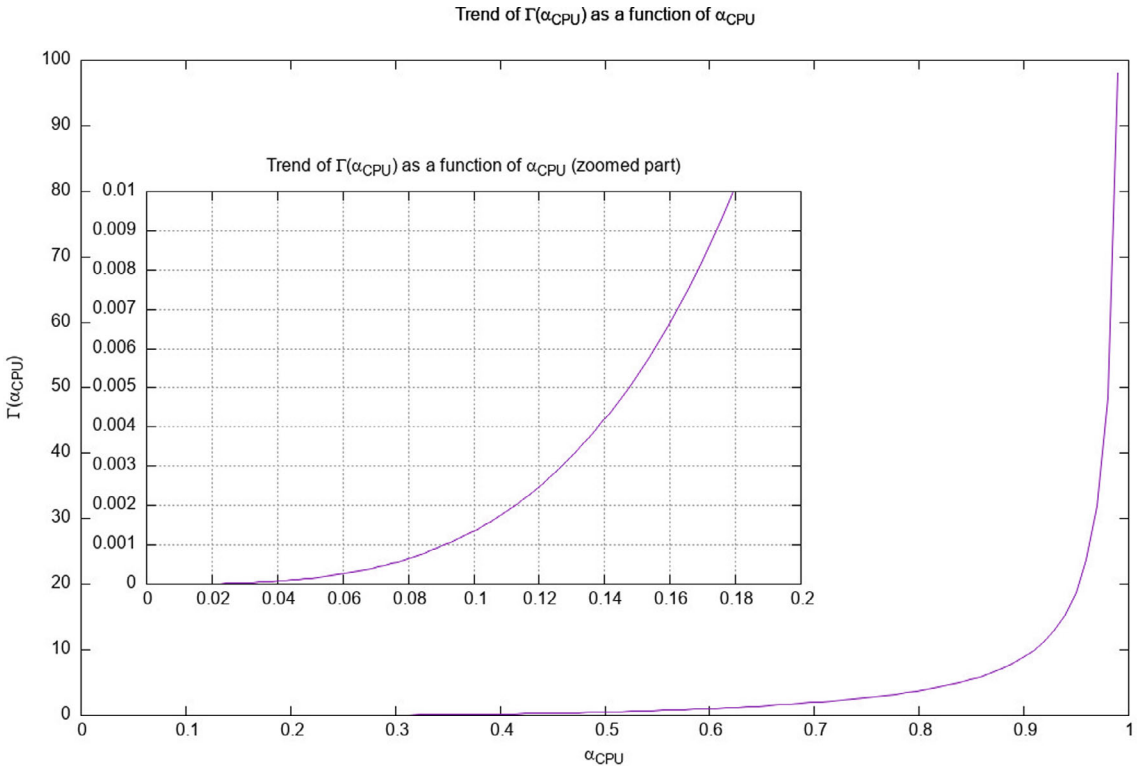


FIGURE 3 The trend of the function $\Gamma(\alpha_{CPU})$ in Equation (15).

$$\underbrace{\frac{\boldsymbol{CompCost}^{\boldsymbol{Schur}}_{\boldsymbol{CPU}}}{\boldsymbol{NCores}_{\boldsymbol{CPU}}\boldsymbol{NClock}_{\boldsymbol{CPU}}}}_{\boldsymbol{CompTime}^{\boldsymbol{Schur}}_{\boldsymbol{CPU}}} \approx \underbrace{\frac{\boldsymbol{CompCost}^{\boldsymbol{Schur}}_{\boldsymbol{GPU}}}{\boldsymbol{NCores}_{\boldsymbol{GPU}}\boldsymbol{NClock}_{\boldsymbol{GPU}}}}_{\boldsymbol{CompTime}^{\boldsymbol{Schur}}_{\boldsymbol{GPU}}}, \tag{12}$$

where $\boldsymbol{NCores}_{CPU}$ and $\boldsymbol{NCores}_{GPU}$ represent respectively the number of cores of CPUs and GPUs, and where $\boldsymbol{NClock}_{CPU}$ and $\boldsymbol{NClock}_{GPU}$ are the number of theoretical flops per second executed respectively by one of the CPUs and GPUs core.

Using (10) and (11), the relation (12) is valid if and only if

$$O\left(\frac{n_1^3}{n_2\left(n_2^2+n_1^2\right)}\right) \approx O\left(\frac{\boldsymbol{NCores}_{CPU}\boldsymbol{NClock}_{CPU}}{\boldsymbol{NCores}_{GPU}\boldsymbol{NClock}_{GPU}}\right). \tag{13}$$

If n_1 is defined in terms of a fraction of n, that is,

$$n_1 = \alpha_{CPU} n, \tag{14}$$

then Equation (13), can be rewritten as:

$$O\left(\underbrace{\frac{\alpha_{CPU}^3}{(1-\alpha_{CPU})\left[(1-\alpha_{CPU})^2+\alpha_{CPU}^2\right]}}_{\Gamma(\alpha_{CPU})}\right) \approx O\left(\underbrace{\frac{\boldsymbol{NCores}_{CPU}\boldsymbol{NClock}_{CPU}}{\boldsymbol{NCores}_{GPU}\boldsymbol{NClock}_{GPU}}}_{F_{Sys}(\boldsymbol{NCores}_*,\boldsymbol{NClock}_*)}\right). \tag{15}$$

so, computational load balance depends just from α_{CPU}, where $0 < \alpha_{CPU} < 1$, and from computing system features. In Figure 3 the trend of the function $\Gamma(\alpha_{CPU})$ in Equation (15) is shown.

It is noteworthy that the relation 15 can be used to determine, starting from the characteristics of the computing resources and the problem dimension n, the value of n_1 which should be able to guarantee the balancing of the computational load (see Section 4.3 for an example of usage).

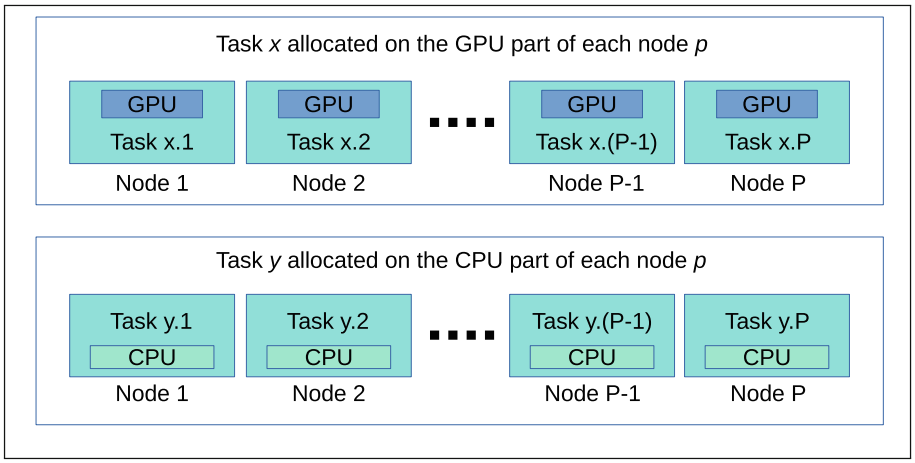


FIGURE 4 Tasks allocation on a distributed memory computing infrastructure.

We note that the execution of the BLAS operations related to each task can be performed by versions of the BLAS library that are optimized for each computing infrastructure. For example, if a cluster of computing nodes equipped with some GPUs is used, each task can be performed by a Distributed Memory based BLAS library (i.e., ScaLAPACK,[36] SLATE,[33] PETSc,[37] etc.): in such case, the allocation of the tasks to the CPUs or GPUs devices is intended that *"the local part Task x.p"* of Task *x* is assigned, that is, on each node, is set the computation on the local data of the same task (see Figure 4).

To define the *amount of communications* needed by tasks allocations (see point 4 above), we note that:

- one communication is needed, at the end of task 1, to send to the GPU (task 3) the factorization of A_{11}. The order of the data amount to be sent is $O(n_1^2)$),
- one communication is needed, at the end of task 4, to send to the GPU (task 7) the vector *w*. The order of the data amount to be sent is $O(n_2)$),
- one communication is needed, at the end of task 8, to send to the CPU (task 9) the vector *u*. The order of the data amount to be sent is $O(n_1)$),

So, if input data is already available on computing devices, just three communications are performed (between GPU and CPU) for a total of $\boldsymbol{CommCost}^{\boldsymbol{Schur}}_{\boldsymbol{CPU\leftrightarrow GPU}}$ data transferred where

$$\boldsymbol{CommCost}^{\boldsymbol{Schur}}_{\boldsymbol{CPU\leftrightarrow GPU}} = O(n_1^2 + n). \tag{16}$$

However, it's important to observe that other communications can be hidden in the implementations of the BLAS operations of each task: for example, in the BLAS implementations for distributed memory systems where the exchange of messages takes place by not CUDA-Aware libraries (see next subsection for a definition of CUDA-Aware Message Passing Library).

The time $\boldsymbol{CommTime}^{\boldsymbol{Schur}}{}_{\boldsymbol{CPU\leftrightarrow GPU}}$ spent during data transfer between GPU and CPU could be modeled as:

$$\boldsymbol{CommTime}^{\boldsymbol{Schur}}_{\boldsymbol{CPU\leftrightarrow GPU}} = \frac{\boldsymbol{CommCost}^{\boldsymbol{Schur}}_{\boldsymbol{CPU\leftrightarrow GPU}}}{\boldsymbol{BW}_{\boldsymbol{CPU\leftrightarrow GPU}}}, \tag{17}$$

where $\boldsymbol{BW}_{\boldsymbol{CPU\leftrightarrow GPU}}$ represents the bandwidth (i.e., the amount of data transferred per second) of the data transfer channel between CPU and GPU.

To define a model of performance for Algorithm 1 able to give indications about its performance in terms of the number of floating point operations per second, we propose the *Theoretical Sustainable Performance* $TSP(n, \alpha_{CPU}, \boldsymbol{NCores}_*, \boldsymbol{NClock}_*)$ metric defined as:

$$TSP(n, \alpha_{CPU}, \boldsymbol{NCores}_*, \boldsymbol{NClock}_*) = \frac{\overbrace{\boldsymbol{CompCost}^{\boldsymbol{Schur}}}^{\text{Estimated number of floating point operations}}}{\underbrace{\boldsymbol{CompTime}^{\boldsymbol{Schur}} + \boldsymbol{CommTime}^{\boldsymbol{Schur}}_{\boldsymbol{CPU\leftrightarrow GPU}}}_{\text{Estimated execution time}}}, \tag{18}$$

where $\boldsymbol{CompTime}^{\boldsymbol{Schur}} = \boldsymbol{CompTime}^{\boldsymbol{Schur}}_{\boldsymbol{CPU}} + \boldsymbol{CompTime}^{\boldsymbol{Schur}}_{\boldsymbol{GPU}}$. Equation (18) can be used (see Section 4.3 for an example of use) to "predict" performance starting from the characteristics of the computing resources $\boldsymbol{NCores}_*, \boldsymbol{NClock}_*$, from the problem dimension *n* and the value of n_1 (defined by α_{CPU}).

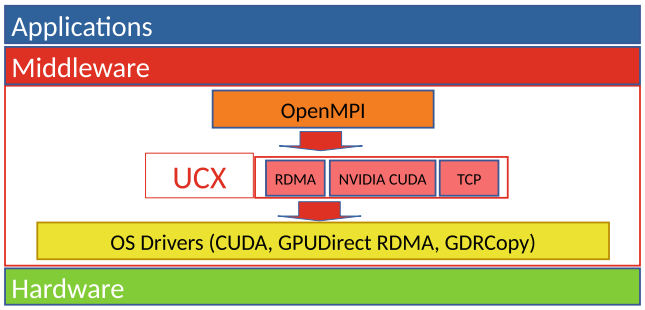


FIGURE 5 The layered architecture of computing resource.

4.2 | The Benchmark evaluation details

In this sub-section, we describe the computing environment used during evaluation tests of Algorithm 1 implementation.

We utilized a heterogeneous computational resource[38,39] equipped with 128 GPUs and approximately 1600 physical cores distributed across 32 nodes. These nodes are interconnected using InfiniBand and NVLink technologies.

The architecture of the computing resources can be depicted as a set of multiple layers (Figure 5). The highest layer of the architecture consists of the application layer which is exposed to users. The lowest one consists of hardware resources and comprises 32 computing nodes. In particular, it provides 1) 128 NVIDIA Volta GPUs and about 1600 physical cores (from Intel Gen 2 Xeon Gold CPUs) distributed on 32 nodes whose connections are based on InfiniBand[40] and NVLink2[41] technologies. The efficient use of cluster technologies is made possible by a software layer interposed between the lowest and the highest levels, namely the middle-ware, which is based on a combination of the following technologies:

1. OpenFabrics Enterprise Distribution (OFED)[42] that makes available drivers and libraries needed by the Mellanox InfiniBand network cards.
2. CUDA Toolkit[43] that makes available drivers, libraries, and development environments enabling NVIDIA GP-GPU usage.
3. *"MPI-CUDA aware"*[44] implementation of OpenMPI[45] through the UCX open-source framework.[46]

Bandwidth and latency in message exchange among processes are critical factors that hinder the full utilization of GPU potential.

In addressing this challenge, NVIDIA has introduced two important technologies: CUDA Inter-Process Copy (IPC)[47] and GPUDirect Remote Direct Memory Access (RDMA).[48] These technologies are designed for intra- and inter-node GPU process communications and are particularly valuable for InfiniBand-based clusters. Additionally, for optimizing inter-node GPU-to-GPU communications for small messages, NVIDIA offers NVIDIA gdrcopy.[49] To integrate these technologies with communication libraries (i.e., OpenMPI), we used the UCX open-source framework. UCX is a communication framework optimized for modern, high-bandwidth, low-latency networks. It exposes a set of abstract communication primitives that automatically choose the best available hardware resources. Supported technologies include RDMA (both InfiniBand and RoCE), TCP, GPU, shared memory, and atomic network operations.

Table 2 shows the hardware and software features of the cluster nodes.

All the BLAS operations listed in Algorithm 1 are using the SLATE library procedures. The SLATE (Software for Linear Algebra Targeting Exascale) library is actively under development to provide essential capabilities for dense linear algebra on current and future distributed high-performance systems. This includes systems based on both CPU+GPU or just on CPU. SLATE will provide coverage of existing ScaLAPACK functionality, including the parallel BLAS and the solution of the linear systems using LU and Cholesky. In this respect, it will serve as a replacement for ScaLAPACK, which after two decades of operation, cannot adequately be retrofitted for modern accelerated architectures. SLATE uses modern techniques such as communication-avoiding algorithms, look-ahead panels to overlap communication and computation, and task-based scheduling, along with a modern C++ framework.[33]

While the BLAS operations provided by SLATE can be utilized on Distributed Memory computing platforms, they lack strategies to fully exploit the heterogeneous capabilities of nodes by simultaneously leveraging both CPUs and GPUs. The SLATE procedures offer a CPU execution mode that can be specified using the macro called `execution target`: if such target is defined as `slate::Target::HostTask`, the execution will happen on the CPUs (cores) using OpenMP tasks[50] allowing the exploitation of the multicore architecture of modern computing nodes.

4.3 | The benchmark evaluation results

In this sub-section, we illustrate some results about the evaluation of the implementation of Algorithm 1 in the software module `schur_solve`. The developed module `schur_solve` uses the modules offered by the SLATE library and uses double precision floating point number (i.e., *sizeof*(*float*) = 8).

TABLE 2 Hardware and Software specs of cluster nodes. Please note that the value of data bandwidth was obtained by the CUDA `bandwidthTest` utility.

Number of nodes	32
InfiniBand NIC type per node	Mellanox MT28908 [ConnectX-6]
Processor type per node	Intel Xeon Gold 6240R CPU@2.40GHz
Number of cores per node ($\boldsymbol{NCores}_{CPU}^{PerNode}$)	48
Number of GPUs per node	4
OS	Linux CentOS 7
InfiniBand Drive	NVIDIA MLNX_OFED 5.3-1.0.0.1
Infiniband Firmware	20.27.6106
CUDA driver & Runtime version	11.1
CUDA Capability Major/Minor version number	7.0
NVIDIA GPUDirect-RDMA	1.1
NVIDIA gdrcopy	2.2
UCX	1.10.0
MPI Distribution	Open MPI 4.1.0rc5
SLATE	Version 2021.05.02
BLAS Distribution	Intel MKL 2021.1.1
GPU Device Type	Tesla V100-SXM2-32GB
The total amount of global memory	32510 MBytes (34089730048 bytes)
(80) Multiprocessors, (64) CUDA Cores/MP	5120 CUDA Cores
GPU Max Clock rate	1530 MHz (1.53 GHz)
UCX Configurations	`UCX_NET_DEVICES=mlx5_0:1`
	`UCX_IB_GPU_DIRECT_RDMA=yes`
	`UCX_TLS=cuda_ipc,cuda_copy,ib`
GPU Theoretical Peak Performance per Node	
$\boldsymbol{PP}_{GPU} = \boldsymbol{NClock}_{GPU} * \boldsymbol{NCores}_{GPU}^{PerNode}$	$\underbrace{1.53}_{\boldsymbol{NClock}_{GPU}} * \underbrace{4 * 5120}_{\boldsymbol{NCores}_{GPU}^{PerNode}}$ GFlops $\approx$ 31.2 TFlops
CPU Theoretical Peak Performance per Node	
$\boldsymbol{PP}_{CPU} = \boldsymbol{NClock}_{CPU} * \boldsymbol{NCores}_{CPU}^{PerNode}$	$\underbrace{2.40}_{\boldsymbol{NClock}_{CPU}} * \underbrace{48}_{\boldsymbol{NCores}_{CPU}^{PerNode}}$ GFlops $\approx$ 0.12 TFlops
The bandwidth of the communication channel between the CPU and GPU	$\frac{12.5\ GB/s}{\text{sizeof(float)}}$
(number of floats per second) $\boldsymbol{BW}_{CPU \leftrightarrow GPU}$	
Data bandwidth was obtained by the CUDA `bandwidthTest` utility	

As a term of comparison, we also show the results related to the execution of the SLATE module `slate::gesv` which performs the solution of linear system 1 exclusively using CPU or GPU and whose computational cost is $\boldsymbol{CC}^{\boldsymbol{LUBased}} = O\left(\frac{2}{3}n^3 - \frac{1}{2}n^2\right)$.

In Figures 6–8, we show the results of module `schur_solve` executed on some nodes of the described cluster: the number of total MPI tasks is $4P$ where P is the number of involved nodes, the number $N_{OpenMPTasks}$ of the OpenMP tasks used for CPU execution of SLATE procedures is fixed to be $N_{OpenMPTasks} = 12$. Then the value for $F_{Sys}(\boldsymbol{NCores}_*, \boldsymbol{NClock}_*)$ in (15) is:

$$F_{Sys}(\boldsymbol{NCores}_*, \boldsymbol{NClock}_*) = \frac{\overbrace{4 * P * 12}^{\boldsymbol{NCores}_{CPU}} * 2.48}{\underbrace{4 * P * 5120}_{\boldsymbol{NCores}_{GPU}} * 1.53} = \frac{12 * 2.48}{5120 * 1.53}. \tag{19}$$

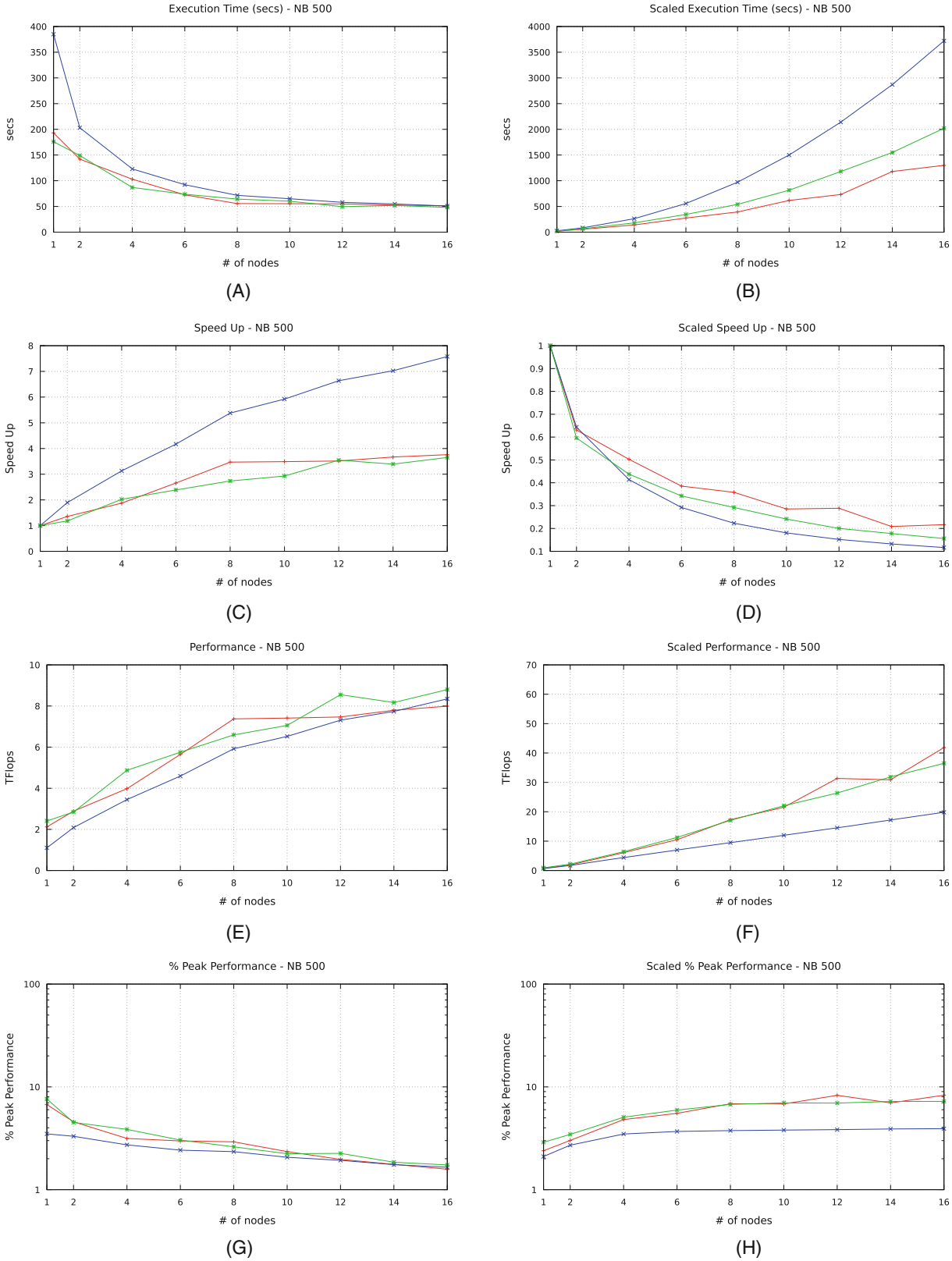


FIGURE 6 Tests results, $NB = 500$: The Execution Time $T(P, n)$ (A), The Scaled Execution Time $T(P, Pn)$ (B), Speed-Up $S(P, n)$ (C), Scaled Speed-Up $SS(P, n)$ (D), the Sustained Performance $SP(P, n)$ (E), the Scaled Sustained Performance $SP(P, Pn)$ (F), the fraction of Peak Performance $SPF(P, n)$ (G), and the Scaled fraction of Peak Performance $SPF(P, Pn)$ (H). Red, green, and blue lines respectively represent the *Implementation of the Schur-Based algorithm*, `slate::gesv` *on GPU*, and `slate::gesv` *on CPU* test results.

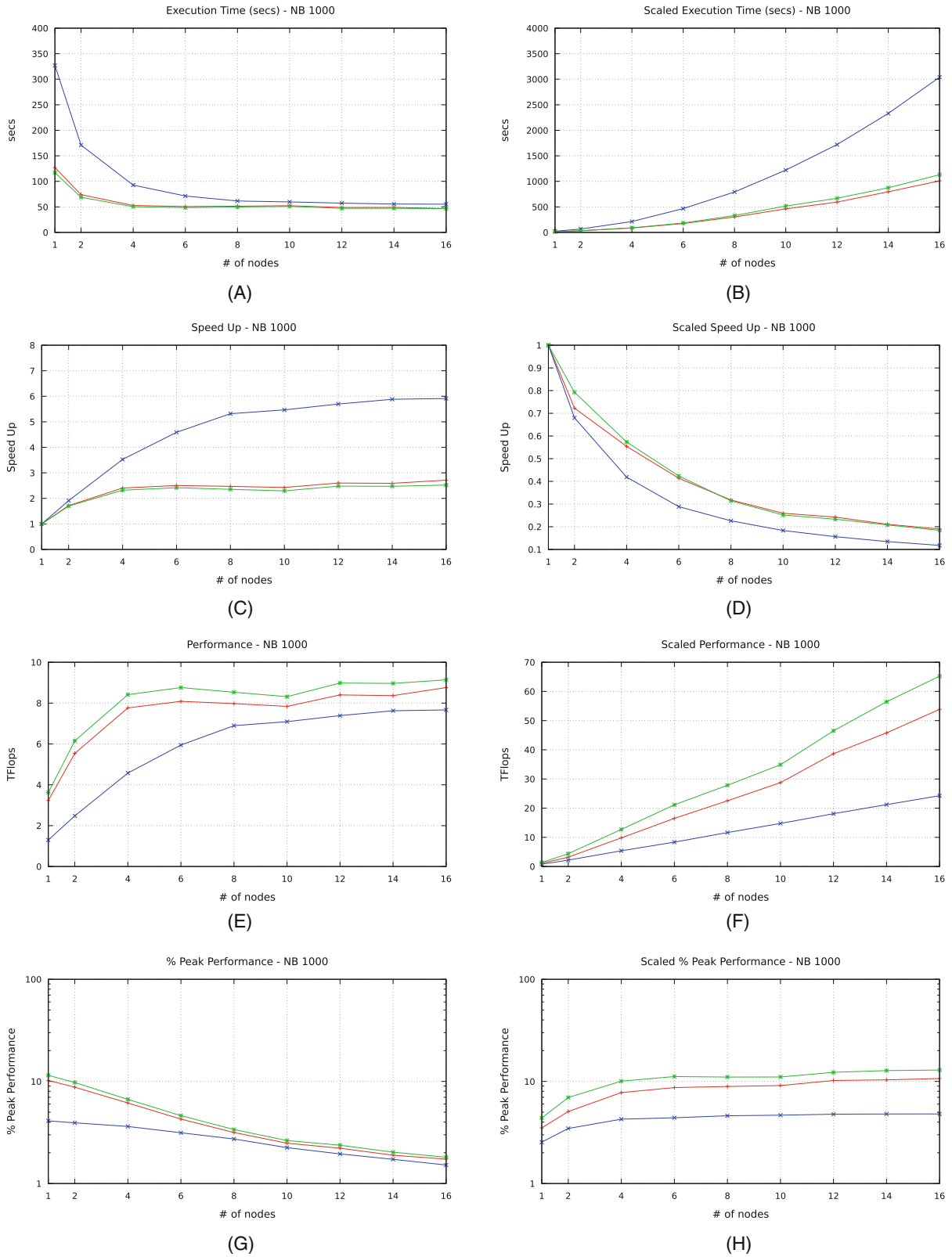


FIGURE 7 Tests results, $NB = 1000$: The Execution Time $T(P, n)$ (A), The Scaled Execution Time $T(P, Pn)$ (B), Speed-Up $S(P, n)$ (C), Scaled Speed-Up $SS(P, n)$ (D), the Sustained Performance $SP(P, n)$ (E), the Scaled Sustained Performance $SP(P, Pn)$ (F), the fraction of Peak Performance $SPF(P, n)$ (G), and the Scaled fraction of Peak Performance $SPF(P, Pn)$ (H). Red, green, and blue lines respectively represent the *Implementation of the Schur-Based algorithm*, `slate::gesv` *on GPU*, and `slate::gesv` *on CPU* test results.

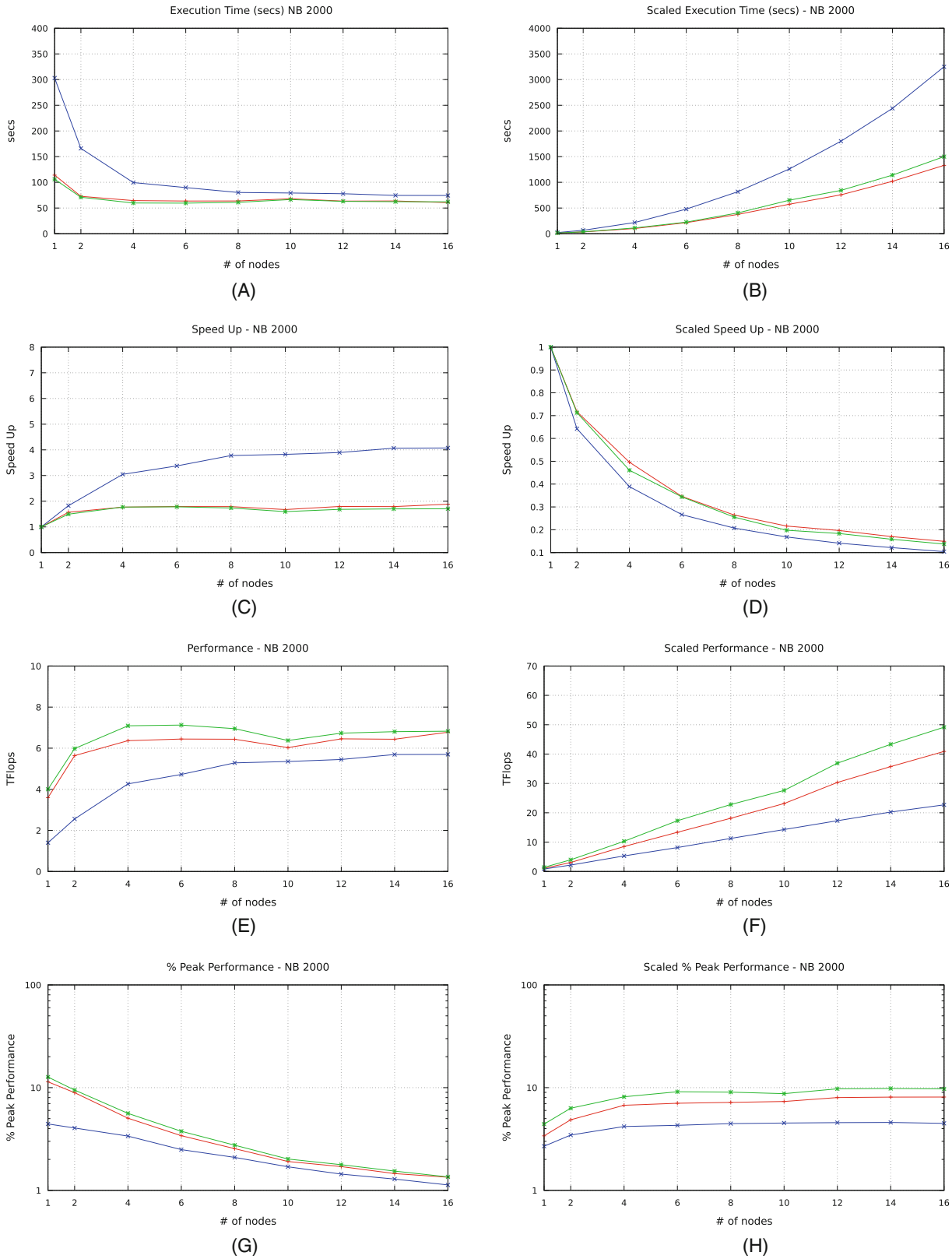


FIGURE 8 Tests results, $NB = 2000$: The Execution Time $T(P, n)$ (A), The Scaled Execution Time $T(P, Pn)$ (B), Speed-Up $S(P, n)$ (C), Scaled Speed-Up $SS(P, n)$ (D), the Sustained Performance $SP(P, n)$ (E), the Scaled Sustained Performance $SP(P, Pn)$ (F), the fraction of Peak Performance $SPF(P, n)$ (G), and the Scaled fraction of Peak Performance $SPF(P, Pn)$ (H). Red, green, and blue lines respectively represent the *Implementation of the Schur-Based algorithm*, `slate::gesv` *on GPU*, and `slate::gesv` *on CPU* test results.

It follows, from (15) and (19), that to get computational load balance, a good choice for α_{CPU} (and then of n_1) should be such that $O(\Gamma(\alpha_{CPU})) = 10^{-3}$. From Figure 3, we can observe that it happens when $0.06 < \alpha_{CPU} < 0.18$ (see the zoomed part of the plot).

The tests, which have the main aim to verify the scalability of Algorithm 1 implementation, are performed using different values for SLATE block dimension *NB* which is used by SLATE to distribute matrices over a distributed memory computational resource.[51]

Plots in Figures 6–8 show:

$T(P, n)$: The execution time (in seconds) of the module `schur_solve` as a function of the number P of nodes for some values of n;
$S(P, n)$: The Speed-Up of the execution as a function of the number P of nodes for some values of n. So,

$$S(P, n) = \frac{T(1, n)}{T(P, n)};$$

$SS(P, n)$: The Scaled Speed-Up of the execution as a function of the number P of nodes for some values of n. So,

$$SS(P, n) = \frac{PT(1, n)}{T(P, Pn)};$$

$SP(P, n)$: The Sustained Performance (expressed in TeraFLOPS) obtained during the execution as a function of the number P of nodes for some values of n. It represents the number of Floating Point operations $CC(n)$ executed by an algorithm in a time range, that is

$$SP(P, n) = \frac{CC(n)}{T(P, n)};$$

$SPF(P, n)$: The fraction of Peak Performance obtained during the execution as a function of the number P of nodes for some values of n. So,

$$SPF(P, n) = \frac{SP(P, n)}{PP(P)}$$

where $PP(P)$ is the Peak Performance of P nodes when for each node all four GPU devices and all the CPU cores are considered where

$$PP(P) = (\boldsymbol{PP}_{GPU} + \boldsymbol{PP}_{CPU})P.$$

See Table 2 for the considered values of $\boldsymbol{PP}_{CPU}$ and $\boldsymbol{PP}_{GPU}$.

The following values for n are considered: $n = 86{,}000$ (strong scalability tests[52]), $n = 30{,}000 * P$ (weak scalability[52] tests). For all the tests, the values $n_1 = 2000$ (strong scalability tests) and $n_1 = 2000 * P$ (weak scalability tests) are considered in line with the considerations previously made about better choices for α_{CPU}.

From Figures 6–8 we can note that:

- weak scalability tests seem to confirm behavior already described in the ancestor of SLATE named DPLASMA[53] (for example, lines in Figures $*$-(F) show a similar trend to those in fig. 7 in Bosilca et al.[53]);
- the role of the blocking factor *NB* is decisive in obtaining performance. The most appropriate value of this parameter (in this specific case $NB = 1000$) makes it possible to reduce the time for solving the problem by up to 50% (i.e., see Figures $*$-(A) and $*$-(B));
- the implementation of the Schur-Based algorithm (the software module `schur_solve`) seems to be less sensitive to the variations of values for parameter *NB*;
- times to solution and speed-up values of the software module `schur_solve` are similar (and in the case of $MB = 500$ better) than those obtained with the module `slate::gesv` executed on GPUs;
- in case the metrics relating to the number of operations per unit of time are considered, the module `schur_solve` does not get the same performance as the module `slate::gesv` executed on the GPU. This could be because the number of operations $\boldsymbol{CC}^{\boldsymbol{LUBased}}$ performed by the former could be different than that of operations $\boldsymbol{CC}^{\boldsymbol{Schur}}$ performed by the latter.

About the fraction of Peak Performance, please note that the performance percentages reported by the Top500 are related to the performance measurement $R_{\max}$ which is obtained using the largest problem size $n_{\max}$ fitting memory of the computing system.[2]

Intending to use the proposed new benchmark implementation similarly to the one already used to draw up the Top500, we have identified, for different α_{CPU} the value of the largest problem size $n_{\max}(1)$ on one nodes, then the value of $n_{\max}(P)$ is obtained scaling $n_{\max}(1)$ by the following rule $n_{\max}(P) = n_{\max}(1)\sqrt{P}$.

TABLE 3 Execution time $T(P, n)$ and Sustained Performance $SP(P, n)$ for some values of α_{CPU}. The value of $n_{\max}(P)$ is obtained by scaling the value of the largest problem size $n_{\max}(1)$ on $P = 1$ node by the following rule $n_{\max}(P) = n_{\max}(1)\sqrt{P}$.

		$\alpha_{CPU} = 0.10$				$\alpha_{CPU} = 0.20$				$\alpha_{CPU} = 0.40$			
P	$PP(P)$	$n_{\max}(P)$	n_1	$T(P, n)$	$SP(P, n)$	$n_{\max}(P)$	n_1	$T(P, n)$	$SP(P, n)$	$n_{\max}(P)$	n_1	$T(P, n)$	$SP(P, n)$
1	31.56	130,000	13,000	845	1.52	150,000	30,000	1100	1.65	181,000	81,000	1730	1.82
4	126.26	260,000	26,000	1940	5.31	300,000	60,000	2490	5.84	362,000	162,000	3220	7.81
16	505.04	520,000	52,000	3950	20.85	600,000	120,000	4870	23.89	725,000	325,000	6670	30.31

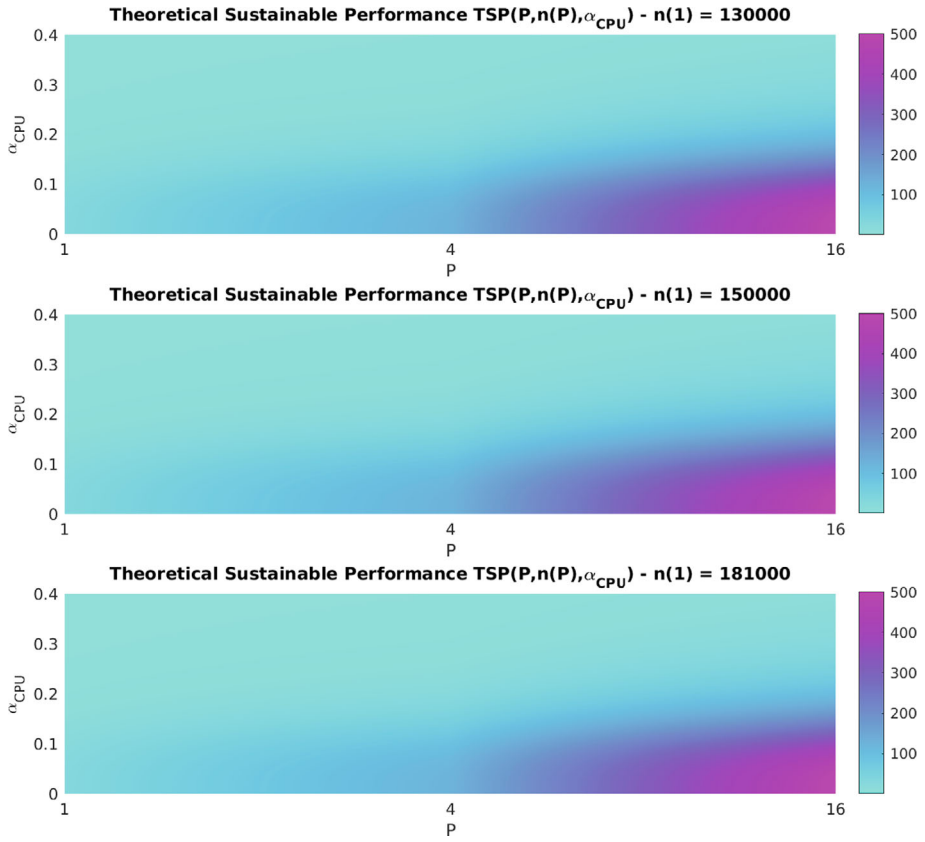


FIGURE 9 *"Heatmap"* of the Theoretical Sustainable Performance $TSP(P, n(P), \alpha_{CPU})$ as a function of α_{CPU} and P for different values of the problem dimension on one node $n(1)$. The value of problem dimension $n(P)$ on P nodes is scaled according to the following rule $n(P) = n(1)\sqrt{P}$.

In Table 3 the Execution Time $T(P, n_{\max}(P))$ and the Sustained Performance $SP(P, n_{\max}(P))$ are shown for the values of $n_{\max}(P)$ already identified. The results listed in Table 3 confirm the very low fraction of Peak Performance obtained using the described implementation of Algorithm 1. Yet Algorithm 1 should be able to achieve performances very close to the peak ones (see Figures 9 and 10). Such figures show respectively

1. the Theoretical Sustainable Performance $TSP(P, n(P), \alpha_{CPU})$ (see Equation (18) that was evaluated using the computing system features listed in Table 2),
2. and the Theoretical Peak Performance Fraction $TSPF(P, n(P), \alpha_{CPU}) = \frac{TSP(P,n(P),\alpha_{CPU})}{PP(P)}$ evaluated using the same features above,

as a function of α_{CPU} and P for different values of problem dimension on one node $n(1)$ (the value of problem dimension $n(P)$ on P nodes is scaled according to the same rule used above). From both the Figures 9 and 10, it turns out that for small values of α_{CPU} (i.e., where $\alpha_{CPU} < 0.15$) the proposed algorithm based on Schur reformulation of the problem (1) could exploit a very big fraction of the Peak Performance.

The low level of performance could potentially be attributed to implementation issues. Identifying such problems, carrying out profiling of the developed software module `schur_solve`, could be useful. For this reason, thanks to the use of the tool `nvprof` available in the CUDA Toolkit, a representation of this profiling has been generated (see Figure 11). The figure represents the profiling data of one of the four MPI tasks executed on one node to solve a problem whose dimensions are $n = 130{,}000$ and $n_1 = 13{,}000$, and where each MPI task uses $N_{OpenMPTasks} = 2$ OpenMP threads.

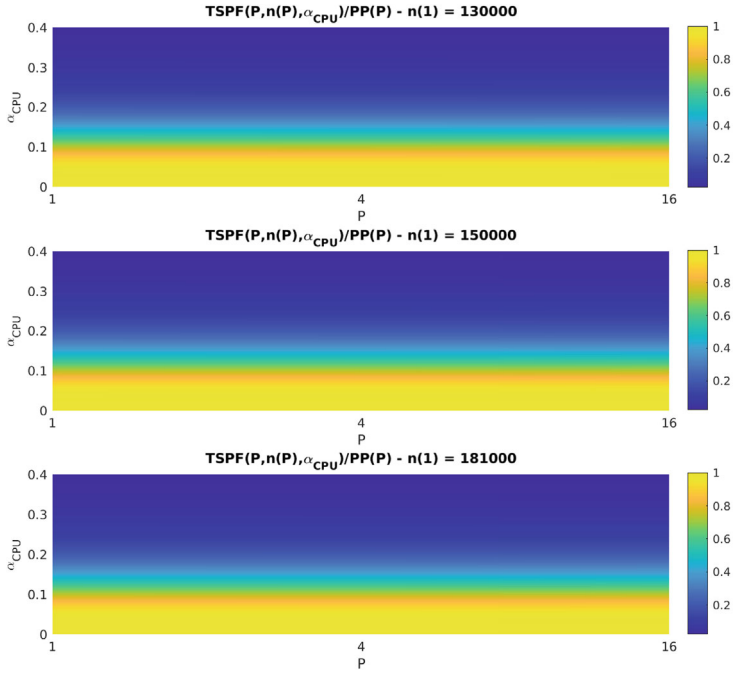
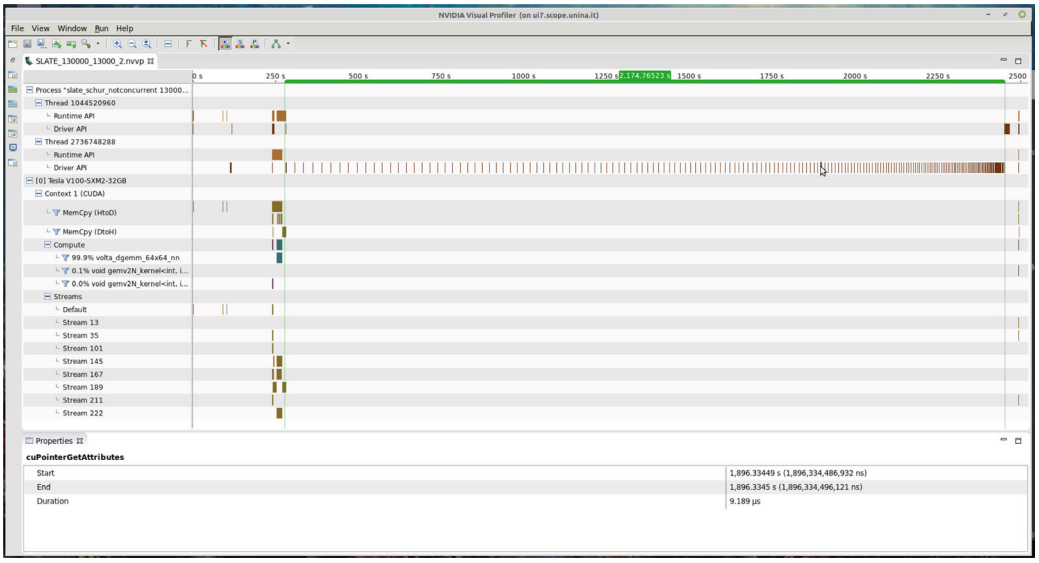


FIGURE 10 *"Heatmap"* of the Theoretical Peak Performance Fraction $TSPF(P, n(P), \alpha_{CPU})$ as a function of α_{CPU} and P for different values of the problem dimension on one node $n(1)$. The value of problem dimension $n(P)$ on P nodes is scaled according to the following rule $n(P) = n(1)\sqrt{P}$.



(A)

```
==358774== Device "Tesla V100-SXM2-32GB (0)"
          Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   48.26%  13.7927s        52  265.24ms  2.3680ms  532.17ms  volta_dgemm_64x64_nn
                   41.71%  11.9206s     25591  465.81us  1.3120us  11.892ms  [CUDA memcpy HtoD]
                    9.98%  2.85365s     13637  209.26us  1.6000us  542.69us  [CUDA memcpy DtoH]
                    0.03%  8.7302ms       234  37.308us  33.152us  39.360us  void gemv2N_kernel<int, int, double, double, double,
                                                                              int=128, int=32, int=4, int=4, int=1, bool=0,
                                                                              cublasGemvParams<cublasGemvTensorBatched<double const >,
                                                                              cublasGemvTensorBatched<double>, double>>(double const )
                    0.02%  4.3383ms        26  166.86us  163.55us  170.82us  void gemv2N_kernel<int, int, double, double, double,
                                                                              int=128, int=8, int=4, int=4, int=1, bool=0,
                                                                              cublasGemvParams<cublasGemvTensorBatched<double const >,
                                                                              cublasGemvTensorBatched<double>, double>>(double const )
```

(B)

FIGURE 11 Profile of the implementation of Algorithm 1 visualized by the NVIDIA Visual Profiler (A) and Summary of GPU activities (B). The figure represents the profile data, obtained by the `nvprof` tool, of one of the four MPI tasks executed on one node to solve a problem where $n = 130{,}000$, $n_1 = 13{,}000$. Each MPI task uses $N_{OpenMPTasks} = 2$ OpenMP threads.

The profiling view (see Figure 11A) reveals that a significant portion of the execution time (see the time range highlighted by the green box) was spent not in computing or CPU-GPU communication actions but in some other action type that seems related to memory access and/or management (see the very big number of calls to the CUDA driver function `cuPointerGetAttributes`). The Summary of GPU activities (see Figure 11B) demonstrates a relatively well-balanced distribution between computational and communication activities on the GPU, with a slight prevalence of the latter.

It's important to investigate these implementation issues, considering that many intricate details are concealed within the procedures of the SLATE library upon which the described implementation is built.

5 | CONCLUSIONS AND FUTURE WORK

This work outlines our initial efforts in designing a new Linpack-like Benchmark, based on the Schur Complement reformulation of the solution of a linear equation system, useful to evaluate the performance of Heterogeneous Computing Resources.

Our objective is not to develop a plan to replace the legacy HPL benchmark, which serves as the de facto standard for evaluating HPC platforms. Nonetheless, there seems to be a very heated discussion on the opportunity to supplement historical benchmarks with new tools capable not only of responding better to the availability of new technology but also of being more representative of the real system workloads (i.e., benchmarks based on sparse solvers). Therefore, this work has the objective of trying to contribute to this discussion in the hope that this can be considered useful for a (re-)formulation, which is considered necessary by many, of the historically consolidated tools.

We provide in-depth insights into the implementation and evaluation, with a primary focus on performance scalability, of our revamped Linpack Benchmark. These details pertain to a computing environment based on multi-NVIDIA GP-GPUs nodes interconnected by an Infiniband network. However, it's worth noting that our proposed approach is adaptable to various accelerator technologies, such as ROCm for AMD GPUs[32] or oneAPI for Intel accelerators.[54] Test results reveal that the benchmark's performance is on par with tools that predominantly emphasize the computational aspect linked to the GPU.

We anticipate that by enhancing the distribution of tasks across computational components and addressing the aforementioned implementation issues, we can elevate the benchmark's quality in measuring the performance of heterogeneous systems, especially in the context of scientific computing. Additionally, we envision that further performance improvements can be realized through the comprehensive utilization of the potential offered by CUDA-aware (or more generally, GPU-aware) MPI implementations. Some of our future work will be dedicated to these endeavors.

ACKNOWLEDGMENTS

This work has been funded by the NextGenerationEU project (HPC National Center, Big Data and Quantum Computing, Italian Center for Super Computing (ICSC), Mission 4, Component 2, Investment 1.4) with project code CN_00000013—CUP UNINA: E63C22000980007. This research used resources from the IBISCo Project Computing Facility available at the SCoPE Data Center of the University of Naples "Federico II".

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Luisa Carracciuolo https://orcid.org/0000-0002-8521-1645
Valeria Mele https://orcid.org/0000-0002-2643-3483

REFERENCES

1. Dongarra JJ. The evolution of mathematical software. *Commun ACM*. 2022;65(12):66-72. doi:10.1145/3554977
2. Dongarra JJ, Luszczek P, Petitet A. The LINPACK benchmark: Past, present and future. *Concurr Comput Pract Exp*. 2003;15(9):803-820. doi:10.1002/cpe.728
3. The LINPACK 1000 × 1000 benchmark program. http://www.netlib.org/benchmark/1000d
4. Petitet A, Whaley RC, Dongarra J, Cleary A. A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. https://www.netlib.org/benchmark/hpl/index.html
5. HPL-MxP mixed-precision benchmark. https://hpl-mxp.org/
6. Dongarra J, Sullivan F. Guest editors' introduction: The top 10 algorithms. *Comput Sci Eng*. 2000;2(1):22-23. doi:10.1109/MCISE.2000.814652
7. Yamazaki I, Hoemmen M, Luszczek P, Dongarra J. Improving performance of GMRES by reducing communication and pipelining global collectives. *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE; 2017:1118-1127.

8. Kwiatkowski J. Evaluation of parallel programs by measurement of its granularity. In: Wyrzykowski R, Dongarra J, Paprzycki M, Wasniewski J, eds. *Parallel Processing and Applied Mathematics*. Springer Berlin Heidelberg; 2002:145-153.
9. Heroux MA, Dongarra J. *Toward a New Metric for Ranking High Performance Computing Systems*. Sandia Report SAND2013-4744. Sandia National Laboratory; 2013.
10. Heroux MA, Dongarra J, Luszczek P. *HPCG Benchmark: A New Metric for Ranking High Performance Computing Systems*. Tech. Rep. UT-EECS-15-736. Electrical Engineering and Computer Science Department of the University of Tennessee; University of Tennessee; 2015.
11. Phillips E, Fatica M. A CUDA implementation of the high performance conjugate gradient benchmark. In: Jarvis SA, Wright SA, Hammond SD, eds. *High Performance Computing Systems. Performance Modeling, Benchmarking, and SimulationACM/IEEE International*. Springer International Publishing; 2015:68-84.
12. Bai Z, Hu D, Reichel L. A Newton basis GMRES implementation. *IMA J Numer Anal*. 1994;14(4):563-581. doi:10.1093/imanum/14.4.563
13. Hoemmen M. *Communication-avoiding Krylov Subspace Methods*. PhD thesis. University of California at Berkeley; 2010:AAI3413388.
14. Carracciuolo L, Mele V, Szustak L. About the granularity portability of block-based Krylov methods in heterogeneous computing environments. *Concurr Comput Pract Exp*. 2021;33(4):e6008. doi:10.1002/cpe.6008
15. Murli A, D'Amore L, Laccetti G, Gregoretti F, Oliva G. A multi-grained distributed implementation of the parallel block conjugate gradient algorithm. *Concurr Comput Pract Exp*. 2010;22(15):2053-2072. doi:10.1002/cpe.1548
16. The high performance conjugate gradients (HPCG) benchmark. https://www.hpcg-benchmark.org/
17. Laccetti G, Lapegna M, Mele V, Romano D, Murli A. A double adaptive algorithm for multidimensional integration on multicore based HPC systems. *Int J Parallel Prog*. 2012;40(4):397-409. doi:10.1007/s10766-011-0191-4
18. Bertero M, Bonetto P, Carracciuolo L, Laccetti G. MedIGrid: A medical imaging application for computational Grids. *Proceedings International Parallel and Distributed Processing Symposium*. IEEE; 2003:1213457. doi:10.1109/IPDPS.2003.1213457
19. Carracciuolo L, Lapegna M. Implementation of a non-linear solver on heterogeneous architectures. *Concurr Comput Pract Exp*. 2018;30(24):e4903. doi:10.1002/cpe.4903
20. D'Amore L, Constantinescu E, Carracciuolo L. A scalable space-time domain decomposition approach for solving large scale nonlinear regularized inverse ill posed problems in 4D variational data assimilation. *J Sci Comput*. 2022;91(2):59. doi:10.1007/s10915-022-01826-7
21. The Top 500 List Website. https://www.top500.org/
22. Tan G, Shui C, Wang Y, Yu X, Yan Y. Optimizing the LINPACK algorithm for large-scale PCIe-based CPU-GPU heterogeneous systems. *IEEE Trans Parallel Distribut Syst*. 2021;32(9):2367-2380. doi:10.1109/TPDS.2021.3067731
23. Golub GH, Van Loan CF. *Matrix Computations*. 3rd ed. The Johns Hopkins University Press; 1996.
24. HPL-AI mixed-precision benchmark. https://icl.utk.edu/hpl-ai
25. The CUDA basic linear algebra subroutine library. https://docs.nvidia.com/cuda/cublas/
26. A GPU accelerated library for decompositions and linear system solutions for both dense and sparse matrices. https://docs.nvidia.com/cuda/cusolver/
27. Kudo S, Nitadori K, Ina T, Imamura T. Implementation and numerical techniques for One EFlop/s HPL-AI benchmark on Fugaku. *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE; 2020:69-76.
28. Fatica M. Accelerating linpack with CUDA on heterogenous clusters. *GPGPU-2*. ACM. Association for Computing Machinery; 2009:46-51.
29. Fatica M, Ruetsch G. Optimization. In: Fatica M, Ruetsch G, eds. *CUDA Fortran for Scientists and Engineers*. Boston: Morgan Kaufmann; 2014:43-114.
30. Kim J, Kwon H, Kang J, Park J, Lee S, Lee J. SnuHPL: High performance LINPACK for heterogeneous GPUs. *ICS'22*. ACM. Association for Computing Machinery; 2022.
31. Chalmers N, Kurzak J, McDougall D, Bauman PT. Optimizing high-performance linpack for exascale accelerated architectures. https://arxiv.org/abs/2304.10397
32. AMD ROCm Open Ecosystem. https://www.amd.com/en/graphics/servers-solutions-rocm
33. Gates M, Kurzak J, Charara A, YarKhan A, Dongarra J. SLATE: Design of a modern distributed and accelerated linear algebra library. *International Conference for High Performance Computing, Networking, Storage and Analysis*. Special Interest Group on High Performance Computing; 2019.
34. Zhang F. *The Schur Complement and its Applications*. Numerical Methods and Algorithms. Vol 4. Springer; 2005.
35. Benzi M, Golub GH, Liesen J. Numerical solution of saddle point problems. *Acta Numer*. 2005;14:1-137. doi:10.1017/S0962492904000212
36. The Scalable LAPACK Project. http://www.netlib.org/scalapack/
37. PETSc, the Portable, Extensible Toolkit for Scientific Computation. https://petsc.org
38. Barone GB, Bottalico D, Carracciuolo L, et al. Designing and implementing a high-performance computing heterogeneous cluster. *Proceedings of 2022 International Conference on Electrical*. Computer and Energy Technologies (ICECET); 2022:1-6. doi:10.1109/ICECET55527.2022.9872709
39. Carracciuolo L, Bottalico D, Michelino D, Sabella G, Spisso B. Benchmarking a high performance computing heterogeneous cluster. *Parallel Processing and Applied Mathematics*. Springer International Publishing; 2023:101-114. doi:10.1007/978-3-031-30445-3_9
40. InfiniBand network standard. https://en.wikipedia.org/wiki/InfiniBand
41. Foley D, Danskin J. Ultra-performance pascal GPU and NVLink interconnect. *IEEE Micro*. 2017;37(2):7-17. doi:10.1109/MM.2017.37
42. NVIDIA Mellanox OFED DOC. https://docs.mellanox.com/display/MLNXOFEDv531001/NVIDIA+MLNX_OFED+Documentation+Rev+5.3-1.0.0.1
43. Nickolls J, Buck I, Garland M, Skadron K. Scalable Parallel Programming with CUDA. *Queue*. 2008;6(2):40-53. doi:10.1145/1365490.1365500
44. Kraus J. An introduction to CUDA-aware MPI. https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/
45. Open MPI: Open source high performance computing. https://www.open-mpi.org/
46. Shamis P, Venkata MG, Lopez MG, et al. UCX: An open source framework for HPC network APIs and beyond. *IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE; 2015:40-43.
47. Interprocess Communication—Programming Guide :: CUDA Toolkit DOC. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#interprocess-communication
48. GPUDirect RDMA—CUDA Toolkit DOC. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html
49. Shi R, Potluri S, Hamidouche K, et al. Designing efficient small message transfer mechanism for inter-node MPI communication on InfiniBand GPU clusters. *21st International Conference on High Performance Computing (HiPC)*. IEEE; 2014:1-10.
50. OpenMP (Open Multi-Processing) Wikipedia Page. https://en.wikipedia.org/wiki/OpenMP

51. Gates M, Charara A, Kurzak J, et al. SLATE users' guide. *SLATE Working Notes 10, ICL-UT-19-01*. Innovative Computing Laboratory; University of Tennessee; 2020.
52. HPC Wiki–Scaling. https://hpc-wiki.info/hpc/Scaling#Strong_or_Weak_Scaling
53. Bosilca G, Bouteiller A, Danalis A, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE; 2011:1432-1441.
54. Intel oneAPI Toolkits. https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html

How to cite this article: Carracciuolo L, Mele V, Sabella G. Toward a new linpack-like benchmark for heterogeneous computing resources. *Concurrency Computat Pract Exper*. 2023;e7962. doi: 10.1002/cpe.7962