The 9th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2024
3–7 June 2024, Lisboa, Portugal

# EFFICIENT HIGH-FIDELITY SIMULATIONS FOR THE ENERGY TRANSITION USING ARM AND X86_64 ARCHITECTURES

**ANI ANCIAUX-SEDRAKIAN[1], RAPHAEL GAYNO[1], THOMAS GUIGNON[1] and ABOUL-KARIM MOHAMED EL MAAROUF[1]**

[1] IFP Energies nouvelles
1-4 avenue Bois-Préau,
Rueil Malmaison, 92852 cedex, France
e-mail: firstname.lastname@ifpen.fr, www.ifpenergiesnouvelles.fr

**Key words:** high-performance computing, portable and component-based design, programming models, SIMD, memory-bound application, sparse linear algebra, iterative methods

**Summary.** High-performance scientific computing is at the heart of digital technology. It enables the simulation of complex physical phenomena in a reasonable time. The current trend for computer hardware architecture is to shift towards more parallelism. To use these systems in the most optimal way different levels of parallelism need to be studied. The demand for a fast solution of simulations coupled with new computing architectures drives the need for an adequate programming model, challenging parallel algorithms, a suitable memory layout, appropriate architecture-specific parameters and a flexible software architecture. This paper focuses on two prominent x86 and ARM hardware architectures. First, it highlights the key considerations for achieving more efficient computation at the node level. It also evaluates whether optimal use of the mentioned computing resources can be possible by considering all levels of parallelism and how. It addresses the aforementioned aspects, through an iterative preconditioned linear solver for sparse, non-symmetric matrices. The scalability and time-to-solution of several test cases are demonstrated and analyzed.

## 1 Introduction

Selecting the best sites for exploiting lithium from hydro-thermal sources, or ensuring long-term storage of CO2 in deep aquifers, are major challenges requiring a deep understanding of fluid movements in the subsurface. To meet these energy transition challenges, numerical simulation is an indispensable tool requiring efficient usage of high-performance scientific computing resources. The most time-consuming part of such Geoscience simulations consists of solving non-symmetric, sparse, ill-conditioned matrices. It can even contribute to 80% of the simulation time, as the Jacobian matrices generated by unsteady and nonlinear problems are solved across multiple Newton iterations and time steps.

Today's hardware architectures make parallel programming for numerical simulation more challenging. In this paper, we study how to improve the performance and exploit optimally computing resources based on x86 and ARM architectures, excluding the consideration of accelerators like GPUs. The optimal performance could not be reached by using the same settings for all parallel computing resources. To tackle this challenge, we focus on sparse linear preconditioned solvers. The reason for this is twofold. First of all, linear system solution is one of the most time-consuming phases of simulations based on Partial Differential Equations (PDEs), such as those in the Geoscience applications, and therefore its performance has a direct impact on the performance of the overall simulation. Second, the used algorithms

are memory-bound based, so providing an efficient solution on the target architectures is challenging, and represents a tricky problem. In the context of solving linear systems, the legacy libraries are not in the optimal position to fully exploit the highly parallel architectures. To deal with this issue, different research works are carried out to bring an adapted solution. Each of them tries also to minimize the impact of the permanent evolution of hardware architectures. There are three main approaches for solving sparse linear systems: direct, iterative, and hybrid, all with different parallelism constraints. Systems resulting from the discretization of PDEs are generally solved using iterative Krylov subspace methods, such as Generalized Minimal RESidual (GMRES), Conjugate Gradient (CG), Bi-Conjugated Gradient Stabilized (BiCGStab) or their different variants preconditioned using various methods like polynomial, SParse Approximate Inverse (SPAI), incomplete LU, incomplete Cholesky, or Multi Grid [1].

Regarding multi-core architectures, several works have been carried out on direct sparse solvers, such as SuperLU[2], SLATE[3], Paradiso[4], or PaStix [5]. These solvers try to manage the shared memory parallelism, through the use of multi-threading technologies like OpenMP, or DAG-based runtime systems like StarPU[6] approaches. For sparse iterative solvers on heterogeneous multi-core systems (CPU+GPU), several studies in different libraries suggest various approaches, such as Hypre, Ginkgo, and Trilinos. Hypre proposes a strategy based on the BoxLoop loop abstraction [7], while Ginkgo [8] uses different programming models (CUDA, HIP, SYCL, OpenMP) through the notion of 'Executors,' which determine where the operations will be executed. Trilinos [9] addresses different programming models using the Kokkos ecosystem [10].

The main contributions of this paper are as follows: First, we analyze the critical performance concerns for x86 and ARM architectures when handling non-symmetric sparse matrices. Second, through the proposed software architecture, we demonstrate how to manage different programming models and architecture-specific features in a portable way without compromising performance. Afterwards, we examine how to exploit available computing resources most efficiently and assess whether all levels of parallelism can be fully leveraged. Finally, we show the performance evaluation using systems arising from hyperbolic-elliptic equations. The rest of this paper is organized as follows. Section 2 summarizes target hardware architecture characteristics and describes the key technological innovations and optimization strategies. Section 3 details how we take into consideration hardware architecture characteristics in preconditioned iterative sparse linear solver context, while Section 4 details our experiments. Section 5 concludes with an outline of potential directions for future research.

## 2 Hardware Architecture

Modern parallel computing resources allow us to exploit multiple levels of parallelism. For example, distributed memory parallelism with MPI, shared memory parallelism with OpenMP, Cilk, Pthreads, or task-based execution systems, and at the most granular level, vector parallelism using SIMDs intrinsic instructions. Additionally, Instruction-Level Parallelism (ILP) where compilers play a crucial role, and parallelism specific to accelerators using CUDA, HIP, OpenACC, etc. represent further levels of parallelism. However, it is important to note that this work does not address the last two levels of parallelism.

Let's begin by examining how to efficiently exploit shared memory parallelism at the CPU level. This can be achieved in several ways: (i) implicit parallelization through multithreaded libraries, (ii) parallel and distributed extensions of the C++ language [11], (iii) task-based runtime systems like TBB, StarPU[6], OmpSs[12], (iv) loop parallelism using directive-based models like OpenMP, and (v) domain partitioning with threading models such as OpenMP and Pthreads. With the mentioned programming models, there are no dependencies on hardware architectures. However, the core level parallelism with

SIMD capability suffers from a lack of portability on different hardware architectures (x86, ARM) and the possibility to use different instruction sets (SSE, AVX, AVX2, AVX-512, NEON, SVE, SVE2). This level of parallelism allows to perform the same operation on multiple data simultaneously. The size of the hardware SIMD registers and the type of data can determine this amount of data. The size of the hardware SIMD registers may vary from one architecture to another. Their size, along with the type of data, can determine the amount of data to be processed. This level of parallelism is an important aspect to study for memory-bound applications. Due to its lack of portability, several programming approaches have been explored to deal with the different and specific types of instructions [13]. These approaches can be classified into four categories: auto-vectorization, done by the compiler, the use of a dedicated language such as OpenMP (guided vectorization), the direct use of intrinsics, or a dedicated library such as Boost-SIMD or MIPP [14].

Besides managing different levels of parallelism and the associated programming models, we are also constrained to use other aspects such as advanced memory hierarchies, and various tunable parameters that impact performance. These parameters include data locality, memory-aware thread-binding policies, data representation, and synchronization overhead, among others. Regarding data locality, some computing resources are based on Non-Uniform Memory Access (NUMA), where the cost of memory access varies depending on the memory's proximity to the processor. In NUMA-based architectures, each core memory address access has a different cost depending on its relative location. Memory localization is then an important issue, particularly on architectures with multiple NUMA nodes with a large number of cores per socket.

## 3 Preconditioned Sparse Iterative Linear Solvers

This section describes how to exploit hardware architecture specifications to achieve a portable and efficient iterative linear algebra library for sparse matrices.

### 3.1 Software Architecture and Programming Models

The proposed software architecture focuses on decoupling the core numerical algorithms from the back-end components specific to the target architecture. By doing so, it simplifies the development of new solvers and preconditioners, enabling developers to concentrate on algorithm design without dealing with the complexities of the underlying hardware. This approach allows for the creation of new solver or preconditioner classes by combining the relevant building blocks. Additionally, there is no specific vector type required; the only constraint is that the vector must have a continuous memory layout. This flexibility permits the use of different floating-point arithmetic operations. One question could be the use of solutions like Raja, SYCL, or SYCL through Raja. These mentioned high-level programming models permit to improve the programming productivity on various hardware architectures. Their additional programming layers also allows for implicit data management and movement which can be considered an advantage. However, as far as we know, we cannot control low-level parallelism, use customized synchronization and reduction operations, specific memory allocation and manage data movement. Kokkos library designed to manage parallelism across various hardware architectures, provides more direct control over memory management and thread synchronisation compared to the previously mentioned solution [10]. Kokkos will be integrated into the proposed software design as an additional programming model, enabling future performance comparisons between Kokkos and the proposed solution. Listing 1 provides an example of how to construct the necessary components for implementing a preconditioned linear solver in a manner that is both portable and abstract.

Listing 1: Preconditioned linear solver with appropriate algebra, sparse matrix-vector product, and preconditionner

```
template<typename AlgebraT,typename MVOperatorT,
         typename PrecOperatorT>
class BiCGStabSolver
{
private:
  AlgebraT::VectorType u;
  AlgebraT::VectorType pp;
  AlgebraT::VectorType r0;
  AlgebraT::VectorType p;
  ...
public:
  BiCGStabSolver(const int vec_size,
                 AlgebraT& algebra,
                 const int max_iter,
                 const double residual);
  ~BiCGStabSolver();
  void solve(MVOperatorT* mv, PrecOperatorT* prec,
             const AlgebraT::VectorType& rhs,
             const AlgebraT::VectorType& sol)
 {
  ...
  for(i=0;i<m_max_iter;++i)
  {
    ...
    prec->y_Fx(p,pp);
    ...
    mv->y_Fx(pp,u);
    alpha = m_algebra.dot(u,r0);
    ...
    }
  }
```

This proposed software architecture is designed to be easily extendable to new hardware architectures by accommodating their unique constraints, features, and programming models, thereby ensuring adaptability and efficiency across diverse platforms. We follow three distinct approaches to maintain easy portability: template-based object-oriented programming, code generation, and specific library usage. The template parameter allows instantiating different Sparse Matrix-Vector products (SpMV) "*MVOperatorT*", different preconditioners "*PrecOperatorT* " and different basic vector level linear algebra operations "*AlgebraT*" corresponding to the target hardware architecture, implemented by its specific programming models/instructions described above. The proposed architecture also allows for the combination of different programming models to handle various levels of parallelism and evaluate them easily. We utilize shared memory parallelism at the CPU level, and more precisely at the NUMA node level. If the use of MPI is an obvious choice with distributed memory, to manage shared memory, several runtimes and approaches (task, static, etc.) are available and discussed. To evaluate some of these approaches in the context of sparse iterative linear solvers, we use a code generation approach to create the corresponding Kernel implementations. From the list of target operators, the developed Pythonic generator can currently generate several kernels as OpenMP, OpenMP-task or OmpSs.

In the context of sparse iterative linear solvers, synchronizations and memory access are the main issues. They have a low ratio of floating-point operations to memory accesses compared to direct methods and ones used for dense matrices. This fact makes using task-based runtime systems less attractive. To take advantage of better parallelism and to avoid as much as possible synchronization cost, which is a real bottleneck, loop-level parallelization in shared memory should be avoided. Our solution involves using a

domain decomposition strategy per NUMA node. The local data for each core can be parallelized using the available SIMD instructions. At the core level, we evaluate two different approaches to leverage vectorization. First, we added specific kernels written by AVX2, and AVX512 intrinsics. This approach raises flexibility and portability issues. To overcome this, we have opted for the use of specific libraries with a unified API, such as MIPP. MIPP permits both to avoid increasing SIMD kernel implementations on the application for other architectures, such as RISC-V or the different extensions and also deals with mixed precision. For this work, we first enriched the MIPP library with SVE instructions.

### 3.2 Local Partitioning Effects

Maximizing memory manipulation efficiency is one of the main performance improvement keys, mainly for NUMA-based systems where the cost of accessing shared memory resources depends on who is accessing them. Data locality and load balancing are crucial in the context of sparse iterative linear solvers, characterized by the fact that they are memory-bound. To benefit from all parallelism levels, we use hierarchical domain partitioning. For both distributed memory (inter node, inter CPU, inter NUMA node) and shared memory parallelisms (intra node, intra CPU, intra NUMA node) graph partitioning algorithms satisfy both criteria. In Section 4, we illustrate the gain obtained by using a multilevel k-way partitioning scheme. In the context of partition decomposition with OpenMP, the best performance can be achieved if we stay in the same NUMA node, and even better if we manage to stay in the same cache (see Figure 4). To enforce suitable data locality, we assign one MPI process per NUMA node. All available cores within each NUMA node then run in parallel using a threading-based parallel programming model. This strategy allows cores on a given socket to access their local memory.

### 3.3 Synchronisation Barrier and Reduction Operations

During each iteration of the preconditioned linear solver, several sparse matrix-vector products, dot products and norm computations are performed. For example, during BiCGStab iterations, two sparse matrix-vector products, six axpy's, and five dot products are computed. Since each processor has the scalars and its corresponding part of the vectors, the saxpy's can be parallelized without communication. However, this is not the case for SpMV's, dot products and norm computations, which require communication to obtain the desired results. Such operations cause scalability issues and severe performance degradation. In the case of the shared memory context, these communications are transformed into synchronization barriers and reduction operations. To address this problem in the shared memory context, the synchronization cost can be reduced by replacing OpenMP's native reductions and synchronization barriers with alternative instructions that offer better performance. In this paper we study a new version so-called ”linear centralized barrier” which is also combined with the reduction operation.

The principle of this barrier is as follows. When threads reach the fence instruction, they signal their presence by reversing their synchronization flag, and then they go into a busy wait. This barrier differs from the ”sense-reversing centralized barrier” [15] (”mcs centralized barrier” in Figure 1) by choosing one thread to check the incoming and outgoing phases of the barrier. This avoids the use of atomic operations, which are efficient for the small number of threads. Regarding reduction operation, we consider a SIMD vector corresponding to a cache line. The first element of the SIMD vector acts as a synchronization variable. When core 0 writes to this variable, the entire cache line is invalidated on core 1, which is waiting for a change in this variable. As a result, core 1 reloads the cache line with the new value of the variable and exits the wait loop. Instead of updating a single vector element, we perform an atomic SIMD write. This ensures that when core 1 exits the wait loop, all elements are correctly

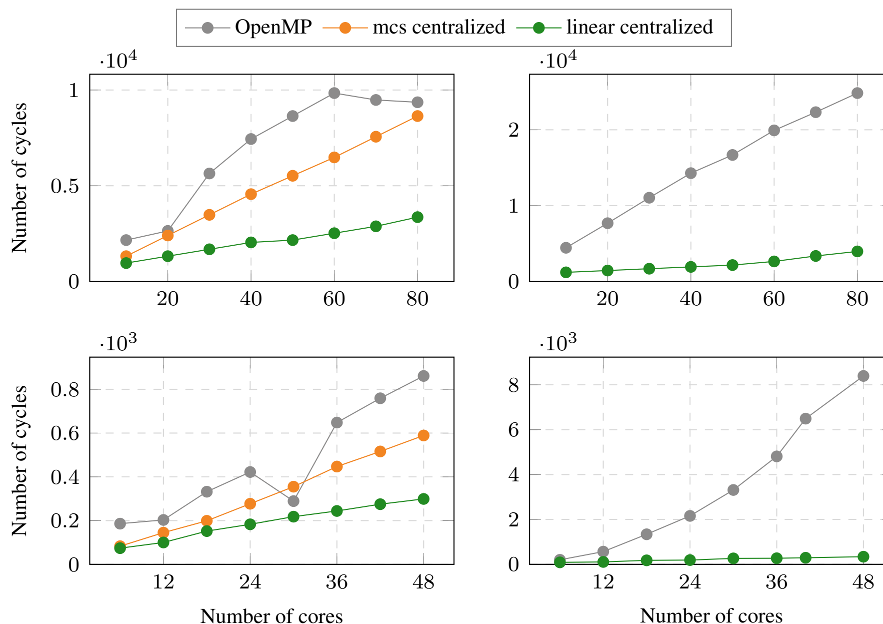


Figure 1: Efficiency of different synchronization barriers (left) and reduction operations (right) for ARM Ampere Altra (top) and Joliot-Curie A64FX (bottom) architectures

updated. We compare the proposed linear centralized barrier and reduction operations overhead relative to OpenMP using the EPCC micro-benchmark suite. We obtained a gain of 2.7 for the synchronisation barrier and 6.28 for the reduction operation over 80 cores using Ampere Altra system. Regarding A64FX system, we obtain a gain of 2.9 for the synchronisation barrier and 24.6 for the reduction operations using 48 cores. Figure 1 shows, the efficiency of the discussed operations on the Ampere Altra and A64FX.

## 4 Performance Results

In this section, we present our experiments and discuss the performance and scalability results for different architectures.

### 4.1 Platforms and Study Cases Description

The performance evaluations are performed on five different architectures described below.

**Ener440**: Supercomputer at IFPEN has 240 bi-socket compute nodes, based on Intel SkylakeX G-6140@2.3 GHz (AVX, AVX2, AVX512) CPUs with 18 compute cores and one NUMA node. The nodes are connected through the Intel Omni-PAth interconnect.

**Topaze**: Supercomputer at CCRT has 864 bi-socket compute nodes, based on AMD Milan@2.45GHz (AVX2) CPUs with 64 compute cores and 4 NUMA nodes. The nodes are connected through HDR-100.

**Joliot-Curie**: Supercomputer hosted by TGCC with several partitions. The ARM partition is based on Fujitsu PRIMEHPC FX700 technology [1] with 48 compute cores, 2.0 GHz, Armv8.2-A SVE (512) instruction set and 4 NUMA nodes. The 80 single-socket ARM A64FX are connected through Mellanox

[1]Fujitsu: https://www.fujitsu.com/global/products/computing/servers/supercomputer

| Matrix | Rows | Block size | Nonzero block elems | Matrix collection |
|---|---|---|---|---|
| Cage8 | 1015 | 1 | 11,003 | Florida matrix collection |
| PqDM1 | 2,660 | 3 | 14,427 | Three phase flow Black-oil model |
| Canta | 8,016 | 3 | 57,187 | Three-phase flow Compositional model |
| Poisson3Db | 85,623 | 1 | 2,374,949 | Florida matrix collection |
| GCS | 185,498 | 3 | 1,094,385 | Three-phase flow Dual-medium model |
| PAB | 4,223,128 | 3 | 25,111,743 | Three-phase flow Black-oil model |

Table 1: Main characteristics of the evaluated matrices and their origin (matrix belonging).

EDR InfiniBand. The Rome partition contains 2292 dual-processor AMD Rome (Epyc) compute nodes at 2.6 GHz (AVX2) with 64 cores per processor. The nodes are connected through HDR-100.

**Ampere Altra**: One 64-bit multi-core ARM CPU with 80 ARM compute cores up to 3.30 GHz maximum and one NUMA node. The used technology is Armv8.2+ with Neon instruction set architecture.

We demonstrate the performance of the proposed work through two series of experiments. The first series consists of a set of non-symmetrical matrices given in Table 1. Some systems are obtained from different representative models of dynamic Geoscience applications at different time steps and others from the Florida collection. They illustrate the obtained behaviour relative to the key points mentioned in the previous sections to obtain a more efficient calculation at the node level and discuss multiple nodes' performance. Due to a lack of space, we do not show the obtained results for all the configurations. The second series demonstrates the scalability of the solution proposed in the ShArc application [2]. ShArc is a mini-application of Geoxim permitting simulation of CO2 geological sequestration. For the presented results we use the K-way partitioning proposed by Metis, and double-precision arithmetic for most of the computation and mixed-precision during the incomplete LU preconditioning phase (the computation phase in double precision and the storage of results in single precision). The results were obtained using the BiCGStab solver, preconditioned with block Jacobi and incomplete LU with zero levels of fill-in. For all numerical experiments, the tolerance for relative residual error is set at 1e-06. The following common software environments are used for the computations: GCC/11.2.0, OpenMPI/4.1.1, hwloc/2.5.0, OpenBLAS/0.3.22, and ParMETIS/4.0.3. Using OMP_WAIT_POLICY environment variable set to ACTIVE and OMP_PROC_BIND to TRUE, and we assume to have a memory-aware thread-binding policy.

### 4.2 Detailed Results at Node Level Computations

In this section, first, we evaluate the impact of core-level parallelism with SIMD capabilities, and then discuss the different aspects to be considered at the node level. Figure 2 and Figure 3 illustrate the vectorization performance gains achieved using i) auto-vectorization, done by the compiler (using fvect-cost-model, loop and slp vectorizer ), ii) handwritten intrinsics and iii) the MIPP library. The numerical experiments are evaluated using different matrices (very small, small, scalar and block matrix) on two different hardware architectures (x86 Intel SkylakeX and ARM A64FX). We don't illustrate the results using Neon intrinsics because the obtained results can't be much given the size of the register. Auto-vectorization, done by the compiler (using fvect-cost-model, loop and slp vectorizer ) is represented by the *compiler-vec* label. The handwritten intrinsics is labelled *AVX512*. The explicit vectorization

[2]Sharc: https://github.com/arcaneframework/sharc

with MIPP library which handles the choice of intrinsics at compile time is tagged with *MIPP-AVX512* and *MIPP-SVE*. Depending on whether we use intrinsics (manual implementation or using the MIPP wrapper), we employ the BCSP layout, an adaptation of the Sell-C-$\sigma$ [16], which allows efficient SpMV implementations using various SIMD intrinsics, instead of BCSR.

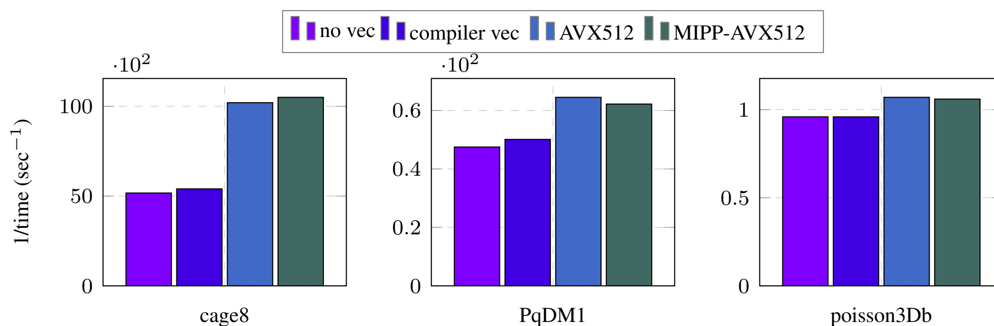


Figure 2: Ener440 Intel(x86): the performance of BiCGStab using vectorization for different size of systems

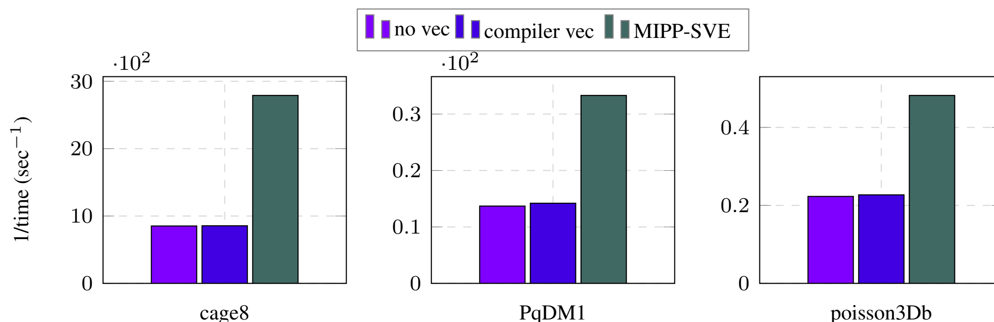


Figure 3: Joliot-Curie A64FX (ARM): the performance of BiCGStab using vectorization for different size of systems

| | **no vec** | **compiler vec** | **MIPP-SVE** |
|---|---|---|---|
| number of iterations | 2,40E+01 | 2,40E+01 | 3,00E+01 |
| time(sec) | 4,02E-02 | 3,46E-02 | 2,56E-02 |

Table 2: Joliot-Curie A64FX (ARM): the time-to-solution, using vectorization, of preconditioned BiCGStab with incomplete LU factorization of the PqDM1 system

The results presented in Figure 2 and Figure 3 are obtained using a non-preconditioned BiCGStab solver. In fact, vectorization of the sparse triangular systems resolution resulting from incomplete LU factorization is a challenging task. Mohamed El Maarouf et al. in [17] present an approach based on a specific multi-coloring ordering which provides enough fine-grain parallelism for SIMD computational units. Table 2 illustrates the behaviour of preconditioned BiCGStab solver with incomplete LU factorization using the proposed solution by Mohamed El Maarouf et al. in [17] using a single core. The number of iterations may increase a bit, but the performance gain remains interesting. How to manage the different levels of parallelism in this context will be addressed in future work. The presented exper-

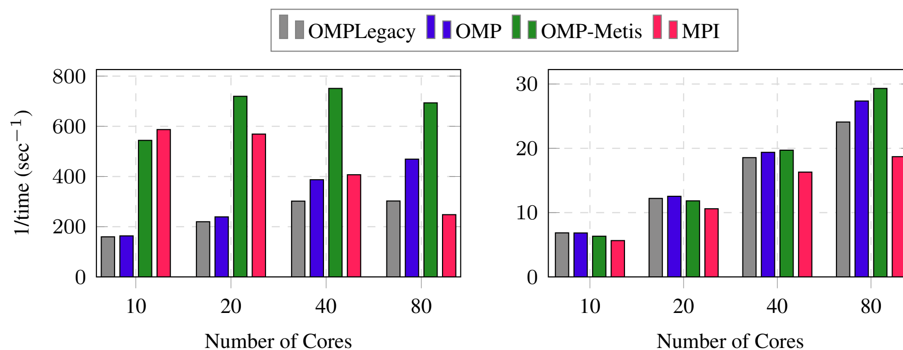


Figure 4: Ampere Altra (ARM): the performance for the solution of the PqDM1 system (1/time to solution) (left), Canta system (right)

iments demonstrate the benefits of core-level parallelism (vectorization) on Intel SkylakeX and A64FX processors, highlighting the performance gains achieved on these architectures. Since the obtained performance via the MIPP library usage is comparable to that of handwritten, for the ARM-based architecture and other upcoming ones we enrich the MIPP library. The other important result to highlight is the performance gap achieved with ARM-based architecture and with Intel SkylakeX on common software environments. The authors in [18] mention how the choice of compiler and libraries can greatly affect simulation speed and energy efficiency. These experiments demonstrate that the excepted performance gains cannot always be guaranteed when using vectorization on every architecture.

The performance results shown in Figure 4 confirm the approach described in Section 3.1. The difference between OMPLegacy and OMP results lies in the use of reduction operations and synchronization barriers. In OMPLegacy, we use OpenMP's built-in reduction and barriers, while in OMP, we use the approach discussed in Section 3.3. The number of iterations remains constant in these experiments.

### 4.3 Detailed Results Using Multiple Nodes

In this section, we focus on the efficient use of multiple nodes and evaluate whether all levels of parallelism can be considered. The obtained results using ARM and x86 architectures are given in Figure 6. Both exhibit the potential gain that can be expected from fine-grained calculations and data localities. In fact, in the context of domain decomposition with OpenMP, as shown in Figure 5 the best performance can be achieved if we stay in the same NUMA node.

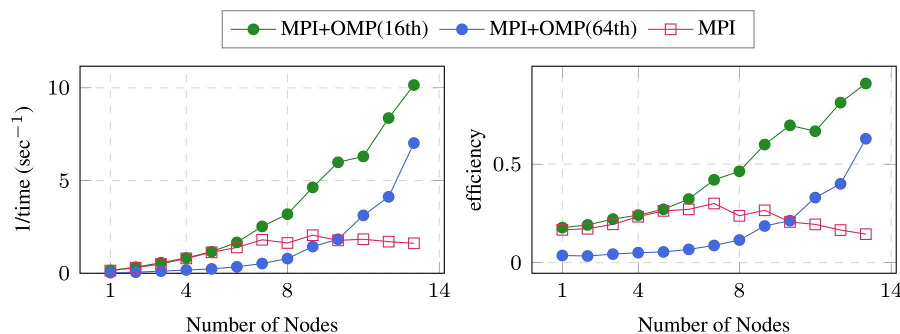


Figure 5: Topaze (AMD x86): the performance (1/time to solution) for PAB system (left), the efficiency relative to one core (right)

Regarding exploiting multiple levels of parallelism, Figure 7 illustrates performance variations when using all levels of parallelism for Intel SkylakeX and A64FX architecture. In both architectures, better performance is obtained when using 8 nodes by considering all the levels of parallelism. The performance gain compared to MPI is 2.3 for SkaylakeX and 5 for A64FX. Focusing on one node we notice different behaviours on these two architectures. Unlike on the A64FX architecture, on a single Intel SkylakeX node, no performance difference is observed. On one node of A64FX, the performance ratio compared to MPI using the SVE intrinsics is 2.2. Taking into account the shared memory parallelism, the gain is 10, and by combining the three levels of parallelism, i.e., distributed memory, shared memory and, vectorization the gain is 16. Schöne et al. in [19] discuss about the variation of the frequency and the power consumption when using vector-level parallelism using SIMD intrinsic. This explains the observed difference in performance gains.

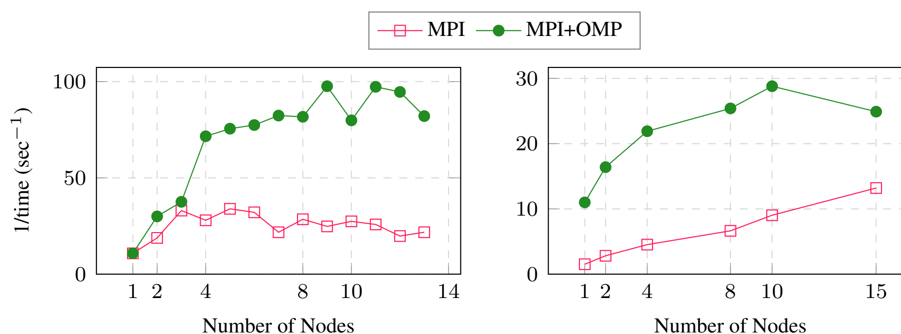


Figure 6: The performance (1/time to solution) for GCS system using Ener440: Intel x86(left), and using Julio-Curie: A64FX ARM (right)

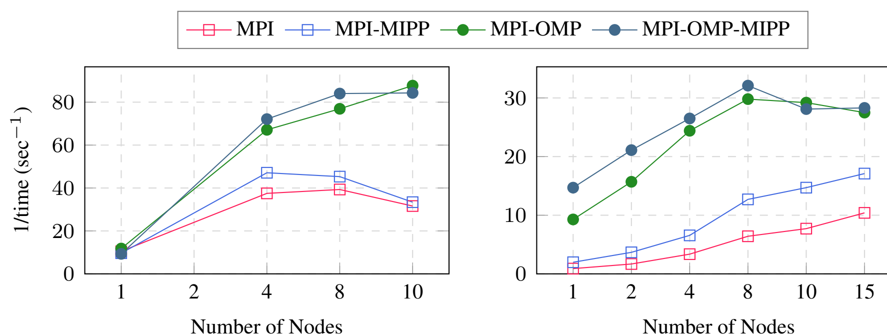


Figure 7: The performance (1/time to solution) of BiCGStab for GCS system using Ener440: Intel x86(left), and using Julio-Curie: A64FX ARM (right)

### 4.3.1 Detailed Results Using Unsteady Model

In this section, we present the results obtained from a simulation of the ShArc mini-application of Geoxim permitting simulation of CO2 geological sequestration. This series is evaluated on the Rome partition of the Joliot-Curie machine. The addressed problem is a two-phase flow unsteady model where we simulate 10-time steps on a structured grid of size 8 388 608. This synthetic test case is a gas injection

in a porous medium containing water. The results in Figure 8 show a good scalability of up to 30 nodes when the distributed memory parallelism fails to scale after 20 nodes. It illustrates the same behaviour as the results presented in the previous section. The efficiency presented in this figure is calculated relative to one full node. Author in [20], presents more comparison of the proposed solution (named MCGSolver) with other software packages like Petsc, and Trilinos using an Arcane-based application.

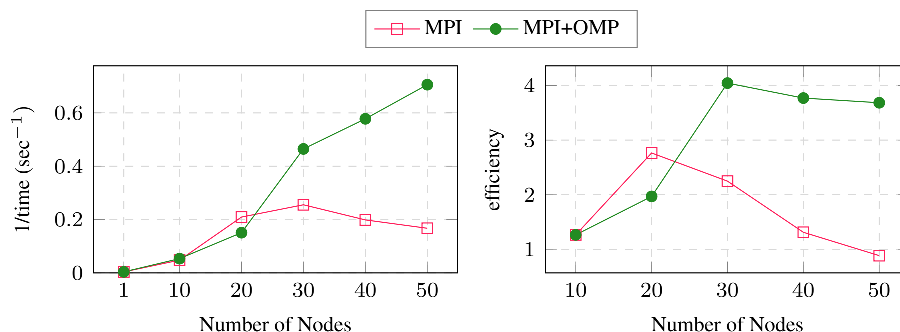


Figure 8: Juloit-Curie (AMD x86): the performance (1/time to solution) for shrac steady model (left), the efficiency relative to one node (128 cores) (right)

## 5 Conclusions and Future Work

Getting high performance on the constantly evolving parallel computing resources remains a challenging task for applications due to the complex hardware architectures and numerous parameters that significantly impact performance. In this paper, we have discussed a subset of key factors essential for optimizing performance, with a focus on managing parallelism at various levels, data layout and data localities for x86 and ARM architectures. This discussion is centered on the use of preconditioned iterative linear solvers based on Krylov subspace methods for non-symmetric sparse matrices, which arise from hyperbolic-elliptic equations and are considered memory-bound application. To achieve optimal performance, several levels of parallelism must be carefully studied: intra-core parallelism through SIMD vectorization, on-node parallelism via multi-threading, and inter-node parallelism using message-passing techniques. We presented a strategy through a preconditioned linear solver, leveraging object-oriented facilities that allow for flexibility in dealing with different programming models. This approach offers an easy extension to new hardware architectures, demonstrating significant performance gains over traditional MPI solutions. We illustrated that the key issue remains the variation of optimal configurations across different systems. Identifying the best configuration for each specific system and application is a difficult task. In future work, we will study the impact of additional hardware parameters and specifications, such as compiler behavior and optimization, prefetchers, and memory hierarchy, on performance and energy consumption, and explore how to find the optimal parameters for a target architecture.

### Acknowledgement

We acknowledge GENCI for giving us access to the Joliot-Curie A64FX ARM (prototype) hosted by the TGCC during projects S142bspe00035 and AD010613383R1.

## REFERENCES

[1] Y. Saad. doi:10.1137/1.9780898718003.

[2] X. S. Lidoi:10.1145/1089014.1089017.

[3] M. Gates, J. Kurzak, A. Charara, A. YarKhan, J. Dongarradoi:10.1145/3295500.3356223.

[4] M. Bollhöfer, O. Schenk, R. Janalik, S. Hamm, K. Gullapallidoi:10.1007/978-3-030-43736-7_1.

[5] P. Hénon, P. Ramet, J. Romandoi:10.1016/S0167-8191(01)00141-7.

[6] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenierdoi:10.1007/978-3-642-03869-3_80.

[7] R. D. Falgout, R. Li, B. Sjögreen, L. Wang, U. M. Yangdoi:10.1016/j.parco.2021.102840.

[8] H. Anzt, T. Cojean, Y.-C. Chen, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y.-H. Tsaidoi:10.21105/joss.02260.

[9] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, K. S. Stanleydoi:10.1145/1089014.1089021.

[10] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, J. Wilkedoi:10.1109/TPDS.2021.3097283.

[11] A. Williams, C++ Concurrency in Action, Second Edition. Manning Publications.

[12] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, E. S. Quintana-Ortídoi:10.1007/978-3-642-02303-3_13.

[13] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, B. Juurlinkdoi:10.1145/2870650.2870653.

[14] A. Cassagne, O. Aumage, D. Barthou, C. Leroux, C. Jégodoi:10.1145/3178433.3178435.

[15] J. M. Mellor-Crummey, M. L. Scottdoi:10.1145/103727.103729.

[16] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. R. Bishopdoi:10.1137/130930352.

[17] A.-K. Mohamed El Maarouf, L. Giraud, A. Guermouche, T. Guignon, A multi-coloring ordering for preconditioned krylov solvers, Compas (2022).

[18] N. A. Simakov, R. L. Deleon, J. P. White, M. D. Jones, T. R. Furlani, E. Siegmann, R. J. Harrisondoi:10.1145/3581576.3581618.

[19] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, D. Hackenbergdoi:10.1109/HPCS48598.2019.9188239.

[20] J. Gratien, C. Chevalier, T. Guignon, X. Tunc, P. Have, S. De Chaisemartindoi:10.23967/eccomas.2022.112.