Evaluation of the performance portability layer of different linear solver packages with alien, an open generic and extensible linear algebra framework.

Ani Anciaux-Sedrakian, Cédric Chevalier, Stéphane de Chaisemartin, Jean-Marc Gratien, Thomas Guignon, Pascal Havé, Nathalie Möller, Xavier Tunc

The 8th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2022
5-–9 June 2022, Oslo, Norway

EVALUATION OF THE PERFORMANCE PORTABILITY LAYER OF DIFFERENT LINEAR SOLVER PACKAGES WITH ALIEN, AN OPEN GENERIC AND EXTENSIBLE LINEAR ALGEBRA FRAMEWORK

Ani Anciaux-Sedrakian[1], Cédric Chevalier[2], Stéphane De Chaisemartin[1], Jean-Marc Gratien[1], Thomas Guignon[1], Pascal Havé[3], Nathalie Möller[2] and Xavier Tunc[1]

[1] IFP Energies nouvelles, 1-4, avenue du Bois-Préau, 92852 Rueil-Malmaison, France
http://www.ifpenergiesnouvelles.fr

[2] LIHP Université Paris-Saclay, CEA, DAM, DIF, F-91297 Arpajon, France
http://www.cea.fr

[4] HAVENEER, https://www.haveneer.com

Key words: Linear solvers, HPC, Performance portability, Kokkos, SYCL

Abstract. Applications to solve large and complex partial derivative equation systems often rely nowadays on frameworks like Arcane, Dune, Feel++. Linear solver packages like PETSc or Trilinos are used to manage linear systems and provide access to a wide range of algorithms. With the evolution of High-Performance Computing, the variety of the hardware features available in new architectures has considerably increased. ARM processors, AMD, Intel and Nvidia GP-GPUs, TPU and FPGA devices are now common. To handle the induced complexity, different strategies are adopted in each linear solver framework. One of them consists in introducing a new layer that provides abstractions to manage the performance portability and to enable several parallel programming models.

In this paper, we evaluate the performance of linear solver packages that rely on tools like SYCL [16], Kokkos [8] or HARTS [11] to handle runtime systems like OpenMP, TBB, CUDA,.... A simulator to solve advection-diffusion problems has been developed with ALIEN, a C++ framework that provides a high level and unified API to handle large distributed matrices and vectors. We have benchmarked different solver algorithms, and have evaluated the efficiency of their implementations, and their capability to perform on different architectures, for instance, large number of cores, GP-GPU accelerators, or processors with large SIMD instructions.

1 Introduction

A wide variety of scientific and engineering applications rely on linear algebra algorithms. Complex physical phenomena such as porous media flow or heat transfer are described using partial derivative equation systems which require solving large sparse linear systems. For applications related to reservoir modeling or high energy physics simulations, most of the computing

time is spent performing linear algebra operations. Using an efficient solver is then mandatory to achieve performance.

The implementation of efficient algorithms for solving linear systems is a difficult task involving the knowledge of experts in applied mathematics and computer science. Fortunately, such algorithms are generally available in highly efficient libraries that developers can plug. However, selecting the best algorithm for a given problem remains, in practice, delicate as it requires handling different kinds of issues: i) the numerical methods complexity leads to linear systems that may be dense or sparse, with different structural properties of symmetry, positiveness, etc. ii) the complexity of algebraic algorithms which generally perform on a limited family of linear systems; iii) the computer science challenges of tuning data structures and algorithm implementations to fit the evolution of the hardware. It is more difficult for multi-physics applications that couple problems requiring different kinds of linear solvers. The interfacing task with different external solvers becomes even more complex when developers need to maintain different inconsistent and changing APIs when updating solver libraries.

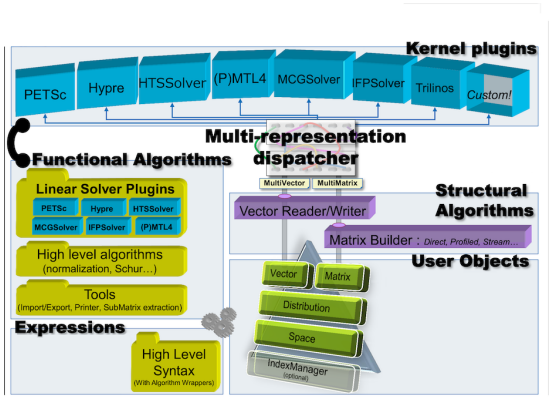
Alien is a C++ framework that provides a high level and unified API to handle large distributed matrices and vectors, perform algebraic operations and solve linear systems. This API has been designed to be accessible and comprehensible to numerical software developers hiding the complexity and the variability of the algebraic structures used in linear solvers packages. It differs from the traditional frameworks in its design based on lightweight structures that encapsulate multiple coexisting internal representations of algebraic objects. These representations are dedicated to the algebra operations or linear solver algorithms available through different linear solver packages or libraries. A mechanism is provided to manage efficiently and transparently conversion between the different object representations. A plugin mechanism based on converter objects ensures the extensibility of the framework to any kind of external linear solver library.

In section 2, we present the general issues met in scientific software development to solve large linear systems in a context where the research community in linear algebra is very active and where many libraries provide a large panel of efficient methods on recent hardware technologies. In section 3, we present Alien, a framework for linear algebra libraries, aiming at providing a flexible, easy to use API for linear algebra. It provides a plugin mechanism that enables users to easily extend the framework with new functionalities provided by any external solver library. In section 4, we study performance portability issues in different linear solver packages using tools like SYCL [16], Kokkos [8] or HARTS [11] providing abstractions above runtime systems like OpenMP, TBB, CUDA,... Finally in section 5, we illustrate the use of Alien evaluating the performance of popular linear solver implementations. We solve linear systems coming from the discretization of a heterogenous advection-diffusion problem. We study the scalability of the tested algorithms regarding the number of cores and GP-GPUs. We analyze the efficiency of the implementations using hybrid MPI and thread parallelism and their capability to take advantage of GP-GPUs or large SIMD instructions.

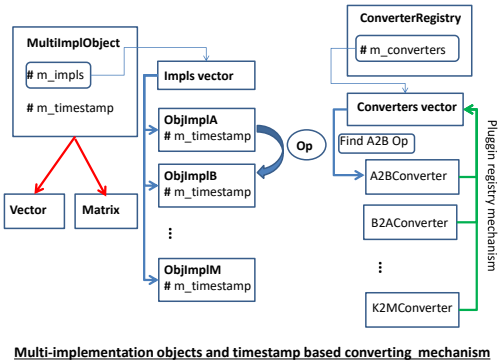
2 Issues with linear solver libraries in scientific software

In the development of numerical software, selecting the more relevant linear solver algorithm for a given problem is complex. For a chosen linear solver package, implementing efficiently

the plugin of an appropriate component may be a painful task. The taxonomy, used to classify linear systems and numerical methods, is large. Choosing the best and most performant linear solver algorithms and preconditioners requires accurate knowledge at different levels. It is important, from a numerical point of view, to well analyze the properties of linear systems. A background in computer science is also necessary to handle the different programming models used to implement efficient algorithms on the various target hardware platforms.
Considering, for instance, the discretization of partial differential equations, one of the major issues is to solve efficiently huge linear systems. While there is a variety of numerical discretization methods (Finite Difference, Finite Element, Finite Volume methods, etc.), it is well known that the structural properties of the matrices, inherited from the numerical characteristics of the used discretization methods, have a tremendous impact on the choice of the best algorithm to solve the linear problem.
At the PDE domain level, we may have problems classified as elliptic, parabolic or hyperbolic arising from advection-diffusion equation problems, we have Helmholtz problems as for the Maxwell equations or boundary-value PDE problem arising for electromagnetism problem. At a purely algebraic level, matrices may be classified regarding the symmetry of matrix structure, the sparsity of matrix graph entries, the positiveness of the systems, the M property feature of matrices, etc. The variety of algebraic methods to solve linear systems is also large. There are two main families of methods, direct methods and iterative methods. Within each of them, they can be classified regarding their robustness and their extensibility. In the large family of Krylov methods, preconditioner algorithms are also classified in subcategories like polynomial methods, factorization based methods, multi-level methods, etc.
Linear solver algorithms are available in different libraries which provide their own data structures with various formats (Dense, Compressed Sparse Row, Compressed Sparse Column, Ellpack, BlockEllpack, etc.) and various implementations to perform on different hardware platforms. Some of them, like PETSc and Trilinos [14], provide in addition to their own algorithm implementation, a generic interface to give access to numerous external linear solver libraries capabilities. For example, PETSc provides access to Hypre [9], SuperLU, ... and Trilinos to PETSc, Hypre, MUMPS, etc. Those libraries are a convenient way to have access to a large panel of solvers. Nevertheless they impose strict requirements on external package versions for compatibility reason. Therefore using the up to date version of a specific solver or plugging any in-house linear solver is often challenging as it requires a deep knowledge of libraries internal structures.
Since the research domain in linear solver methods is very active, mainly driven by the evolution of hardware that imposes to revisit algorithms, having access to different solver packages is important, at least for benchmark purpose to evaluate up to date methods. This is also important in the development of multi-physics models when coupling different physical models. The linear systems arising from each physical model may have different numerical properties. It may be then necessary within the same application, to have access to algorithms available in different libraries.



(a) Alien framework



(b) Multi-implementation mechanism

3 Alien an open generic and extensible linear algebra framework

Alien is a C++ framework that provides a high level and unified API to handle large distributed matrices and vectors, to perform BLAS 1 and 2 operations and to solve linear systems or eigenproblems. This framework differs from the previously cited generic frameworks in its design based on lightweight structures that encapsulate multiple coexisting internal representations of algebraic objects available through different linear solver packages or libraries.

Alien has been designed to provide an extensible API, related neither to a specific solver nor to specific data structures. Its extensibility is ensured by providing closed specialized objects that perform specific functionalities. Extending the library consists in providing new functional objects without impacting existing classes.

In Alien's design, high-level user functional features are separated from back-end data storage implementations. At the user level, operations can be performed without any knowledge of the underlying data structure implementations.

Alien architecture, represented in Figure 1a, is based on a *Core* layer composed of the *Matrix* and *Vector* classes representing algebraic objects. Useful concepts like *Space*, *Distribution* are provided to modelize global structural information shared by objects on which algebraic operations can be performed. The *Matrix* and *Vector* classes are multi-representation objects. They rely on the *Multi Representation* mechanism illustrated in Figure 1b. This mechanism is based on the *MultiImplObject* concepts, kind of handlers on a collection of different implementations of a same object. Each implementation object has a timestamp which is incremented on modification. *Converter* objects are helper tools that implement the conversion between one implementation to another. They can be registered in a *ConverterRegistry* manager by the means of a plugin mechanism. An automatic lazy conversion mechanism between implementations ensures that, when accessing a specific implementation, for instance *ObjImplB* in Figure 1b , the up-to-date version of the specific object is returned, regardless of how the algebraic object was created, filled or manipulated before. By the means of a timestamp manager, the version of *ObjImplB* is compared to the lastest up-to-date version, for instance *ObjImplA*. *ObjImplB* can be updated if necessary, by the means of the *A2BConverter Converter* object registred in the ConverterRegistry. The memory footprint of this mechanism is limited to the number of

implementations concurrently used. The overhead induced by conversion operations is reduced thanks to the timestamp mechanism. The cost of these operations depends on the implementation of the converter objects. It is usually equivalent to a few matrix-vector product operations. Matrices and vectors can be handled as a composition of submatrices and subvectors. This mechanism is useful for multi-physics simulations where the global linear systems might be assembled as sub-systems representing different models and their coupling interactions. The *CompositeMatrix* and *CompositeVectors* classes modelize the composition of sub-matrices and sub-vectors objects that can be addressed independently as *Matrix* and *Vector* objects.
At the API level, data structures are not manipulated directly. Some *Builders* and *Accessors* objects are provided to write and read linear algebra objects. This enables to decouple, for example, the assembly of a matrix to the underlying data structure of a matrix. Assembly operations, writing a right-hand side, e.g. are generic operations and do not depend on the implementation. Using specialized objects that carry the knowledge on the data structure strives towards several objectives: i) the simulation code is written without any knowledge of the data structure; ii) linear systems can be built directly in the expected data structure for one solver, or in any data structure then converted to the right data structure; iii) changing the data structure of the linear system corresponds to change the builder object; iv) several building strategies can be used (keeping the profile or not for example), with different specific builder objects.

Listing 1: Matrices

```
auto pm = Env::parallelMng();
auto s  = Space(10,"MySpace");
auto md = MatrixDistribution(s,s,pm);
auto A  = Matrix(md);
{
  auto builder =
   DirectMatrixBuilder(A,eResetValues);
  builder.reserve(30) ;
  builder.allocate();
  for(Integer row=0;row<10;++i)
  {
      builder(row,row) = 2.;
      if(row+1<10)
        builder(row,row+1) = -1.;
      if(row-1>=0)
        builder(row,row-1) = -1.;
  }
}
```

Listing 2: Vectors

```
auto pm = Env::parallelMng();
auto s  = Space(10,"MySpace");
auto vd = VectorDistribution(s,pm);
auto x  = Vector(vd);
{
  auto writer = LocalVectorWriter(x);
  for(Integer i=0;i<10;++i)
      writer[i] = 1.;
}
```

The Alien API provides *LinearAlgebra* objects implementing the BLAS 1 and BLAS 2 functionalities with a Lapack, ScaLapack-like API. A mechanism of expressions similar to the one provided by the Eigen library is also available to perform algebraic operations as unary or binary operators on Matrix and Vector objects. Listing 3 illustrates how to perform simple algebraic operations with matrices and vectors with Alien API.

The Alien API provides some interface classes like *ILinearSolver* and *IEigenSolver* to solve linear systems or eigenvalue problems. Listing 4 illustrates how to solve linear systems or eigenvalue problems with Alien API.

To extend Alien with external linear solver packages, a plugin mechanism is provided. The mechanism is based on some common interface functions to implement, some converter objects that enable the conversion of data structures from one to another implementation, and a registry

Listing 3: Algebraic expressions

```
auto pm = Env::parallelMng() ;
auto s  = Space(10,"MySpace");
auto md = MatrixDistribution(s,s,pm);
auto vd = VectorDistribution(s,pm);
auto A  = Matrix(md);
auto x  = Vector(vd);
auto y  = Vector(vd);
auto r  = Vector(vd);
Real lambda = 0.5 ;
r = y-A*x ;
y = y+(lambda*r) ;
x = A*(lambda*y) ;
```

Listing 4: Linear solvers

```
auto pm = Env::parallelMng() ;
auto s  = Space(10,"MySpace");
auto md = MatrixDistribution(s,s,pm);
auto vd = VectorDistribution(s,pm);
auto A      = Matrix(md);
auto x      = Vector(vd);
auto y      = Vector(vd);
auto r      = Vector(vd);

  auto solver=createSolver(/*...*/) ;
  auto status = solver->solve(A,x,y);
  if(status.succeed) {
    r = y - A* x ;
    cout<<"res2 = "<<dot(r,r)<<endl ;
  }
```

Listing 5: Uzawa method

```
bool solveUzawaMethod(ILinearSolver* solver,
                      Real omega,
                      int nb_iterations,
                      CompositeMatrix const& matrix,
                      CompositeVector const& b,
                      CompositeVector& xk) {
  /*  [  A B ]  [u]    [f]
      [ tB 0 ]  [p]    [g]  */
  Matrix const& A  = matrix(0,0); //subMat M[0,0]
  Matrix const& B  = matrix(0,1); //subMat M[1,1]
  Matrix const& tB = matrix(1,0); //subMat M[1,0]
  Vector const& f  = b(0);        //subVec B[0]
  Vector const& g  = b(1);        //subVec B[1]
  Vector& uk = xk(0) ;
  Vector& pk = xk(1) ;

  Vector ru(A.rowSpace().distribution()) ;
  Vector rp(tB.rowSpace().distribution()) ;

  for(int k=0;k<nb_iterations;++k) {
    //Update velocity
    ru = f - B*pk ;
    solver->solve(A,ru,uk) ;

    auto status = solver->getStatus();
    if(!status.succeeded) return false ;

    rp = g - tB*uk ;
    //Update pressure
    pk = pk - omega*rp ;
  }
  return true ;
}
```

mechanism that enables the end-user to select the desired functionalities and their specific implementations. Any external library can then be plugged in and made available as a module in the Alien framework. Each implementation is identified by a unique string key value and an associated tag type value. The Core of Alien provides a few intermediate backends, *CSR* or *DOK*, usually used to write converters between different backends.

The Core of Alien provides also some generic implementations of a few Krylov solver algorithms (CG, BiCGStab) and preconditioners (ILU0, Chebychev, Neunman, ILUFP). These implementations depend only on the abstract backend matrix, the vector structures and their associated LinearAlgebra objects implementing matrix-vector operations. These algorithms can be instantiated for instance with a parallel MPI based CSR linear algebra implementation for CPU, and a SYCL based implementation for Nvidia, Intel or AMD GP-GPU. More details can be found in the following section 4.

In the following example, we illustrate the use of Alien implementing the Uzawa algorithm to solve the Stokes problem 1 which leads to a saddle point linear system. The Stokes problems consists in finding $u, p(\mathbf{x})$ $\mathbf{x} \in \Omega$:

$$\begin{cases} \alpha\nabla^2\mathbf{u} + \nabla p = f & \text{in } \Omega \\ \nabla \cdot \mathbf{u} = 0 & \text{in } \Omega \\ \mathbf{u} = \mathbf{g} & \text{on } \partial\Omega_d, \\ \partial_n\mathbf{u} = \mathbf{h} & \text{on } \partial\Omega_n, \end{cases} \tag{1}$$

The discretization of Equation 1 with classical numerical schemes, positioning the degrees of freedom for the velocity and the pressure on geometrical entities of different types (for instance

velocity on faces or pressure on cells) leads to a saddle point system with a symmetric indefinite matrix as follows:

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix} \tag{2}$$

with $A \in \mathbb{R}^{n \times n}$ definite on the kernel of $B \in \mathbb{R}^{m \times n}$ $m < n$.

This saddle point linear system can be solved with the iterative Uzawa in algorithm. Listing 5 illustrates how this algorithm can be implemented with Alien.

4 Performance portability layers in linear solver packages

Nowadays, the diversity in computer architectures for HPC is very large with a wide range of processor and memory technologies, including GPUs, Many-Core Processors, ARM, FPGAs, and ASICs, as well as new memory technologies High-Bandwidth Memory (HBM), Non-Volatile Memory (NVRAM). To handle seamlessly at the software level the coming issues, performance portability tools have been developed for a few years to manage in a unified way parallelism, heterogeneous and multi level memory in order to write portable parallel algorithms. These tools provide abstractions such as: *MemorySpace* models to address Single/Multi Node, LocalHost memory or Remote memory, *ExecutionSpace* models to deal with serial execution, multi-threads execution with OpenMP, Posix, TBB,... on standard processors or Cuda, OpenCL, HIP on accelerators devices (NVidia, AMD, ...) They enable writing generic parallel algorithms with lambda functions using the task programming paradigm. They provide *Parallel Loop* concepts and *Generic Parallel Collections* to hide the specificities of the various available runtime systems. Among these tools, we can cite Kokkos [8], Raja [4], SYCL [16], HARTS [11].

Kokkos [8] is a modern C++ framework developped at Sandia National Laboratories providing a model of compute node as a collection of abstractions modeling memory and execution spaces. It enables to separate the parallel programming paradigm to write efficient algorithms with C++ functors or lambda expressions, from the execution level on specific back-end. Some array abstractions model the specificities of memory in a unified way. Generic parallel algorithms such as *parallel_for*, *parallel_scan*, *parallel_reduce* provide dispatch mechanisms to take advantage of multiple runtime systems like OpenMP, TBB, Cuda, ... regarding the specificities of memory and processing units. In the Trilinos linear solver package, algebraic data structures, provided by the Tpetra package are based on the Kokkos framework. The parallel algorithms can then be written at a high level with the solver and preconditioner packages, and executed with different runtime systems, like OpenMP on many-core architectures or Cuda on Nvidia GP-GPU.

SYCL [16], from the Khronos group, is a standard for heterogeneous computing, originally proposed as a single source C++ approach of OpenCL programmming. The concepts provided by SYCL are very similar to the ones of Kokkos. Computing nodes are modelled as a collection of devices. Parallel algorithms are written with lambda expressions that can be submitted to work queues. The concepts provided to handle memory are different. They rely on buffer objects and different kinds of accessors with specific intents (read, write,...) to handle the different type of memories. This safer memory access mechanism enables the runtime systems to create a dependency graph of buffer accesses, to better manage automatically data movement walking along theses graphs. In the Core of Alien, a SYCL backend with Block Ellpack matrix structure

is provided. The *SYCLLinearAlgebra* provides efficient BLAS operations implementations on a large variety of GP-GPU. They have been validated using the HipSYCL framework [1] that enables to handle Nvidia GP-GPU with Cuda and AMD GP-GPU with HIP.

HARTS [11], yet another runtime system layer, provides abstractions to model memory spaces, execution spaces and tools to write algorithms as graph of tasks that can be executed in parallel with various underlying runtime systems. In the HTSSolver linear framework, MCKernel algebra provides a generic API to perform parallel BLAS 1,2 operations, matrix-vector operations independently of the underlying data-structures and dispatch mechanisms to choose the more efficient data-structures regarding specific optimisations like AVX512 instructions,...

5 Benchmark of preconditioners provided in various popular linear solver packages

In this section, we evaluate the performance of popular preconditioners available in different linear solver packages. The advection-diffusion problem 3 is discretized with a classical finite volume two point flux approximation on the unit cube Ω. Find $u(\mathbf{x})$ $\mathbf{x} \in \Omega$:

$$\begin{cases} v = -\kappa \nabla u & \text{in } \Omega, \\ \nabla \cdot (-\kappa \nabla u) = f & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega_d, \\ \partial_n u = f & \text{on } \partial\Omega_n, \end{cases} \tag{3}$$

We use a $Nx \times Ny \times Nz$ structured regular grid discretization of Ω and a heterogeneous diffusion tensor κ with the following values

$$\kappa(x, y, z) = \kappa_0 e^{-\frac{\alpha}{2}(1+sin(2\pi\frac{x}{L_x}))(1+sin(2\pi\frac{y}{L_y}))}.$$

We have plugged in Alien the in-house IFPEN solver libraries MCGSolver [2,3] and HTSSolver [12, 13], and the popular open-source solver packages PETSc version 3.10, Trilinos version 13.2 and Hypre version 2.24. The framework is compiled with GCC 8.3 with the OpenMP support and activating AVX2 vector instructions. The main solver packages features are described in Table 1.

We solve the linear system coming from the discretization of problem 3 ($\alpha = 10, L_x = 0.125, L_y = 0.25$) with a preconditioned BiCGStab solver using a 10^{-8} stop criteria tolerance parameter. We compare the performance of the standard relaxation (SSOR), Chebyshev polynomial of degree 1, ILU preconditioners and different multi-level algorithms like Algebraic Multi-Grids solvers (Hypre BoomerAMG [10], Trilinos MueLu solver [15], NVIDIA AmgX solver [6]). For the ILU preconditioner, we use the classical ILU0 algorithm and the iterative FILU variant proposed in [5] and provided in the ShyLU package of Trilinos and the ILUFP option of HTSSolver. In our experiments we use the FILU preconditioner with 3 iterations for triangular resolutions and 15 iterations for the factorization. For the AMG preconditioner options, we select the PMIS parameter for the coarsening algorithm of BoomerAMG while Aggregation is used with MueLu and AmgX. The matrix hierarchy level max is set to 25. We use Gauss-Seidel as smoother for BoomerAMG and MueLu, and Jacobi smoother for AmgX. A V cycle is used with BoomerAMG and a W cycle for MueLu and AmgX. We select the default values for the other parameters.

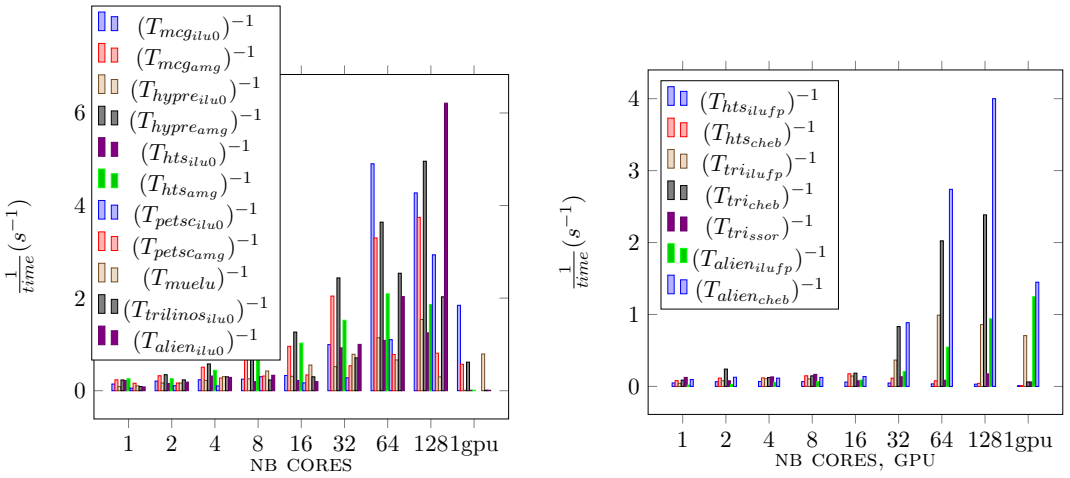
Packages	Hypre	PETSc	Trilinos	MCGSolver	HTSSolver	AlienCore
Version	2.4	3.7	13.2	2.1.0	2.0.0	1.1.4
Language	C	C	C++	C++	C++	C++
MPI	yes	yes	yes	yes	yes	yes
Threads	OpenMP	OpenMP	OpenMP TBB, Posix	OpenMP Posix	OpenMP TBB, Posix	no
AVX512			yes	yes	yes	
Cuda	yes	yes	yes	yes		SYCL
Direct Solver		SuperLU MUMPS	KLU2		SuperLU MUMPS	no no
Krylov	CG,BiCG GMRES	CG,BiCG GMRES	CG,BiCG GMRES	CG,Piepline CG, BiCG, GMRES	BiCG	CG, BiCG
Poly			Chebyshev	Neumann	Chebyshev Neumann	Chebychev Neumann
Relaxation		Jacobi	GS, SymGS	Jacobi, GS		no
ILU		ILU(k,t)	ILU(k,t) FAST_ILU	ILU(k), ILU(0) ILUFP	ILU(0) ILU0PF	ILU(0) ILUFP
AMG	BoomerAMG	BoomerAMG AmgX	MueLu AmgX	BoomerAMG AmgX	BoomerAMG	no

Table 1: External linear solver features

We study the numerical robustness of the tested algorithms and their scalability regarding the number of cores and GP-GPUs. The experiments have been run on Topaze, a multi-node Linux cluster of the CCRT, the French Computing Centre for Research and Technology. This cluster is composed of 6 144 dual-socket Nodes with 2x64-cores AMD Milan at 2.45GHz, 256Go per node, 2Go per core. Some of these nodes are equipped of GP-GPU Nvidia Ampere A100 The benchmark is realized with the following hardware configurations: full Mpi, full threads (OpenMP), and hybrid MPI + Cuda with one Mpi process and one GPU. We realize the experiments on a single node of the cluster with linear systems of 10^6 rows obtained using a $100 \times 100 \times 100$ regular mesh. The elapsed time T in seconds of the full resolution and the number of iterations at convergence N_{iter} are gathered in table 2a. The throughputs, given by the inverse $\frac{1}{T}$ of the resolution times, are compared in Figure 2a.

The analysis of Figure 2a shows that within one node, the best performances are obtained using GP-GPU version of the AMG preconditioner available with the AmgX library provided by both in house solver packages MCGSolver and the MueLu package of Trilinos. With configuration using only CPUs, the best performance are obtained with the BoomerAMG algorithm provided by our in house solver packages MCGSolver and HTSSolver.

The Trilinos and HTSSolver framework are based on, respectively Kokkos [7] and the HARTS layer [11] that enable to design data structures and to write algorithms with high level abstractions that encapsulate various parallel paradigms. So when algorithms are written with those abstractions, some dispatch mechanisms enable to handle in a transparent way parallelism on distributed memory with Mpi, parallelism on shared memory with threads using either OpenMP, Posix threads, or TBB, and parallelism on GP-GPU accelerator devices with Cuda. We evaluate these performance portability features, benchmarking the FILU, Chebyshev polynomial and SOR preconditioners on one node with the following hardware configurations denoted *mpi*, *omp* and *gpu* for respectively full Mpi, full threads OpenMP, one Mpi process and one GP-GPU. These preconditioners are interesting despite their lack of robustness regarding AMG preconditioners for their good smoothing properties in multi-level algorithms. We have



(a) Single node

(b) Performance portability

Figure 2: Throughput performance results for a $100 \times 100 \times 100$ test case. Higher is better.

gathered the elapsed times and iteration numbers results in Table 2b.

The analysis of the results shows that the best performances are usually obtained either with one MPI process and a GP-GPU either with the full Mpi configuration with 64 or 128 cores. It shows also the interest of the hybrid MPI+X configuration which enables to fully take advantage of computing power offered by the GP-GPUs and extra available cores. When a main application is run with a small number of MPI processes per node, the performance of the solvers can then be increased using the extra available cores of the many-core processors with threads. The analysis of the results in figure 2a shows that the best performances are obtained with AMG preconditioners performed on 128 cores with the MPI configuration. The performances obtained using 1 MPI process and a GP-GPU are equivalent to those performed on 64 cores with the MPI configuration. In figure 2b and table 2b, we can notice that single source code algorithm's implementation based on a performance portability layer, do not present really any overhead regarding their equivalent specific hand written implementations in packages like PETSc and MCGSolver. The performance of GP-GPU implementations based on Kokkos or SYCL are quite the same.

6 Conclusion

In this paper, we have presented Alien, a C++ framework that provides a high level and unified API to handle large distributed matrices and vectors, to perform algebraic operations and to solve linear systems or eigenproblems. We have detailed its design based on lightweight structures that encapsulate multiple coexisting internal representations of algebraic objects available through different linear solver packages or libraries.

We have illustrated the capability of the framework to write at a high level algebraic algorithms. Thanks to a plugin mechanism, the framework has been easily extended with other in-house and external software packages. We have performed a benchmark of a large variety of linear solver algorithms with different hardware configurations. We have studied the perfor-

Package	MCGSolver		HTSSolver		PETSc		Trilinos		Hypre	
Precond	ILU0	AMG	ILU0	AMG	ILU0	AMG	ILU0	MueLu	ILU0	AMG
Nb Cores	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T
1	186/6.933	7/4.25	90/4.51	3/3.87	387/17.9	5/6.18	172/11.0	13/9.46	125/11.2	3/4.29
2	205/4.792	7/3.08	194/6.40	3/3.85	351/9.24	5/6.03	92/4.26	12/5.99	66/5.95	3/2.89
4	212/4.227	7/1.96	102/3.16	4/2.26	463/9.68	5/3.66	126/3.32	9/3.29	65/4.48	4/1.72
8	197/4.011	8/1.42	191/5.11	3/1.43	151/3.31	6/3.15	207/4.30	8/2.34	57/3.89	4/1.15
16	198/3.043	8/1.04	194/4.54	3/0.97	346/5.90	7/2.92	198/3.32	7/1.80	58/3.21	4/0.78
32	218/1.005	8/0.48	106/1.08	3/0.65	660/3.58	7/1.85	229/1.41	7/1.26	85/1.92	5/0.41
64	222/0.204	8/0.30	194/0.92	3/0.47	688/0.90	7/1.27	107/0.39	10/1.50	58/0.87	4/0.27
128	240/0.234	8/0.26	196/0.80	3/0.53	387/0.34	6/1.23	241/0.49	6/3.33	72/0.65	4/0.20

(a) Single-node results : ILU0 and AMG preconditioners

Package	MCGSOLVER		HTSSolver			Trilinos			AlienCore	
Precond	ILU0	ILUFP	ILU0	ILUFP	Chebychev	ILUFP	Chebychev	SSOR	ILUFP	Chebychev
Nb Cores	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T	N_{iter}/T
1 mpi	186/6.93	189/25.7	90 /4.51	287/20.1	600/12.1	235/22.8	646/13.8	67/6.46	294/53.9	555/10.3
gpu	309/0.54	-	-	-	-	239/1.41	511:0.54	143/15.9	269/0.80	626/0.69
2 mpi	205/4.79	214/16.0	194/6.40	282/15.8	464/8.01	-	570/8.63	70/4.21	304/35.6	542/7.81
omp	323/7.47	224	17.0	190/6.23	286/23.6	532/9.59	238/12.6	282/4.12	143/12.6	-
4 mpi	212/4.22	206/11.0	102/3.16	300/15.8	604/8.97	-	599/8.19	81/3.50	262/18.0	625/8.63
omp	283/5.49	213/11.8	103/4.01	292/23.6	537/9.59	224/9.28	648/8.10	87/7.53	-	-
8 mpi	197/4.01	198/11.0	191/5.11	274/15.1	550/8.40	-	604/8.62	89/5.02	267/14.8	536/7.94
omp	321/3.24	264/7.45	174/5.94	270/18.8	536/6.74	228/9.56	527/6.78	69/5.96	-	-
16 mpi	198/3.04	207/7.51	194/4.54	276/10.5	417/4.51	-	613/6.56	69/1.76	275/11.3	588/7.25
omp	221/2.14	224/4.77	204/7.29	286/15.7	526/8.70	238/6.87	597/5.41	143/12.8	-	-
32 mpi	218/1.00	228/1.67	106/1.08	275/3.23	600/2.13	-	534/1.68	92/0.46	332/4.86	328/1.12
omp	358/1.98	302/2.97	205/9.93	295/17.6	528/9.49	225/2.72	530/1.20	86/7.46	-	-
64 mpi	222/0.20	229/0.63	194/0.92	294/1.54	592/0.71	-	531/0.60	74/0.22	283/1.83	564/0.36
omp	350/0.37	249/0.72	102/9.02	275/8.28	372/12.7	240/1.00	671/0.49	135/11.5	-	-
128 mpi	24/0.23	243/0.37	196/0.80	283/1.31	594/0.51	-	271/0.63	155/0.86	276/1.04	572/0.25
omp	385/0.32	271/0.52	122/17.4	292/8.98	531/23.8	229/1.16	563/0.41	67/5.74	-	-

(b) Performance portability results : ILU0, ILUFP, Cheb, SSOR preconditioners

Table 2: Performance results

mance of algorithms written with abstractions to hide the underlying hybrid MPI+X parallelism with OpenMP for threads and Cuda for GP-GPU.

Alien, part of an open-source project is available at :
`https://github.com/arcaneframework/alien`

REFERENCES

[1] Alpay, A., H.V.: Sycl beyond opencl: The architecture, current state and future direction of hipsycl. in proceedings of the international workshop on opencl

[2] Anciaux-Sedrakian, A., Eaton, J., Gratien, J., Guignon, T., Havé, P., Preux, C., Ricois, O.: Will gpgpus be finally a credible solution for industrial reservoir simulators? In: SPE Reservoir Simulation Symposium. OnePetro (2015)

[3] Anciaux-Sedrakian, A., Gottschling, P., Gratien, J., Guignon, T.: Survey on efficient linear solvers for porous media flow models on recent hardware architectures. Oil Gas Sci. Technol. – Rev. IFP Energies nouvelles Volume 69, Number 4,Dossier: Geosciences Numerical Methods (2014)

[4] Beckingsale, D., Hornung, R., Scogland, T., Vargas, A.: Performance portable c++ programming with raja. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. pp. 455–456 (2019)

[5] Chow, E., Patel, A.: Fine-grained parallel incomplete lu factorization. SIAM journal on Scientific Computing **37**(2), C169–C193 (2015)

[6] Eaton, J.: Gpu-accelerated algebraic multigrid for commercial applications. Manager NVAMG CUDA Library. NVIDIA (2013)

[7] Edwards, H.C., Sunderland, D., Porter, V., Amsler, C., Mish, S.: Manycore performance-portability: Kokkos multidimensional array library. Scientific Programming **20**(2), 89–114 (2012)

[8] Edwards, H.C., Trott, C.R.: Kokkos: Enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (xsw 2013). pp. 18–24. IEEE (2013)

[9] Falgout, R.D., Yang, U.M.: hypre: A Library of High Performance Preconditioners. In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (eds.) Computational Science ICCS 2002. pp. 632–641. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2002)

[10] Falgout, R.D., Yang, U.M.: hypre: A library of high performance preconditioners. In: International Conference on Computational Science. pp. 632–641. Springer (2002)

[11] Gratien, J.M.: An abstract object oriented runtime system for heterogeneous parallel architecture. In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. pp. 1203–1212. IEEE (2013)

[12] Gratien, J.M.: A robust and scalable multi-level domain decomposition preconditioner for multi-core architecture with large number of cores. Journal of Computational and Applied Mathematics p. 112614 (2019)

[13] Gratien, J.M.: Introducing multi-level parallelism, at coarse, fine and instruction level to enhance the performance of iterative solvers for large sparse linear systems on multi- and many-core architecture. In: 2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar). pp. 85–95. IEEE (2020)

[14] Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A.: An Overview of the Trilinos Project. ACM Transactions on Mathematical Software **31**(3), 397–423 (2005). https://doi.org/10.1145/1089014.1089021

[15] Hu, J.J., Prokopenko, A.: Muelu: Multigrid framework for advanced architectures. Tech. rep., Sandia National Lab.(SNL-CA), Livermore, CA (United States); Sandia National ... (2015)

[16] Keryell, R., Reyes, R., Howes, L.: Khronos sycl for opencl: a tutorial. In: Proceedings of the 3rd International Workshop on OpenCL. pp. 1–1 (2015)