Oil & Gas Science and Technology – Rev. IFP Energies nouvelles (2016) **71**, 65
© A. Roussel et al., published by IFP Energies nouvelles, 2016
DOI: 10.2516/ogst/2016020

OPEN ACCESS

Dossier

SimRace 2015: Numerical Methods and High Performance Computing for Industrial Fluid Flows
SimRace 2015 : Méthodes numériques et calcul haute performance pour la simulation d'écoulements complexes

Using Runtime Systems Tools to Implement Efficient Preconditioners for Heterogeneous Architectures

Adrien Roussel[1,2,*], Jean-Marc Gratien[1] and Thierry Gautier[2]

[1] *IFP Energies nouvelles, 1-4 avenue de Bois-Préau, 92852 Rueil-Malmaison Cedex - France*
[2] *Inria Grenoble - Rhône-Alpes, 655 avenue de l'Europe, CS 90051, 38334 Montbonnot Cedex - France*
e-mail: adrien.roussel@ifpen.fr

* Corresponding author

Abstract — *Solving large sparse linear systems is a time-consuming step in basin modeling or reservoir simulation. The choice of a robust preconditioner strongly impact the performance of the overall simulation. Heterogeneous architectures based on General Purpose computing on Graphic Processing Units (GPGPU) or many-core architectures introduce programming challenges which can be managed in a transparent way for developer with the use of runtime systems. Nevertheless, algorithms need to be well suited for these massively parallel architectures. In this paper, we present preconditioning techniques which enable to take advantage of emerging architectures. We also present our task-based implementations through the use of the HARTS (Heterogeneous Abstract RunTime System) runtime system, which aims to manage the recent architectures. We focus on two preconditoners. The first is ILU(0) preconditioner implemented on distributing memory systems. The second one is a multi-level domain decomposition method implemented on a shared-memory system. Obtained results are then presented on corresponding architectures, which open the way to discuss on the scalability of such methods according to numerical performances while keeping in mind that the next step is to propose a massively parallel implementations of these techniques.*

Résumé — **Utilisation de moteurs exécutifs pour une implémentation efficace de préconditionneurs sur architectures hétérogènes** — La résolution de grands systèmes linéaires creux est une étape coûteuse de la modélisation de bassin ou la simulation de réservoir, et le choix de préconditionneurs robustes impacte fortement la performance de ceux-ci. Les architectures hétérogènes à base d'architectures de type *General Purpose computing on Graphic Processing Units* (GPGPU) ou many-cœurs introduisent de nouveaux challenges de programmation qui peuvent être gérés de manière transparente pour le développeur grâce à l'utilisation de moteurs exécutifs. Il faut néanmoins que les algorithmes soient adaptés à ce type d'architectures massivement parallèles. Nous présentons dans cet article les méthodes de préconditionnement permettant de tirer au maximum avantage des architectures émergentes. Nous présentons également une implémentation de celles-ci avec le moteur exécutif HARTS (*Heterogeneous Abstract RunTime System*) conçu pour gérer de manière efficace les architectures hétérogènes. Nous détaillons les implémentations de deux préconditionneurs. L'un est un préconditionneur ILU(0) porté sur architecture distribuée. L'autre est une méthode de décomposition de domaines multi-niveaux implémentée sur architecture à mémoire partagée. Après avoir détaillé ces deux préconditionneurs, une étude sur la portabilité de telles méthodes et leur scalabilité sur différents modèles d'architectures sera menée.

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

INTRODUCTION

In basin modeling or reservoir simulations, multiphase porous media flow models lead to solve complex non-linear Partial Differential Equations (PDE) systems. These PDE are discretized following a Finite Volume (FV) scheme which lead to a non-linear system solved with an iterative Newton solver. At each Newton step, the system is linearized and then solved with an iterative method such as BiConjugate Gradient Stabilized (BiCGStab) [1] or Generalized Minimal RESidual (GMRES) [1] algorithms, well suited for large sparse and unstructured systems. This resolution part is the most time consuming part and represents 60% to 80% of the global simulation time. Simulator's performance depends on the efficiency of the linear solver. As the cost of such iterative methods strongly depends on the number of iterations required to converge, the choice of a robust preconditioner is important.

In High Performance Computing (HPC), hardware technologies advances led to a popularization of multi-core architectures enhanced by accelerators such as General Purpose computing on Graphic Processing Units (GPGPU) or many-core architectures. Taking advantage of this kind of computer nodes introduces new challenges in terms of data locality, memory coherency, load balancing between computational units and scheduling. To deal with it, various approaches appeared to manage different levels of parallelism: programming models, programming environments, schedulers, data management solution or runtime systems. Runtime systems represent a layer between hardware and application in the way that a developer does not need to consider the hardware layer to program efficiently. It enables to manage hardware architecture and take advantage of different kind of technologies that compose such systems. HARTS (Heterogeneous Abstract RunTime System) [2] is a runtime system providing abstract concepts to distribute tasks on available computational units in an efficient way. Its purpose is to take advantage of heterogeneous architectures.

Whereas different algorithms exist for multi-core architectures, programming efficiently for heterogeneous architectures requires much more effort. A good preconditioner must reduce significantly the conditioning of the matrix, and to be easy to apply. Considering new challenges, the extraction of even more parallelism of algorithms is essential to program efficiently massively parallel architectures. In this way, some preconditioning techniques present interesting characteristics because of their natural partitioning. Multi-Level Domain Decomposition (DDML) methods present such properties. Due to their parallel nature, such a method can be split in independent coarse grained tasks in which we can introduce fine grained tasks to obtain different levels of parallelism. Contrary to Incomplete LU Factorization with no-fill, ILU(0), which require much more effort in order to reorganize the algorithm's structure to be efficient on parallel machines.

Anciaux-Sedrakian *et al*. [3] review some of existing preconditioning techniques on recent massively parallel architectures by using GPGPU features. In this paper, we extend it and we focus on scalability on such methods according to their numerical performances and on their implementations with HARTS runtime system. A task-based implementation of preconditioners and a solver is presented in this paper to provide a convenient way to distribute work among processing units. We show the importance of the choice of a preconditioner according to the underlying architecture. We compare two distinct preconditioning techniques and study their scalability taking advantage of different levels of parallelism and memory, but also their numerical performances by increasing the number of partitions. We limit our study on a quite moderate number of cores to take fully advantage of a computational node before going on a next level of parallelism.

In the first section, we review hardware technologies and give an detailed overview of HARTS which provide significant performances on parallel architectures with task-based algorithms.

The second section is dedicated to data partitioning taking advantage of different levels of memory.

An abstract linear algebra Application Programming Interface (API) implemented with various technologies in a transparent way for users is presented in the third section.

The fourth section is focused on preconditioners and their parallel algorithms.

Results according to different programming models are gathered and analyzed in the fifth section.

We conclude this work in the last section, and give some perspectives for the future on heterogeneous computing nodes.

1 HARDWARE CONTEXT

Computer architectures are increasingly complex, and applications that would follow the hardware evolution has to adapt their algorithms to extract even more parallelism. In this section, we will review the main features of recent hardware technologies and how it is possible to take advantage of them. We will first remind the main characteristics of heterogeneous architectures and the importance to adapt algorithms to be well-suited for gain in scalability. We then explain one of the available ways to program efficiently such architectures with the use of runtime systems.

1.1 Massively Parallel Architectures

Current computer architectures are mostly hierarchical. Nowadays, clusters are composed of several nodes which

contain several sockets where can be found multi-core processors and accelerators. It involves a memory hierarchy, and the way of binding cores should be determinant, specifically when it involves communication, synchronization.

Parallelism can be expressed at a coarse grained level between nodes, and at a finer grained size all cores of a node can cooperate to achieve tasks by the use of shared-level parallelism. Moreover, the democratization of heterogeneous architectures highlights that data locality consideration grows and it is now applied to I/O devices such as accelerators or network interfaces [4].

Data transfers between local memories have a cost and could be a bottleneck for memory bound applications. In that case, it is important to minimize them by reusing data when possible while maintaining memory coherency. The distribution of work between available computing units and the load balance are also some key issues. There is no unique solution to schedule pieces of work, and each of them may be more or less adapted to each specific architecture.

Both of accelerators' architectures present interesting features for HPC, but impose to applications to extract more parallelism than multi-core architectures. For instance, the Simple Instruction Multiple Threads (SIMT) model of GPU accelerators enables application to process the same instruction on many concurrent threads, so we need to process a large amount of data to benefit of this feature. Parallel applications are not necessary ready to take advantage of massively parallel architectures so they need to be modified according to new imposed challenges.

Programming efficiently such architectures becomes more and more complex for application's developers. Various approaches emerged to help them and to manage this complexity. Programming models like CUDA [5], OpenCL [6] enable to develop applications for accelerators. OpenMP [7] programming standard shows its interest since the version 4.0 by enabling computational offloading on accelerators by the use of specific directives.

1.2 Heterogeneous Abstract RunTime System (HARTS)

The popularization of runtime systems such as X-Kaapi [8], StarPU [9] or OmpSs [10] have proven their efficiency on heterogeneous architectures. Among them, HARTS [2] relies on abstract concepts between application and hardware to distribute and manage the work flow and the associated data movements. We present in this section, HARTS' key features and give some details of the underlying abstract concepts.

1.2.1 Hardware Topology for Heterogeneous Architectures

Thanks to the Hardware Locality software Hwloc [11], HARTS has a global view of hardware components. Figure 1 illustrates a hardware topology obtained from a computational node with two sockets which each one contains a octo-core processor with 16 Gb of memory. In such an architecture, data access time depends on memory location so the illustrated architectures is composed of two NUMA (stands for Non Uniform Memory Access) nodes. Static information, which can be used at compile time to generate the appropriate algorithms with the right optimization, lead to a tree generation, where each node is a hardware element. Dynamic information can be used during run time for low-level features and optimization. It is represented as argument of a node. It enables to instantiate algorithms with dynamic optimization parameters.

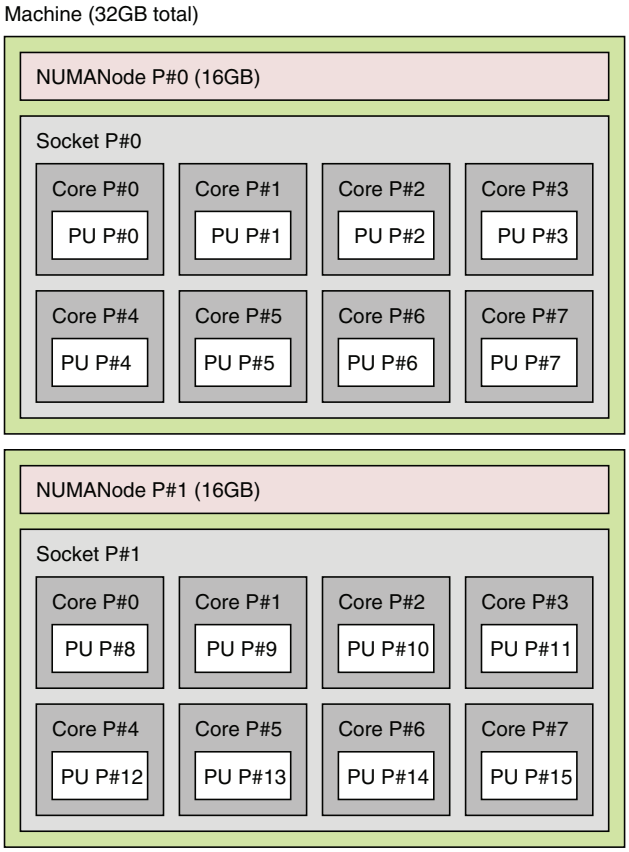


Figure 1

Hwloc illustration on a dual-socket machine.

1.2.2 Task-Based Algorithms and Task Management

Task-Based Algorithms

The work unit in HARTS is the task, which represents a small piece of work. It is a function call without any return statement except through its effective parameter. Tasks are

stored in a task pool on which a thread can push or extract them. When the pool is empty and all threads finish their work, the execution stops. HARTS enables tasks to have multiple representations depending on which hardware resources will execute it. New children tasks can be created during task's execution and can be inserted to the task pool.

Tasks are organized in a centralized task manager which maintains the centralized collection of created tasks. Each task is associated to an unique identifier. The centralized task pool uses a public task pool which can be operate by all the working threads, where each of them has a list of tasks' ids they have to process.

Data Flow Paradigm, Direct Acyclic Graph (DAG)

The work flow of an application can be described as a Direct Acyclic Graph (DAG) and then processed by HARTS, which manage the dependencies between tasks. Such dependencies may come from the data flow graph analysis according to task input and output data and their associated access modes. Dependencies can also be expressed explicitly at tasks' creation. A DAG is represented by a collection of root tasks. The addchild method help to create relationship dependencies between tasks. Walking along the graph consists in iterating on each task and on its children.

1.2.3 Data Management

Manipulated data in a task need a particular attention in their management. With HARTS, pieces of data manipulated by task objects are encapsulated in DataHandler objects, managed by a centralized data manager, with an unique identifier. A piece of data is describe following its access mode (Read, Write, ReadWrite). The data management layer provides also tools to split data with partitioners and to manipulate part of these data with partial views of them.

1.2.4 Scheduling Strategies

Scheduling strategy refers to the way of work distribution among available computation units. Even if it exists no unique solution to efficiently schedule tasks, the main goal remains to obtain an optimal work load for each threads.

Until now, implemented schedulers are static. It imposes a good weight estimation of the work flow, due to the static distribution of work among the computational resources. At execution time, roots of different DAG are given to the scheduler to dispatch tasks and balance them between computation units. Scheduler can be used in conjunction with Driver concept to implement a specific parallel behavior, such as the ForkJoin concept to execute a collection of independant tasks or the Pipeline concepts to execute an ordered collection of tasks differing the management of their children tasks at the end of the collection process.

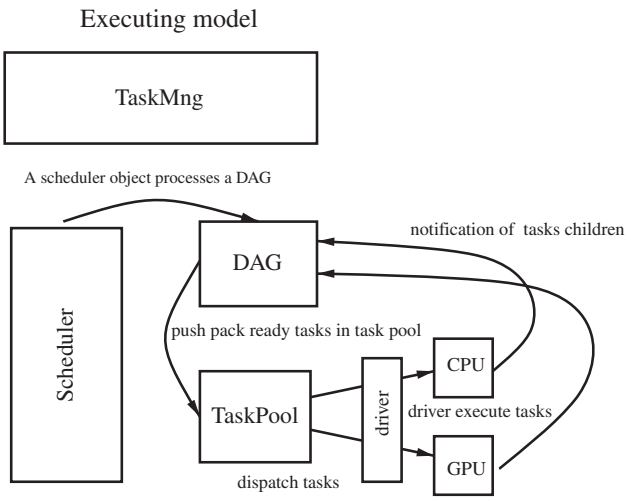


Figure 2
HARTS's executing model.

1.2.5 Executing Model

Now we have explained all the components and the abstraction provided by the runtime, we can present the executing model of HARTS, illustrated in Figure 2. First, a scheduler object processes a DAG of tasks belonging to the centralized task manager. Ready tasks are pushed back in a task pool and they are then dispatched on idle computational units following the scheduler behavior to enhance the global performances. Tasks are dispatched on hardware devices through a Driver object. Driver concept implement specific behavior such as ForkJoin or Pipeline behaviors. A DAG is completely processed once the task pool is empty.

2 ITERATIVE METHODS FOR LARGE SPARSE SYSTEMS

The efficiency of the linear solvers is a key point of the simulator's performance. Each iteration of these algorithms are based on a sequence of algebraic operations on matrices and vectors. Among these operations we focus on the most widely used, usually provided by libraries implementing the BLAS (Basic Linear Algebra Subprograms) level 1 and 2 [12].

We first discuss of the data structure used to represent large sparse and untructured Matrix and Vector. We present the way we manage multi-level parallelisms both at a coarse and a fine grain parallelism. As these techniques are well

suited to task centric programming model and data flow paradigm, we discuss on how we have implemented them with different runtime system tools. We focus in particular on the implementation of the Sparse Matrix Vector product (SpMV), of some preconditioners. Finally we discuss on specific issues due to the iterative characteristic of the algorithms.

2.1 Large Sparse Matrices, Data Structures

Our work is based on large unstructured matrices. Common data structures are inefficient to store them. As there is many zero entries which are not necessary to keep in memory, we need a specific structure for sparse matrices which store only non-zero values. We describe our matrices following the Compressed Sparse Row (CSR) format. This structure, well described in [1], reduces memory footprint to store elements of sparse matrices while storing only non-zero entries.

2.2 Multi-Level Parallelisation of Linear Algebra Operation

2.2.1 Linear System Dual Graph, Graph Partitions

Let consider a matrix A and its coefficients $(a_{i,j})$ with $0 \leq i < n$ and $0 \leq j < n$. Let $G_A = (V, S)$ be the dual graph of A where $V = (v_i)$ is the set of vertices representing the rows of A, and $S = (s_{i,j})$ the set of edges connecting vertices v_i and v_j such that $a_{i,j} \neq 0$.

We define a partition P of G_A in p sub-parts as the set $(V^k)_{0 \leq k < p}$ of subsets of V where $V^k \subset V$ and $V^{k_1} \cap V^{k_2} = \emptyset$ if $k_1 \neq k_2$ and $V = \cup (V^k)_{0 \leq k < p}$.

For each V^k, we define Vi^k the set of interior vertices $v_i \in V^k$ such as if $a_{i,j} \neq 0$ then $v_j \in Vi^k$ and Vb^k the set of boundary vertices $v_i \in V^k$ such as there is at least one $k_2 \neq k$ and one $v_j \in V^{k_2}$ such as $a_{i,j} \neq 0$. We have clearly $V^k = Vi^k \cup Vb^k$.

Let consider a vector x and its components (x_i) with $0 \leq i < n$. The vector components x_i are associated to the vertices v_i of G_A while the non zero entries $a_{i,j}$ of A are associated to the edges $s_{i,j}$ of G_A.

2.2.2 Data Distribution

In modern heterogeneous hardware architecture, we have often several nodes connected by an external network interconnection like for instance an Infiniband network. Each node can be composed of several sockets connected by a Quick Path Interconnect (QPI) link for example. On each socket we have a multi-core processor where each core has his own L1 cache memory – the closest cache memory level to the core – and share or not the over level of cache memory. We classically need to handle 3 levels of memory, a distributed memory at the cluster level, connected memories

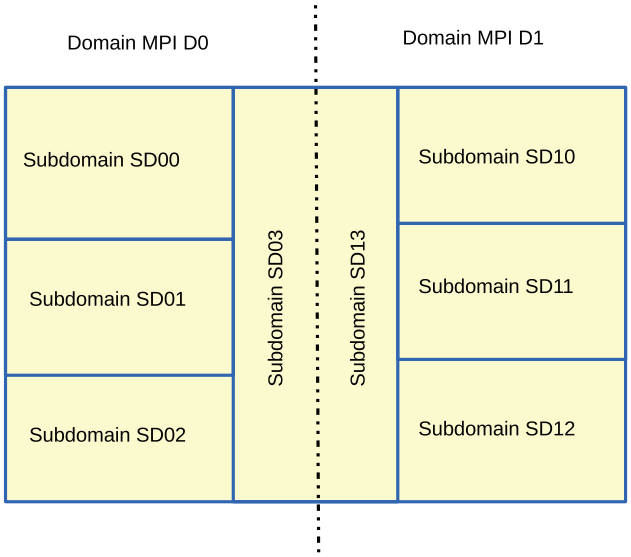


Figure 3

Hierarchical domain partition for multi-level parallelism.

at the multi-sockets node level, and finally a cache memory hierarchy at the multi-core processor level. To handle these 3 level memory hierarchy, we use multi-level domain decomposition techniques (Fig. 3) based on a hierarchical partition of the dual graph of our linear system. A first partition P_0 of G_A enable to distribute at a first level the data on the distributed memory of the cluster nodes. For each sub part V^k level 1, we can consider the sub graph Gi_A^k composed with the vertices $v_i \in Vi^k$. Gi^k can be partition in a number of sub-parts level 2 regarding the number of sockets of the node and then recursively each of them partitioned in sub-parts level 3 regarding the number of cores on each sockets. We use some 3 level hierarchical partitioner algorithms, which are aimed to reduce the amount of data transferred in the external network between interconnected nodes by minimizing the size of the sets Vb_{l1}^k, then with the second level, to reduce the amount of transferred data through the QPI link between the different sockets of each nodes minimizing the size of the sets of Vb_{l2}^k. Finally the last level partition ensure cache memory locality of the data processed by threads with their core affinity, avoiding concurrency of data belonging to the same cache lines.

At an algebraic level, linear systems with a matrix and vectors are associated to their dual graphs. The components of vectors represent the scalar data associated to the vertices of the graph while the non zero entries of the matrix are associated to the edges of the graphs. The partition of the dual graph enables to partition the vectors and the matrices data, and gather them contiguously in the memory.

2.2.3 Parallelism

The dual graph hierarchical partition enables to handle different levels of parallelism. We use an hybrid parallelism based on MPI (Message Passing Interface) parallelism at the coarse level to handle the distributed memory architecture and the data distribution at the node level, and thread parallelisms inside each nodes. This one is better appropriate to handle shared memory architecture and to reduce the memory footprint which is all the more important when the number of cores increases for a limited local memory size.

2.2.4 Data Flow and Task Centric Implementation

Considering the partition of the matrices and vectors data regarding the graph partition in Vi^k and Vb^k, we split any algebraic algorithms processing these data in tasks grouping elementary operations processing the piece of data associated to the set of vertices Vi^k and Vb^k. Internal tasks (Ti^{k_1}) processing data of V^{k_1} are clearly independent of the internal tasks (Ti^{k_2}) processing data of V_{k_2} with $k_1 \neq k_2$. They can only depend of other tasks (Ti^{k_1}) and (Tb^{k_1}) of the same domain. Boundary tasks (Tb^{k_1}) can depend of other boundary tasks (Tb^{k_2}) of other domains V_{k_2}. The data flow paradigm enables to manage the dependencies between tasks processing the data of the same sub sets of vertices V^k. The global algorithm is based on a DAG of these tasks Ti^k and Tb^k. The dependencies between data of different sub sets Vb^k are managed by boundary tasks Tb^k which precede them. As partitioners are aimed to minimize the size of sub sets Vb^k, we can fancy that the DAG of algorithms is composed of different independent tasks, linked by boundary tasks Tb^k with reduced cost.

3 ABSTRACT LINEAR ALGEBRA API

We start here from a mathematical view of linear algebra algorithms. We describe such algorithms to take advantage of multi-level parallelism in a transparent way for users. In this section, we present an abstract linear algebra API aimed to implement common linear algebra algorithms with a high level formalism. We illustrate how we handle parallelism and the data distribution early described with the example of the multiplication between a sparse matrix and a vector.

3.1 Linear Algebra Algorithms and API's Description

Linear algebra algorithms can often been described as sequences of matrices and vectors algebraic operations. These operations are mostly BLAS [12] level 1 and 2 vector operations, some sparse matrix-vector products or specific preconditioning operations. For example, we can consider in Algorithm 1, one step of the BiConjugate Gradient Stabilized (BiCGStab [1]), the Krylov methods for solving large sparse linear systems.

Algorithm 1: BiCGStab Algorithm

```
Matrix A;
Vector b, p, pp, r, v;
Scalar a;
do
  pp = inv(precond).p;
  v = A.p;
  r + = v;
  a = dot(p,r);
  if (a == 0) break;
  ...;
while(|r|<tol*|b|);
```

To implement a parallel version of such algorithms in a transparent way, we have defined an abstract algebraic API detailled in Listing 1 aimed: (i) to hide hardware specificities; (ii) to manage memory allocation and locality; (iii) to manage parallel loops; (iv) to provide most BLAS 1 and 2 functionalities; (v) to provide tools to split vectors and manage vector views and range iterators, to manage transparently data according to their distribution.

Listing 1: Linear Algebra API.

```
class AlgebraKernel
{
  void allocate (Vector & v,
                 std : : size_talloc_size);
  void assign (Vector & v, LambdaT op);
  void axpy (ValueT const &a,
             Vector const& x,
             Vector const& y);
  doubledot (Vector const& x,
             Vector const& y);
  void mult (Matrix const& A,
             Vector const& x,
             Vector const& y);
  void exec (Precond const & P,
             Vector const& x,
             Vector const& y);
  void assertNull (double const& value);
};
```

Algorithm's step are then implemented as a DAG of operations, associated to a Sequence objects. These objects refer to a sequence of algebraic operation organized in a persistent data structure. In this way, they can be replayed several time which is in particular interesting for iterative

methods implementations. Listing 2 illustrates the implementation of the Algorithm 1 with our API's formalism.

Listing 2: BiCGStab sequence.

```
Algebr aKernelType alg;
Matrix A; Vector p, pp, r, v;
double alpha;
SequenceType seq = alg.newSequence ();
alg.exec (precond, p, pp, seq);
alg.mult (A, pp, v, seq);
alg.axpy (1., r, v, seq);
alg.dot (p, r, alpha, seq);
alg.assertNull (alpha, seq);
while (!iter. stop ())
{
   alg. process (seq);
}
```

We have implemented this API with HARTS and with various other runtime system layers like XKaapi, OmpSS and OpenMP runtime systems.

4 PARALLEL PRECONDITIONERS

As the cost of these iterative algorithms depends directly on the number of iterations required for convergence, the choice of a robust parallel preconditioner and appropriate solver options is important as they have a strong impact on the cumulative number of iterations for the whole simulation. It is important to parallelize preconditioner to take advantage of new recent architectures. We now review some parallel preconditioners developed in the MCGSolver library, and explain how they have been parallelized. We first introduce ILU(0) then present a multi-level domain decomposition methods.

4.1 Incomplete LU Factorization (ILU)

Given a factorization of a large sparse matrix A, in such a way that

$$A = LU \tag{1}$$

where L is a lower triangular matrix, and U an upper triangular matrix. It is well known that usually in the factorization procedure, the matrices L and U have more non zero entries than A. These extra entries are called fill-in elements. The incomplete factorization consists in dropping some of these elements. An incomplete factorization preconditioner consists in taking $M = \bar{L}\bar{U} \approx A$, where $\bar{L}$ and $\bar{U}$ stand for the incomplete LU factors of A. For an incomplete factorization with no fill-in, so-called ILU(0), we take the zero pattern of A.

Such an algorithm is not natively parallel, and a distributed graph decomposition is needed to perform it to extract a limited amount of parallelism. This defines interior nodes in a subgraph that do not depend on variables from other node. These nodes can be independently eliminated and the associated operations can be done in parallel. But interface nodes are defined such as they are coupled with rows which are located on another processor and need a particular attention. The parallel algorithm is given by [1]. At each solver iterate, a backward and a forward steps are requiring to perform ILU(0) preconditioning on which we have to separate operations coming from interior and interface nodes. In interface computation, some communications are performed to receive updated values from neighbor subdomains as illustrated in Figure 4 representing the distributed DAG of the BiCGStab algorithm preconditioned with a parallel ILU(0) preconditioner. Dependencies between tasks involve synchronizations between processors reducing the scalability of this preconditioner up to very large number of cores.

As it is illustrated in this figure, the task programming approach enables to overlap transparently communication between processes minimizing in that way the overhead of communication.

4.2 Multi-Level Domain Decomposition Preconditioner

Due to their parallel nature, domain decomposition methods present interesting algorithmic features for parallel architectures. Domain decomposition methods consist in solving the problem in

$$\Omega = \bigcup_{i=1}^{p} \Omega_i \tag{2}$$

by solving the sub-problems on the p sub-domains Ω_i. These kind of methods are interesting for parallel computer architectures. The Additive Schwarz Method (ASM), described in [1], consists in:

$$u^{m+1} = u^m + J^{-1}(b - Au^m) \tag{3}$$

where J stands for the Jacobian matrix, which is the block diagonal part of the matrix A. The Multiplicative Schwarz Method (MSM) take for M the Gauss-Seidel matrix, M_{GS}, built from the block lower triangular part of A. Contrary to this method, ASM is naturally parallel as it provides independent tasks. At each step, a linear system for each subdomain has to be solved. Due to the shape of the matrix M

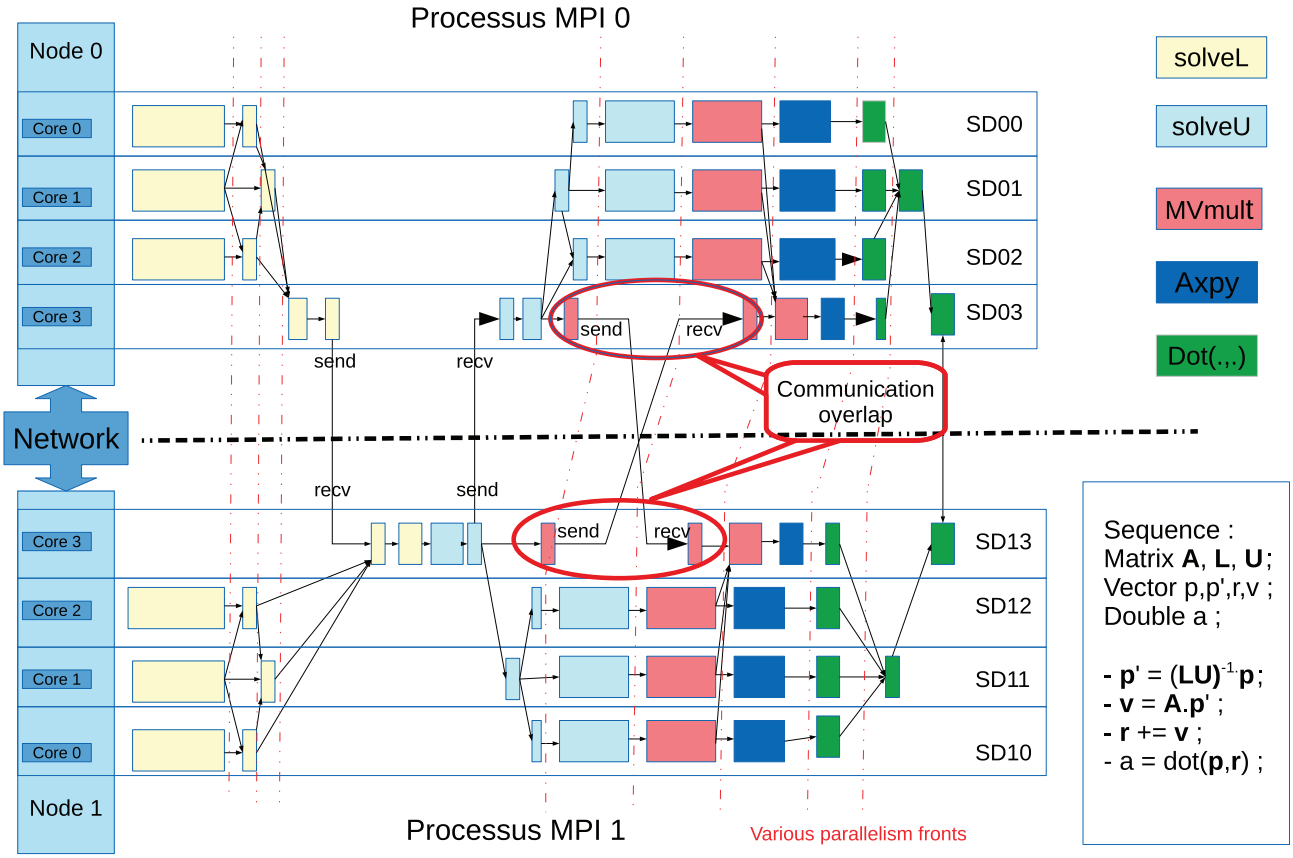


Figure 4

DAG of ILU(0) preconditioner.

and the domain partitioning, this operation can be done in parallel with a direct method (because of the reduced size of the problem). But ASM suffers from a lack of extensibility. In fact each sub-domains only communicate with their neighbors at each step, which explains why the convergence rate slow down when the number of sub-domains increases. This problem is solved by the 2-level Additive Schwarz Method (2-level ASM) reviewed by [13], described as follows:

$$u^{m+1} = u^m + P^{-1}(b - Au^m)$$

$$P^{-1} = M_j^{-1}\left(I_n + AZE^{-1}Z^T\right) + ZE^{-1}Z^T \qquad (4)$$

where $E = Z^T AZ$ is the coarse operator computed from Z, the deflation matrix. It imposes a global communication by solving a coarse problem. Coarse operators may differ from their property to converge more or less faster depending on the structure of the matrix A. Figure 5 illustrates the DAG of macro operations on which the synchronization is well represented. However, the cost of the coarse resolution is related to the size of coarse operators. This size has to be adapted to ensure the extensibility while preserving the parallel performance of the method. The rest of the work flow is totally parallel which grants good performance on parallel machines as tasks can be processed independently.

For our work, we selected two coarse operators. The first one, introduced in [14], is based on the indicator functions of each sub-domain. We denote it by the Nicolaides coarse operator. For homogeneous problems, this easy-to-compute coarse operator offers good convergences rates while for heterogeneous problems, these rates decreased. Inversely, the second implemented coarse operator described in [15] is proven to be numerically stable and robust. We denote it by the GenEO coarse operator. It consists in solving an eigenvalue problem in the overlaps and is built with the lowest eigenvalues found.

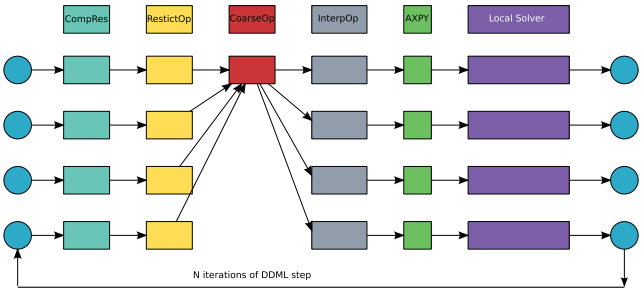


Figure 5

Multi-level domain decomposition method DAG.

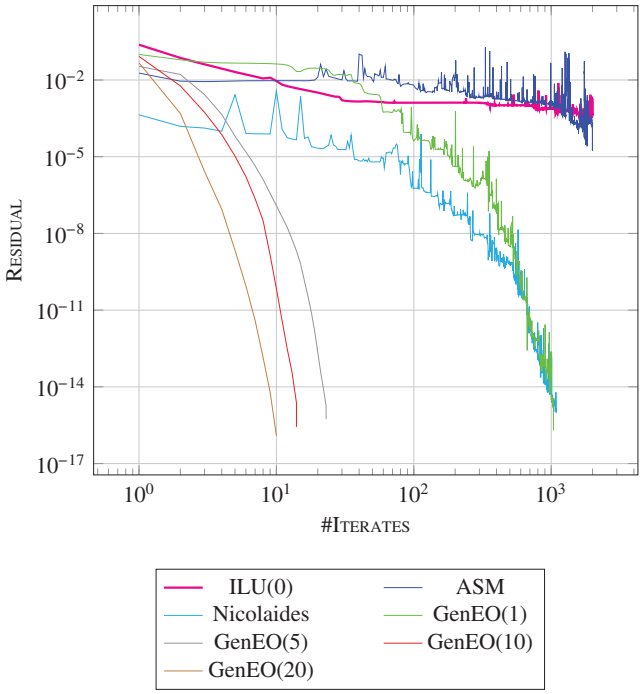


Figure 6

Convergence status of preconditioned BiCGStab on a system coming from laplacian 2D discretization.

4.3 Convergence Issues

Preconditioner aims to improve convergence rates of a linear solver. In some cases, it is not sufficient and we have to develop new algorithms to reach convergence criteria. To be close to the physical phenomena involved in petroleum reservoir simulator, we applied a permeability field on the system obtained from a 2D discretization of a Laplace problem on a unit cube mesh of size $nx \times ny$. The following permeability field was applied, where *floor* $(x) = [x]$:

$$perm\,(i,j) = \begin{cases} 10^3 \times [10 \times j/ny] + 1 \\ \text{if } [10 \times i/nx] \equiv 0\,[2] \text{ and } [10 \times j/ny] \equiv 0\,[2] \\ 1 \text{ else} \end{cases} \quad (5)$$

We then study the speed of convergence of a BiCGStab algorithm preconditioned with each preconditioning method previously studied in Section 4. The evolution over iterations of the relative error is illustrated in Figure 6, where the application was performed with 40 sub-domains. In this figure, ILU(0) stands for the Incomplete LU factorization with no fill, while the "ASM" denomination stands for 1-level Additive Schwarz Method. We also compare several coarse operators used with 2-level ASM preconditioners mentioned in Section 5.3. We denote the Nicolaides coarse operator by "Nicolaides", and the GenEO coarse operator by "GenEO(p)", where p stands for the number of computed eigenvalues for the eigen problem. For Domain Decomposition (1 and 2 level) preconditioner, we incorporate 10 steps of the method as preconditioner.

The BiCGStab method failed to converge with ILU(0) and ASM preconditioners. The same method preconditioned with GenEO(1) or Nicolaides methods converge over more than 10^3 iterates. On the other hand, GenEO(p) with $p > 1$ requires a small number of iterates to converge.

5 EXPERIMENTS

In this section, we benchmark the different implementations of our linear solver described in the previous sections with various linear systems on different hardware configurations.

The first benchmark consists in performing several SpMV operations comparing the HARTS based implementation to a hand written implementation with POSIX threads (*i.e.* pthread library) in order to evaluate the overhead of the runtime system layer.

We then present experimentation with the full BiCGStab solver preconditioned with the DDML preconditioner. We compare the different coarse operators described in Section 5.3.

We finally expose results obtained on distributed memory architecture with the full BiCGStab solver preconditioned with ILU(0) preconditioner.

For all these experiments, we used the METIS [16] graph partitioner. Tests are run on a Linux cluster system with shared-memory dual-socket nodes, each socket equipped with a octo-core Sandy Bridge processor.

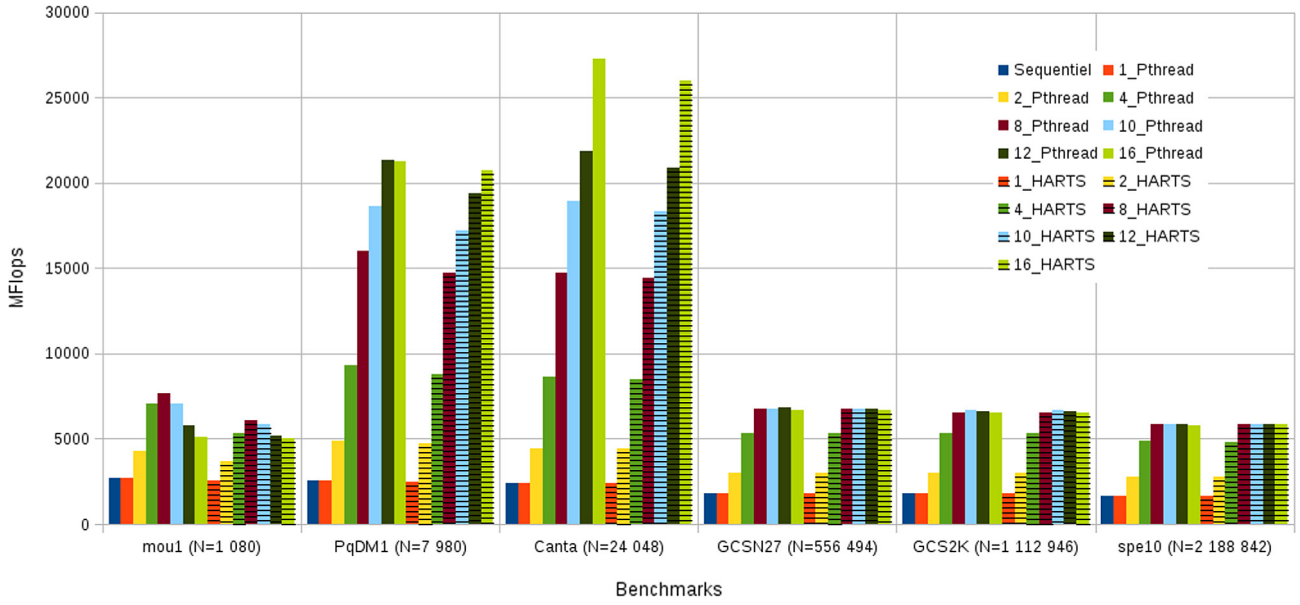


Figure 7

SpMV Performances - HARTS - Round Robin a nity.

5.1 Sparse Matrix Vector Product

This benchmark consists in performing 10^3 SpMV products with various matrices of distinct and representative sizes. As the SpMV is one of the most important operation performed in iterative linear solvers, the performance results obtained in this benchmark are representative of that can be expected for the whole solver. The METIS graph partitioner has been used to partition data and parallelize algorithms with tasks. We have used a round-robin affinity strategy to bind threads to core. The performance results for each matrix are gathered in Figure 7. We have tested a family of six linear systems extracted from a reservoir simulator. We give for each systems its size (the number of rows and unknowns). For each linear system, the benchmark has been executed with 1, 2, 4, 8, 10, 12, 14 and 16 threads. We compare the performance of the HARTS implementation to a hand written one with the pthread library.

For small benchmarks, even if threads can benefit from cache effect, the performances are affected by the cost of small data communication between threads. Increasing the size of processed data highlights the performance of the runtime system comparing to pthread implementation. Once data are stored in cache memory, communication does not impact any more the performance. Larger benchmarks accentuate the fact that data locality is primordial for parallelism. The peak of performance is reached for more than eight threads as data is well distributed among sockets and as the runtime system's cost is negligible in comparison of data computation. All the obtained results underline the cost of the runtime systems, which overhead seems not important even negligible for large linear systems.

5.2 Distributed ILU(0) Preconditioner

As early mentioned, ILU(0) preconditioner is not naturally parallel. The algorithm is decomposed in several sequences of operations with underlying dependencies. We present here the distributed algorithm used with a BiCGStab solver, taking advantage of different levels of parallelism, as described in Section 2.2.2. The purpose of this test is to evaluate the hybrid parallelization MPI + thread of this preconditioner on a collection of linear systems coming from a Finite Volume discretization of a 2D Laplace problem on a unit cube with various sizes of mesh. In Figure 8, we compare the performance of the solver in a full MPI mode to the one in a full thread mode with HARTS. We can notice that up to 10 cores, the behaviour of the two modes is quite similar. For a number of cores larger than 10, we observe a lack of scalability for the two modes, mostly for the thread mode.

In Figure 9 we analyze the performance of the solver preconditioned with the ILU(0) method with a multi-level parallelization. For this test we use a mesh of size 2000×2000. For this benchmark, tests are run on up to four nodes. We first experiment the configuration of one MPI process per node, with 16 threads distributed among the two sockets of the node. We then use two MPI processes

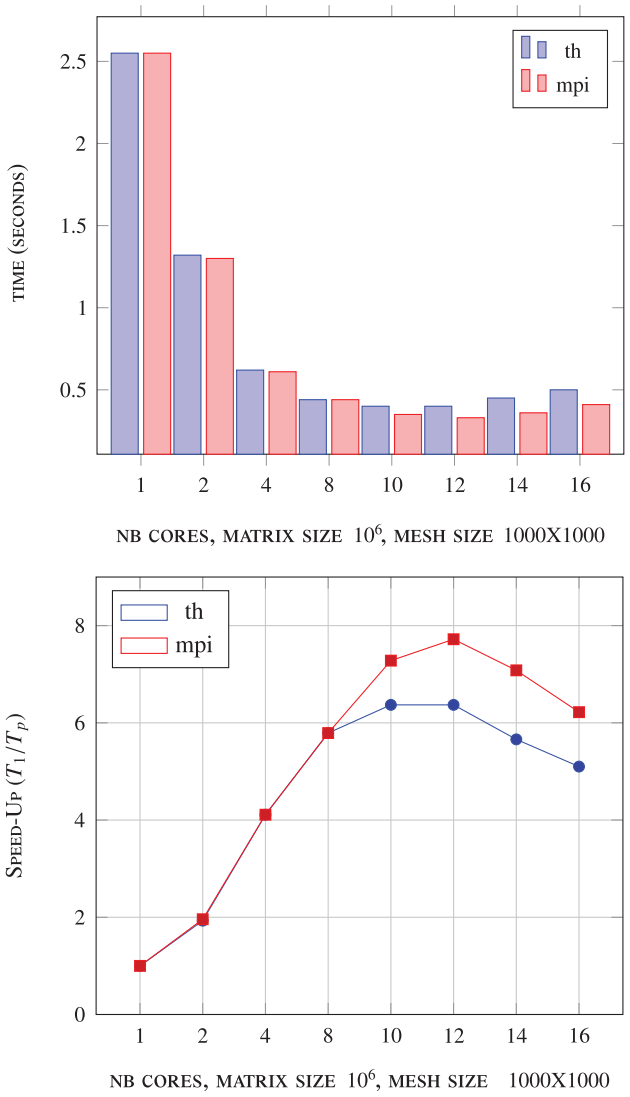


Figure 8

ILU(0) preconditioner within one node.

per node, each process is attached to a disjoint socket with eight threads per socket. We finally experiment a full MPI implementation, on which we associate 16 MPI processes per node with one thread. For all of these configurations, we use a coarse-grained size parallelism for each MPI processes, and then a finer grain size parallelism with HARTS threads at the socket level. As we can see on results, the assignment of one partition per node (*i.e.* one MPI process) presents a small overhead compared to other implementations. By adding more MPI processes, we can see that communication between nodes does not have a significant impact on results because of the small pieces of data to send and receive. The scalability in a node is still not optimal, as previously shown.

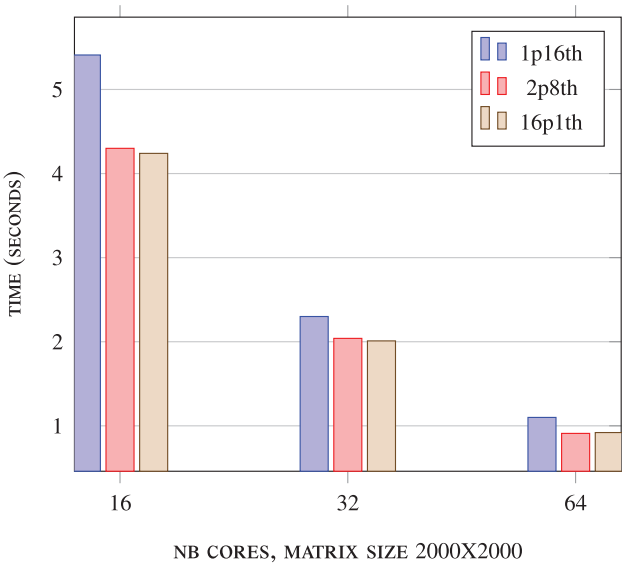


Figure 9

ILU(0) preconditioner on multi nodes architectures.

5.3 Shared-Memory Multi-Level Domain Decomposition Methods

As early mentioned, domain decomposition methods naturally fit on parallel architectures. This experiment aims to reveal the advantages of this category of preconditioning techniques on shared-memory systems. For this benchmark, the data locality has a strong impact in the sense that communication are only inside the node with shared memory optimization. As these methods can be used both as a solver or as a preconditioner, we evaluate first the parallelization of the solver, then as a preconditioner for the BiCGStab solver with 10 steps of DDML solver. For our experiments, we use a fixed number of subdomains equal to 40. For the 2-level ASM, we experiment the Nicolaides and GenEO coarse operators described in this section. For the GenEO coarse operator, we use different number of computed eigenvalues.

Performance results are gathered in Figure 10. We can notice that, contrary to ILU(0) in Figure 8, the speed up keeps on growing up to 16 cores. The scalability declines for more than 10 cores for both benchmarks with the GenEO(10) coarse operator. This degradation is due to the fact that the coarse system's size grows with the increasing number of computed eigenvalues while the resolution of this system remains sequential as illustrated in Figure 5. In spite of this sequential part, the parallel implementation of the 2-level ASM remains quite as efficient as the 1-level ASM with only independent tasks.

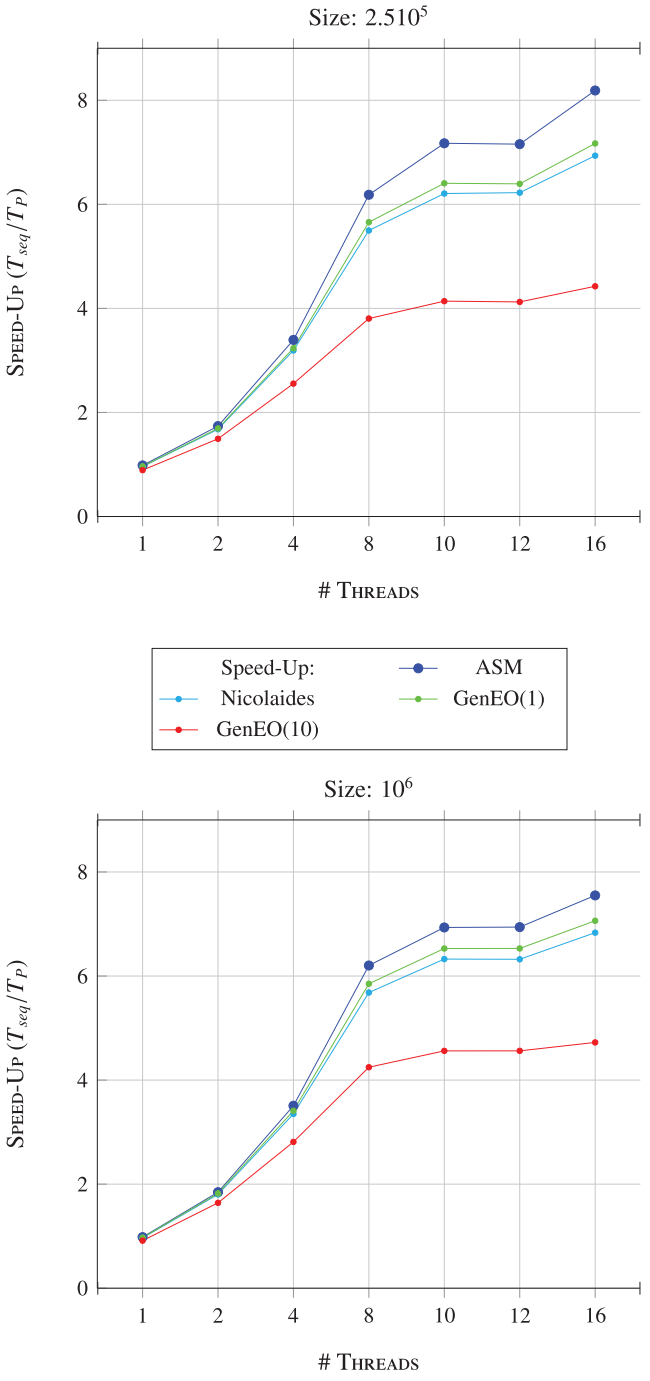


Figure 10

Speed-Up – laplacian 2D sparse system – DDML solver using a 1-level ASM and 2-level ASM with GenEO and Nicolaides coarse operators. Test performed with 40 sub-domains.

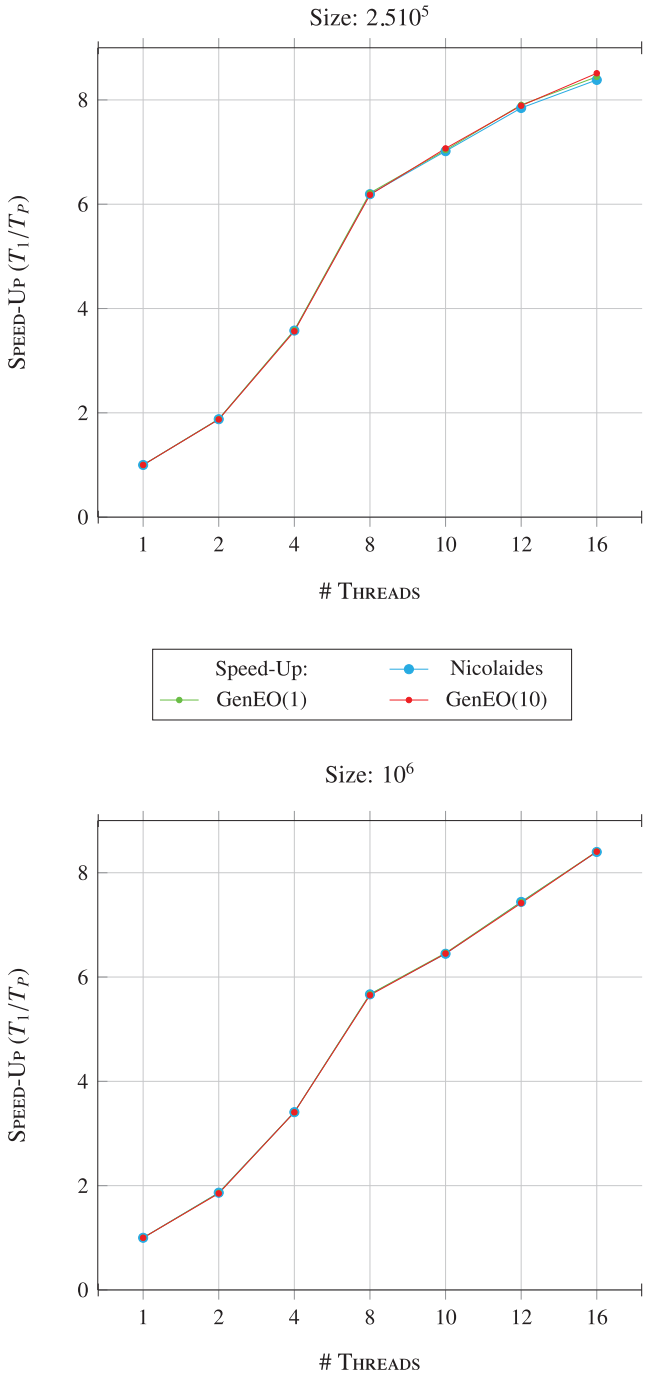


Figure 11

Speed-Up – laplacian 2D sparse system – BiCGStab solver with DDML preconditioner. Test performed with 40 sub-domains.

Figure 11 refers to the implementation of DDML algorithms used as a preconditioner of the BiCGStab solver. We can notice that the speed-up keeps on growing up to 16 cores whatever is the size of the linear system. As a preconditioner the DDML method seems more scalable than the ILU(0) preconditioner.

CONCLUSION AND PERSPECTIVES

In this paper we have presented the implementation of iterative linear solvers for large and sparse linear systems with runtime system tools to handle efficiently and seamlessly the complexity of new heterogenous architectures. We have introduced an abstract linear algebra API that enables to write at a high level algebraic algorithms hiding the use at a low level of the task programming paradigm leading to Direct Acyclic Graph processed by schedulers that distribute tasks among available resources. We have detailed the implementation of the sparse matrix-vector product, the ILU(0) preconditioner and the Multi-Level Domain Decomposition preconditioner. We have focused on multi-level parallelization strategies using threads at a fine level and MPI at a coarse level. We have validated our approach with some experiments with various linear systems comparing different implementations on different hardware configurations, evaluating the runtime system overhead and the efficiency of the implemented algorithms. According to the performance results, our implementation seems efficient even for not naturally parallel algorithms like the ILU(0) factorization. We have obtained interesting scalability for the DDML preconditioner. The whole preconditioned BiCGStab algorithm turns to be scale not only within one node using all the cores, but also on several nodes with the muti-level hybrid parallelism using MPI at the coarse level and threads with HARTS at a finer level.

In future work, thanks to our abstract linear algebra API, we plan to benchmark other runtime systems like XKaapi, OmpSS or OpenMP runtime, and other hardware configurations with many integrated core processors like the Intel Xeon-Phi processor.

REFERENCES

1 Saad Y. (2003) *Iterative methods for sparse linear systems*, 2nd edn., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0898715342.

2 Gratien J.-M. (2013) An abstract object oriented runtime system for heterogeneous parallel architecture, in: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW'13*, IEEE Computer Society, Washington, DC, USA, pp. 1203-1212. ISBN 978-0-7695-4979-8. DOI: 10.1109/IPDPSW.2013.144.

3 Anciaux-Sedrakian A., Gottschling P., Gratien J.-M., Guignon T. (2014) Survey on Effficient Linear Solvers for Porous Media Flow Models on Recent Hardware Architectures, *Oil Gas Sci. Technol. - Rev. IFP* **69**, 4, 753-766.

4 Goglin B. (2014) Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc), in: *International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014. IEEE. URL https://hal.inria.fr/hal-00985096.

5 NVIDIA Corporation. *NVIDIA CUDA compute unified device architecture programming guide*. NVIDIA Corporation, 2007.

6 Stone J.E., Gohara D., Shi G. (2010) OpenCL: a parallel programming standard for heterogeneous computing systems, *IEEE Des. Test* **12**, 3, 66-73.

7 OpenMP Architecture Review Board (2013) *OpenMP application program interface version 4.0*. URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

8 Gautier T., Lima J.V.F, Maillard N., Raffin B. (2013) Xkaapi: a runtime system for data-flow task programming on heterogeneous architectures, in: *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, USA, May 2013. URL https://hal.inria.fr/hal-00799904.

9 Augonnet C., Thibault S., Namyst R., Wacrenier P.-A. (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, *Concurr. Comput.* **23**, 2, 187-198.

10 Bueno-Hedo J., Planas J., Duran A., Badia R.M., Martorell X., Ayguadé E., Labarta J. (2012) *Productive programming of GPU clusters with OmpSs*, 21 May. URL http://www.computer.org/csdl/proceedings/ipdps/2012/4675/00/4675a557-abs.html.

11 Broquedis F., Clet-Ortega J., Moreaud S., Furmento N., Goglin B., Mercier G., Thibault S., Namyst R. (2010) Namyst. hwloc: a generic framework for managing hardware affinities in HPC applications, in: IEEE (ed.), *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010. DOI: 10.1109/PDP.2010.67. URL https://hal.inria.fr/inria-00429889.

12 Blas (basic linear algebra subprograms. URL http://www.netlib.org/blas/.

13 Dolean V., Jolivet P., Nataf F. (2015) *An introduction to domain decomposition methods: algorithms, theory, and parallel implementation*. URL http://bookstore.siam.org/ot144/.

14 Nicolaides R.A. (1987) Deflation of conjugate gradients with applications to boundary value problems, *SIAM J. Numer. Analysis* **24**, 2, 355-365.

15 Spillane N., Dolean V., Hauret P., Nataf F., Pechstein C., Scheichl R. (2014) Abstract robust coarse spaces for systems of PDEs via generalized eigenproblems in the overlaps, *Numerische Math.* **126**, 4, 741-770.

16 Karypis G., Kumar V. (1998) A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Comput.* **20**, 1, 359-392.

Manuscript submitted in December 2015
Manuscript accepted in September 2016
Published online in November 2016

Cite this article as: A. Roussel, J.-M. Gratien and T. Gautier (2016). Using Runtime Systems Tools to Implement Efficient Preconditioners for Heterogeneous Architectures, *Oil Gas Sci. Technol* **71**, 65.