This paper is a part of the hereunder thematic dossier published in OGST Journal, Vol. 69, No. 4, pp. 507-766 and available online here.

Cet article fait partie du dossier thématique ci-dessous publié dans la revue OGST, Vol. 69, n°4 pp. 507-766 et téléchargeable ici.

DOSSIER Edited by/Sous la direction de : Z. Benjelloun-Touimi

Geosciences Numerical Methods

Modélisation numérique en géosciences

Oil & Gas Science and Technology – Rev. IFP Energies nouvelles, Vol. 69 (2014), No. 4, pp. 507-766

Copyright © 2014, IFP Energies nouvelles

507 > Editorial
J. E. Roberts

515 > Modeling Fractures in a Poro-Elastic Medium
Un modèle de fracture dans un milieu poro-élastique
B. Ganis, Y. Girault, M. Mez, G. Singh and M. Wheeler

529 > Modeling Fluid Flow in Faulted Basins
Modélisation des transferts fluides dans les bassins failles
I. Faille, M. Thibaut, M.-C. Cacas, P. Havé, F. Willien, S. Wolf, L. Agelas and S. Pegaz-Fiornet

555 > An Efficient XFEM Approximation of Darcy Flows in Arbitrarily Fractured Porous Media
Une approximation efficace par XFEM pour écoulements de Darcy dans les milieux poreux arbitrairement fracturés
A. Fumagalli and A. Scotti

565 > Hex-Dominant Mesh Improving Quality to Tracking Hydrocarbons in Dynamic Basins
Amélioration de la qualité d’un maillage hexa-dominant pour la simulation de l’écoulement des hydrocarbures
B. Yahiaoui, H. Borouchaki and A. Benali

573 > Advanced Workflows for Fluid Transfer in Faulted Basins
Méthodologie appliquée aux circulations des fluides dans les bassins failles
M. Thibaut, A. Jardin, I. Faille, F. Willien and X. Guichet

585 > Efficient Scheme for Chemical Flooding Simulation
Un schéma numérique performant pour la simulation des écoulements d‘agents chimiques dans les réservoirs pétroliers
B. Bracconier, E. Flauraud and Q. L. Nguyen

603 > Sensitivity Analysis and Optimization of Surfactant-Polymer Flooding under Uncertainties
Analyse de sensibilité et optimisation sous incertitudes de procédés EOR de type surfactant-polymère
F. Douarche, S. Da Veiga, M. Feraille, G. Enchéry, S. Touzani and R. Barsalou

619 > Screening Method Using the Derivative-based Global Sensitivity Indices with Application to Reservoir Simulator
Méthode de criblage basée sur les indices de sensibilité DGSM : application au simulateur de réservoir
S. Touzani and D. Busby

627 > An Effective Criterion to Prevent Injection Test Numerical Simulation from Spurious Oscillations
Un critère efficace pour prévenir les oscillations parasites dans la simulation numérique du test d’injection
F. Verge, D. Yiberti, E. Salina Borello and C. Serazio

653 > Well Test Analysis of Naturally Fractured Vuggy Reservoirs with an Analytical Triple Porosity – Double Permeability Model and a Global Optimization Method
Analyse des puits d’essai de réservoirs vakuolaires naturellement fracturés avec un modèle de triple porosité – double perméabilité et une méthode d’optimisation globale
S. Gómez, G. Ramos, A. Mesejo, R. Camacho, M. Vásquez and N. del Castillo

661 > Comparison of DDFV and DG Methods for Flow in Anisotropic Heterogeneous Porous Media
Comparaison des méthodes DDFV et DG pour des écoulements en milieu poreux hétérogène anisotrope
V. Baron, Y. Couëdière and P. Sochala

671 > Adaptive Mesh Refinement for a Finite Volume Method for Flow and Transport of Radionuclides in Heterogeneous Porous Media
Adaptation de maillage pour un schéma volumes finis pour la simulation d‘écoulement et de transport de radionucléides en milieux poreux hétérogènes
B. Amaziane, M. Bourgeois and M. El Fatini

701 > A Review of Recent Advances in Discretization Methods, a Posteriori Error Analysis, and Adaptive Algorithms for Numerical Modeling in Geosciences
Une revue des avancées récentes autour des méthodes de discrétisation, de l’analyse a posteriori, et des algorithmes adaptatifs pour la modélisation numérique en géosciences
D. A. Di Pietro and M. Vohralík

731 > Two-Level Domain Decomposition Methods for Highly Heterogeneous Darcy Equations. Connections with Multiscale Methods
Méthodes de décomposition de domaine à deux niveaux pour les équations de Darcy à coefficients très hétérogènes. Liens avec les méthodes multi-échelles
V. Dolean, P. Jolivet, F. Nataf, N. Spillane and H. Xiang

751 > Survey on Efficient Linear Solvers for Porous Media Flow Models on Recent Hardware Architectures
Revue des algorithmes de solveurs linéaires utilisés en simulation de réservoir, efficaces sur les architectures matérielles modernes
A. Anciara-Sedrakian, P. Gottschling, J.-M. Gratien and T. Guignon
Survey on Efficient Linear Solvers for Porous Media Flow Models on Recent Hardware Architectures

Ani Anciaux-Sedrakian1*, Peter Gottschling2,3, Jean-Marc Gratien1 and Thomas Guignon1

1 IFP Energies nouvelles, 1-4 avenue de Bois-Préau, 92852 Rueil-Malmaison Cedex - France
2 SimuNova, Helmholtzstr. 10, 01069 Dresden - Germany
3 Institute Scientific Computing, TU Dresden, 01062 Dresden - Germany

e-mail: ani.anciaux-sedrakian@ifpen.fr - peter.gottschling@simunova.com - jean-marc.gratien@ifpen.fr - thomas.guignon@ifpen.fr

* Corresponding author

Abstract — Survey on Efficient Linear Solvers for Porous Media Flow Models on Recent Hardware Architectures — In the past few years, High Performance Computing (HPC) technologies led to General Purpose Processing on Graphics Processing Units (GPGPU) and many-core architectures. These emerging technologies offer massive processing units and are interesting for porous media flow simulators used for CO2 geological sequestration or Enhanced Oil Recovery (EOR) simulation. However the crucial point is “are current algorithms and software able to use these new technologies efficiently?”

The resolution of large sparse linear systems, almost ill-conditioned, constitutes the most CPU-consuming part of such simulators. This paper proposes a survey on various solver and preconditioner
algorithms, analyzes their efficiency and performance regarding these distinct architectures. Furthermore it proposes a novel approach based on a hybrid programming model for both GPU and many-core clusters. The proposed optimization techniques are validated through a Krylov subspace solver; BiCGStab and some preconditioners like ILU0 on GPU, multi-core and many-core architectures, on various large real study cases in EOR simulation.

INTRODUCTION

In basin modeling or reservoir simulations, multiphase porous media flow models with highly heterogeneous data and complex geometries, lead to solve complex non-linear Partial Differential Equations (PDE) systems. These PDE are discretized with a cell-centered finite volume scheme in space and a fully implicit scheme in time, leading to a non-linear system which is solved with an iterative Newton solver. At each Newton step, the system is linearized, and the generated linear system is solved with a Bi-Conjugated Gradient Stabilized (BiCGStab) [1] or Generalized Minimal RESidual (GMRES) [1] algorithm, well suited for large, sparse and unstructured systems. This resolution phase constitutes the most expensive part of the simulation, representing nearly 60% to 80% of the total simulation time. The efficiency of the linear solvers is therefore a key point of the simulator’s performance. 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. Indeed these appropriate options have a strong impact on the cumulative number of iterations for the whole simulation.

Furthermore, to continue increasing the performance of architecture following the Moore’s law; reducing the energy consumption, hardware developers have to design heterogeneous architectures based on multi-core CPU enhanced with GPU accelerators. Indeed, the multi-GPU architectures are economically more interesting than the multi-core ones.

However, these architectures introduce new challenges in terms of algorithms and system software design; algorithms need to be adapted or sometimes completely rewritten according to the targeted architecture. Moreover, the bottleneck of each algorithm can vary for different hardware configurations. For instance, with GPU accelerator (heterogeneous multi-GPU) the main issue is to optimize the data transfer between the host memory and the remote local memory of the GPU while in the case of multi-core architecture, the major problem is to manage the concurrent access of the cores to the memory under the constraint of limited memory bandwidth. The heterogeneous multi-GPU architecture combines both constraints; low latency for data transfer between the CPU and the accelerators (GPU or many-core cards) and concurrent access of the cores to the memory.

In this paper, we focus on different types of preconditioners, on their implementation with different libraries (Hypre, Petsc, PML4, IFPSolver, MGCSolver) and we study their performance regarding different hardware architectures. We show how the choice of appropriate solver options and parallel preconditioners is tightly coupled to the hardware configuration. In the first section, we present some hardware and software considerations in the context of GPUs. The second section is a review of the main preconditioner algorithms commonly used with Krylov solvers. The third section describes the porting of preconditioned BiCGStab to GPU. Results are gathered and analyzed in the forth section, while section five concludes this work.

1 HARDWARE CONTEXT

In the exascale computing roadmap, using accelerators (GPUs and many-core) is a key point. These kinds of architectures offer a very high peak of floating-point operations per second, a very high peak memory bandwidth, for very interesting ratios of Flop per watt and Flop per euro.

In the past twenty years, GPU have evolved from fixed-function processors to massively parallel floating point engines. Hence, the idea of General-Purpose Computation on Graphic Processing Units (GPGPU) emerged to take advantage of the processing power of GPU for non-graphical tasks. General purpose programming tools like BrookGPU [2], CUDA [3], or OpenCL [4], ease GPU use for all programmers. However to fully benefit from the GPU, computational power still requires that algorithms exhibit a high degree of parallelism and regular data structures. This stems from the specific GPU hardware architecture: hundred basic computation cores (also called Stream Processor; SP) handle arithmetic and load/store operations without any flow control. The latter is handled by an additional unit called sequencer. This leads to a programming model called Data Parallel [5, 6]. Such programming and execution model is similar to those used in the 80’s and 90’s on massively parallel supercomputers [5, 6]. Nvidia GPU
provide a massively multithreaded execution model where threads are identical instruction flows working with different data.

Here, we focus on the two most widely available and used Nvidia GPU architectures at the time of writing this manuscript, namely: FERMI and Kepler. The FERMI architecture provides 448 SP for C/M2070 and up to 512 SP for M2090 models, grouped into 14 SM (16 for M2090) each containing 32 SP. With the Kepler architecture, NVIDIA provides the SMX (Streaming Multiprocessor eXtreme) architecture. Each SMX contains 192 cores, each capable of completing a single-precision floating-point multiple and adds (fused, to avoid truncating the intermediate result) in every clock cycle. In the K20 product, 13 of these SMX units are combined to give a total of 2,496 cores, giving a peak performance of 3.52 TFlops in single or 1.17 TFlops in double precision.

On the software side, CUDA and OpenCL have emerged as the leaders of general purposes software tools. While CUDA only provides support for Nvidia GPU, OpenCL is more generic and support Nvidia, AMD GPU and also multi-core CPU. In this work, we use the CUDA programming environment. Moreover, we consider that the GPU and the CPU process disjoint parts of the code. In this context, the GPU role is to decrease the computation time by assisting the CPU for the dedicated parts, even when taking into account the data transfer penalty.

2 PRECONDITIONERS OVERVIEW

In multiphase porous media flow models, the discretization and the linearization of the PDE lead to large linear systems to compute the pressure or/and the temperature which are elliptic/parabolic unknowns and the saturations or/and the compositions which are hyperbolic unknowns. These linear systems, often ill-conditioned due to the heterogeneous and anisotropic geological data, are classically solved with iterative Krylov subspaces methods such as GMRES, CGS, BiCGstab.

Efficient preconditioners are necessary to improve the linear systems’ condition numbers so that the used iterative solvers can converge with a reasonable number of iterations.

For a given sparse linear system, there is often no single preconditioner which yields the best performance on all computer architectures. The efficiency of preconditioners depends numerically on the problem size, on the heterogeneity, the discontinuities and the anisotropies of the problem. But the efficiency of their implementation depends on the target hardware architecture. Some algorithms may be numerically efficient and help to reduce the number of iterations considerably, but they may be difficult to parallelize and not efficient in terms of flops on certain hardware configurations. Even more, to have performance on hardware configurations like GPGPU, it is preferable to have algorithms offering a high degree of parallelism at a fine-grain level, while with multi-core configuration, to avoid memory access concurrency problems, it is preferable to have algorithms offering a coarse-grain level degree of parallelism. Therefore, the compromise between the numerical efficiency of a preconditioner and the performance of its algorithm’s implementation for each hardware configuration must be studied. In this paper, we focus on BiCGstab linear solver (used for both symmetric and non-symmetric systems) preconditioned by Neumann Polynomial, BSSOR, ILU, AMG and CPR-AMG [7]. The strong and the weak points of these preconditioners are discussed below. In our study, we use the before-mentioned preconditioners implemented in Hype [8], PETSc [9], PMTL4 [10], MGCSolver [11] and IFPSolver [12] libraries described in Section 4.3.

2.1 Neumann Polynomial

The Neumann Polynomial preconditioner is the simplest polynomial preconditioner. Let \( \omega \) be the largest eigenvalue of matrix \( A \) and \( P_m(A) \) a polynomial of degree \( m \) then \( M^{-1} = P_m(A) \) is defined as:

\[
P_m(A) = \omega(I + (I - \omega A) + \sum_{k=2}^{m} (I - \omega A)^k)\]

This preconditioner – based on matrix-vector products – is very well parallelisable on every kind of hardware configurations. Nevertheless, it is not numerically robust because it does not reduce significantly the condition number of the matrix. For ill-conditioned systems provided by reservoir simulations, it is often not efficient enough numerically to ensure the Krylov solver convergence. On multi-core architectures, this preconditioner is usually not competitive with other preconditioners. However, as its construction is not expensive and it can be parallelized easily at a fine-grain level, the interest to it has increased again few years ago because it is easy to implement for GPGPU, its algorithm being well suited for the GPGPU’s data parallel programming model. Thus, the fact that it requires many more iterations to converge than most of other preconditioners is compensated by the performance of efficient matrix-vector product implementations.

The behaviour of this preconditioner and its handling on the different architectures is discussed in the numerical experimentation Section 4.
2.2 BSSOR Preconditioner

The Block Symmetric Successive Over-Relaxation (BSSOR) preconditioner consists of applying one or several iterations of the block Symmetric Successive Over Relaxation (SSOR) method well described in [1].

Let us consider the linear system $Ax = b$ where $A$ is a matrix, $x$ and $b$ vectors. Be $L$, $D$, and $U$ the lower, diagonal, and the upper parts of $A$ respectively so that $A = L + D + U$. The backward Gauß-Seidel iteration computes $x^{BGS}$ from $x^k$ by solving the upper triangular system:

$$Lx^k + Dx^{BGS} + Ux^{BGS} = b$$

with a backward substitution algorithm.

The forward iteration calculates $x^{FGS}$ from $x^k$ by solving the lower triangular system:

$$Lx^{FGS} + Dx^{FGS} + Ux^k = b$$

with a forward substitution algorithm.

The symmetric version of the method applies successively the backward then the forward iteration as follows:

$$Lx^k + Dx^{BGS} + Ux^{BGS} = b$$

$$Lx^{SGS} + Dx^{SGS} + Ux^{BGS} = b$$

The relaxation step consists in computing the iterate $k + 1$ as the weighted average of the iterate $k$ and the symmetric Gauß-Seidel solution $x^{SGS}$ with a relaxation factor $\omega$: $x^{k+1} = \omega x^k + (1 - \omega)x^{SGS}$.

Although backward or forward substitution algorithms are sequential algorithms, the SSOR algorithm is interesting in its block version because it can be parallelized at a fine-grain level by the means of a block-coloring renumbering procedure (graph-coloring approach). This consists in introducing the dual graph of the matrix and partitioning the vertices into independent blocks of different colors.

Let $G(A)$ be the dual graph of a matrix $A = (a_{ij})$. The vertices of $G(A)$ represent the rows (resp. columns) of $A$. The edges $(i,j)$ connecting the vertices $i$ and $j$ represent the non-zero elements $a_{ij} \neq 0$ of $A$. The coloring algorithms assign to each vertex a color that is different from all its neighbors’ color. Thereby, the number of colors is kept as small as possible by heuristics (finding a coloring with minimal colors is an NP-complete problem [13]). We can easily notice then, that in the backward or forward substitution algorithms, operations on rows of the same color are completely independent as they depend only on vector components of colors different from the current color.

2.3 Incomplete LU Factorization: ILU

The Incomplete LU factorization preconditioners – well described in [38] – are very common preconditioners in reservoir or basin simulators. They apply an incomplete LU factorization of the matrix $A$ to compute a sparse lower triangular matrix $L$ and sparse upper triangular one $U$ such that $A \approx LU$. The factorization is incomplete in the sense that the non-zero $LU$ factors not belonging to the original matrix pattern – the so called fill-in entries -- may be dropped. In the zero degree incomplete version, ILU(0), all these entries are dropped. In the more accurate but more expensive version ILU($k, \tau$), they are dropped regarding two parameters: a maximal fill-in level factor $k$ and a drop-in threshold $\tau$ to filter small entries [1, 37]. The preconditioning operation solves $LUx = y$ with a backward substitution followed by a forward substitution [1].

This preconditioner, well known to be efficient for standard cases (i.e. not ill-conditioned and moderate problem size), is not naturally parallel, since its algorithm is recursive. This algorithm can be parallelized at coarse or fine-grain level with various graph renumbering techniques. In this paper, we study three different strategies of parallelization: the first one is a graph-coloring approach allowing for a high level of parallelism at fine-grain level, and two others for coarse-grain parallelism, the block Jacobi approach and the domain distribution technique, based on 1D, 2D or 3D partitions generated via the libraries like Metis [14] or IFPPartitioner [15].

The graph coloring technique, generally used to improve the level of parallelism at a fine-grain level, is efficient in iterative solution methods for example in the SPMD context. However, in the ILU(0) context, this renumbering technique, modifying the shape of the original matrix graph, has an influence on the rate of dropped fill-in entries during the incomplete factorization. This well-known renumbering effect may decrease the numerical efficiency of the preconditioner, as it may increase the number of iterations to achieve convergence compared to the factorization with the natural ordering. This strategy is adopted in MCGSolver to take advantage of GPU accelerators that require fine-grain parallelism.

Several tests on different cases showed that the block Jacobi ILU(0) preconditioner may suffer from numerical unstability since the links between each domain are not taken into account. To achieve a fully parallel version of the ILU factorization, that takes into account all links between adjacent domains while preserving the parallel performance, the IFPSolver renumbers cells and groups them in interior ones – those not connected to other...
sub-domains – and interface ones – connected to other sub-domain – which are handled in a Schur renumbering style. The interface cells have been sub-grouped into three kinds of interface cells:
- the upstream cells, only connected to domains with a higher rank than the current domain;
- the mixed cells, connected to both domains with rank higher and lower than the current domain;
- the downstream cells, only connected to domains with rank lower than the current domain.

This renumbering technique allowed us to improve the degree of parallelism at coarse-grain level. The computations on interior cells are independent and may be executed in parallel. We obtained a second level of parallelism on interface cells, computing upstream cells then downstream cells independently and optimizing the communication needed to handle the dependencies between sub domains. The whole mechanism of this optimization, applied in IFPSolver, is discussed in detail in [16].

### 2.4 Algebraic MultiGrid

Algebraic MultiGrid (AMG) [35, 37] is a robust preconditioner for elliptic problems. It is appreciated for its extensibility qualities with M-matrix systems: the number of iterations required to converge only depends minimally on the problem size and can be entirely size-independent. This is an important feature for massive parallel applications. However, its construction, much more expensive than traditional ILU preconditioner often suffers of scalability on large number of processors.

The main idea of Algebraic MultiGrid is to remove smooth error components on the coarser grid. This is done by solving the residual equation on a coarse grid and then interpolating the error correction onto the finest grid. The coarsening scheme is the major and crucial part of the AMG preconditioner for both the sequential and the parallel version.

AMG is based on the coarsening heuristics. The traditional coarsening scheme proposed by Ruge and Stüben (RS) [19], tends to work well for two spatial dimensions (2D) problems. Several works, concerning coarsening algorithms, are done in order to adapt them to 3D problems. The following coarsening algorithms, well-known for three-dimensional (3D) problems, are implemented for the parallel environment. The first one, Parallel Modified Independent Set (PMIS) algorithm [20], is a modification of an existing parallel algorithm; CLJP. In this algorithm, the coarse-grid result is independent of the number of processors and the distribution of the points across processors. The second one, the Hybrid Modified Independent Set (HMIS) algorithm [20], is obtained by combining the PMIS algorithm with a one-pass RS scheme.

A depth study in [20] is done concerning the effect of different coarsening schemes on the multi-core architecture, using Hypre BoomerAMG. In our situation, PMIS and HMIS Boomer AMG coarsening algorithms give almost the same result for both total number of solver iterations and total solver time.

### 2.5 CPR-AMG

The CPR-AMG is a two-stage preconditioner which is introduced by Lacroix et al. [7]. This preconditioner, based on the CPR\(^1\) accelerations, extracts and solves pressure subsystems. Then, residuals associated with the computed solution are corrected by an additional preconditioning step on the whole system.

CPR-AMG combines one V cycle of AMG (we can use one of the BoomerAMG from the HYPRE library, SAMG, or at some point PMTL4 AMG) preconditioner on a pressure equation together with an ILU(0) or BSSOR preconditioner on the full system.

A basic presentation of this two-stage preconditioner could be defined as follows: let \(Ax = b\) be the linear system to solve, where \(A\) represents the full system with \(A_{pp}\); pressure block coefficients, \(A_{ss}\); non-pressure block coefficients (saturation block\(^2\) or saturation and concentration block\(^3\)) and \(A_{ps}\); \(A_{xp}\); the coupling coefficients:

\[
A = \begin{pmatrix}
A_{pp} & A_{ps} \\
A_{xp} & A_{ss}
\end{pmatrix}
\]

\[
x = \begin{pmatrix}
x_p \\
x_s
\end{pmatrix}
\quad \text{and} \quad b = \begin{pmatrix}
b_p \\
b_s
\end{pmatrix}
\]

- apply ILU(0) (or BSSOR) on the full system; \(\text{ILU0}(A)x^{(1)} = b\)
- compute the new residual; \(r_p = b_p - A_{pp}x_p^{(1)} - A_{ps}x_s^{(1)}\)
- apply AMG-Vcycle on the residual pressure equation; \(\text{AMG}(A_{pp})x_p^{(2)} = r_p\)
- correction of the pressure unknowns; \(x_p = x_p^{(1)} + x_p^{(2)}\)

This preconditioner is the most expensive preconditioner which permits to decrease the number of iterations

\(^1\) CPR intends to decompose the pressure part (elliptic unknowns) and approximately solve it. The pressure solution is used to constraint the residual on the full system, thus achieving a more robust and flexible overall solution strategy.

\(^2\) In black-oil context.

\(^3\) In compositional context.
to the convergence considerably. In contrast to the Neumann Polynomial, CPR-AMG is expensive to build and apply, but it offers a very fast solution. Thanks to CPR accelerator and to AMG, large-scale and complex reservoir problems, in fully implicit formulation, are solved efficiently.

3 BI-CGSTAB ACCELERATION WITH GPU IMPLEMENTATION

This section examines the advantages and the difficulties encountered by deploying GPU technology to perform sparse linear algebra computations. We study first the Sparse Matrix-Vector product (SpMV) which is the base Kernel, common among all the preconditioners discussed before and two preconditioner based on SpMV: ILU(0), BSSOR and Polynomial. We focus on SpMV and preconditioners because they consume most of the compute time in BiCGStab iterations. Thus, the effort is most important in these components while the vector operations are efficiently enough realized by Nvidia’s CUBLAS library.

3.1 SpMV

Sparse Matrix Vector product (SpMV) on GPU has been widely studied since introduction of General-Purpose GPU (GPGPU) computation [21-23]. Although SpMV exhibits simple and massive parallelism: in $y = A \cdot x$ each $y_i$ computation is independent from each other whereas each $y_i$ is a dot product that also exhibits parallelism; the before-mentioned investigations always show the importance of choosing the right sparse format for the right sparse structure due to the need of massively fine-grain and regular parallelism for efficient GPU use. Thus unstructured sparse formats like CSR or COO do not suit well for GPU computation. Nvidia now provides in its CUDA toolkit SpMV for different sparse formats and this should be considered as a reference.

Although Nvidia provides efficient SpMV for Ellpack and Hybrid format (mix of Ellpack and COO), these SpMV are still designed for generic purpose and do not use linear specific systems that come from the reservoir simulator. To use these specifics, we propose an SpMV that explores the sparse block structure of the matrix $A$ where each non-null element is a small $3 \times 3$ or $2 \times 2$ block, depending on the underlaying Black-oil simulation. By using blocks, we reduce the indirection cost in SpMV: within the sparse matrix $A$, a column index represents 2 or 3 contiguous elements of $x$ so that the effort loading column indices is reduced by a factor of 2 or 3 (Fig. 1). Furthermore, the data reuse is better in block matrices and the access patterns are more regular. We also use a reordering of $A$ that groups lines with the same width (number of non-null blocks). Thus, instead of computing $y = A \cdot x$, our SpMV computes $y = P \cdot A \cdot P^{-1} \cdot x$ with the permutation matrix $P$.

3.2 Preconditioners

We develop an ILU(0), a BSSOR and a polynomial preconditioner for GPU. These three preconditioners exhibit different levels of parallelism: while the polynomial...
preconditioner exhibits a high level of parallelism because it uses the same SpMV as in the Krylov iteration, ILU(0) is based on triangular solve with the incomplete factors $L$ and $U$. The parallelism of the sparse triangular strongly depends on matrix ordering and if we consider “natural” ordering coming from the reservoir simulator ($i,j,k$ grid traversal), the degree of parallelism in the triangular solve is at most the size of the largest sparse row. Such parallelism is insufficient for GPU use.

To go beyond this limitation, a well-known solution is to reorder the matrix according to the node coloring of the associated adjacency graph. Graph coloring consists of assigning a color to each node such that no adjacent node has the same color while trying to minimize the number of colors, i.e., to maximize the number of nodes with the same color. A simple greedy algorithm that tries to minimize the number of colors locally does not necessarily minimize the number of colors globally but provides sufficient results in our context because this algorithm:

- gives two equally sized sets with the large majority of nodes;
- gives additional sets with the remaining nodes while the total number of sets is less than or equal to the graph depth plus one.

However, a drawback is that the number of nodes in additional sets can be very small compared to the first two sets. Also, this greedy algorithm finds the classical red black coloring with graphs coming from structured 2D or 3D simulations.

Once the graph is colored, we reorder the matrix by grouping equations (nodes) of the same color. Figure 2 shows the matrix profile for this reordering: the matrix has $c$ diagonal blocks ($D_i$) where $c$ is the number of colors and other sparse coefficients lie in upper $U_i$ and lower $L_i$ blocks.

With this structure, the triangular solution process $Lx = y$ or $Dx = y$ is decomposed into series of diagonal solver steps where the right hand side is built using the solutions $x_i$ from the previous steps:

$$
D_1 x_1 = y_1 \\
D_2 x_2 = y_2 - L_2 x_1 \\
D_3 x_3 = y_3 - L_3 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\
\vdots \\
D_c x_c = y_c - L_c \begin{bmatrix} x_1 \\ \vdots \\ x_{c-1} \end{bmatrix}
$$

Each solver step is fully parallel because every vector component $x_i$ can be computed independently from another. In practice, we use an SpMV for $z_i = L_i \begin{bmatrix} x_1 \\ \vdots \\ x_{c-1} \end{bmatrix}$ and a diagonal solve for $D_i x_i = y_i - z_i$.

Then the GPU computation of each $x_i$ is efficient if each $D_i$ has a large number of rows.

The GPU version of Algebraic MultiGrid (AMG) preconditioner is under study. Different works [24] and implementations of this preconditioner are already available in different libraries like PETSc [25], NVAMG (GPU implementation) [26] and Trilions [27] (many-core implementation). The integration of these libraries in our simulator is under study, in order to compare their performance with that of the MCGSolver.

### 4 NUMERICAL EXPERIMENTATION

High-performance computing has always been used in the petroleum industry for the simulation at fine scale of large full-field reservoirs. The used parallel reservoir simulator [28] in this paper is a thermal multi-purpose simulator permitting steam injection. It is the new generation IFP Energies nouvelles research reservoir simulator, based on Arcane [29]. In this paper, the strong and the weak points of the before-mentioned preconditioners are reviewed using this simulator.

#### 4.1 Platform Description

The used platform – ener110(4) – is composed of 378 nodes. Each node contains 2 Sandy Bridge octo-cores

---

**Figure 2**

Matrix reordering for GPU triangular solve.
E7-2670 CPUs with a tact rate of 2.6 GHz. The 14 nodes of ener110 have two K20 GPU cards based on the Kepler architecture [30]. The nodes are interconnected with QDR infiniband links (40 Gbit/s) and offer 110 Teraflops.

### 4.2 Study Case Description

The results presented in this paper were obtained on three different variants of the Spe10 model, on the tenth SPE comparative solution project model [31] and on a real thermal test case. Spe10 is a incompressible water-oil, black-oil, two-phase flow problem. It is built on a Cartesian regular geometry with no top structure or faults and simulates a 3D waterflood in a geostatistical model. In this model one water injection well at the center of the reservoir and four production wells at the four corners of reservoir are defined and specified with bottom hole pressure.

- case 1: the original Spe10 model contains more than one million active cells (1 122 000 cells; 61/221/86), with heterogenous porosity and permeability simulating only 30 days;
- case 2: the original Spe10 model with a homogeneous porosity of 0.2 and a permeability of 100 simulating 30 days;
- case 3: fine-scale geological model of Spe10 with 17 952 000 cells (241/881/86) that describes a homogeneous porosity of 0.2 and a permeability of 100 simulating 2 000 days;
- case 4: a thermal model which contains 26 700 cells (6 × 50 × 89), one producer well and one steam injector well (the steam injector is positioned above the producer), with heterogenous porosity and permeability simulating 800 days.

### 4.3 Solver Packages Overview

There exist various software packages for parallel computers which offer different preconditioners. This section presents a brief description of some of these libraries used in this paper.

#### 4.3.1 IFPSolvers-MCGSolver

IFPSolvers (IFPS) and MCGSolver (MCGS) are software packages developed by IFPEN to provide linear solver algorithms for its industrial simulators in reservoir simulation [12], basin simulation [32], or in engine combustion simulation [33]. The provided algorithms aim for efficiency on IFPEN customer hardware configurations like Windows 64 platform, Linux cluster with multi-core nodes partially accelerated with GPGPU. These packages provide some Krylov solvers (BiCGStab, GMRES, CG), some parallel preconditioners like ILU(0), CPR-AMG, BSSOR, PolyN (Neumann polynomial preconditioner), some specific domain partitioner and matrix graph renumbering algorithms. The packages are interfaced with external libraries like Hypre [8] or SAMG [34].

#### 4.3.2 PMTL4

The Parallel Matrix Template Library v4 (PMTL4) [10, 35] provides linear algebra operations on distributed data as a C++ template library. Available data types are distributed vector and sparse and dense matrix types as well as abstractions to conveniently handle distribution and migration. On top of the usual matrix and vector operations, PMTL4 contains a comprehensive set of iterative Krylov subspace solvers including the before-mentioned CGS, GMRES, and BiCGStab. Due to the generic design of the library, all solvers could be directly used from the sequential MTL4 version without re-implementation. Parallelism is introduced here just by re-compilation: if the vectors and matrices have distributed types then the overloading mechanism selects the according parallel operations. The library establishes its own domain-specific language embedded in C++ in order to provide an intuitive, textbook-like interface. However, this user-friendliness does not harm the performance. To the contrary, advanced meta-programming techniques not only avoid run-time overhead but even enable new forms of performance tuning [36]. The future development will include more specialized solutions for applications like the one in this paper. At the same time, those components will be designed in a flexible and combinatorial manner for reusing them productively in further applications.

The current ILU implementation is a block version of ILU(0), i.e. the preconditioner is independently applied on the local unknowns of each subdomain ignoring all entries on interior boundaries. PMTL4 will also provide an interface to Euclid in the near future – the hooks within Euclid are already prepared by David Hysom.

#### 4.3.3 PETSc

The Portable, Extensible Toolkit for Scientific computation (PETSc) is a parallel open-source library (suite of data structures and routines) which permits large-scale application codes on parallel (and serial) architectures to solve their PDE systems. PETSc uses the MPI
standard for all message-passing communication. PETSc has an efficient implementation of Block Jacobi with ILU (0, 1 or 2) subdomain solves. In this paper, we use PETSc version 3.1.

4.3.4 HYPRE

HYPRE is a library for solving large, sparse linear systems of equations on massively parallel computers. HYPRE provides a high-performance parallel Algebraic MultiGrid preconditioner: Boomer AMG, for both structured and unstructured problems. It uses MPI for all message-passing communication. In this paper, we use version 2.0.

4.4 Detailed Results

In this section, we analyze the performance of different preconditioners on the recent architectures: homogenous many-core distributed architectures and heterogenous hybrid ones. We compare our solutions proposed in IFPSolver and MCGSolver with PMTL4 and the well-known and widely used library PETSc. The performed simulations, using all preconditioners of the presented libraries, are performed without any accuracy loss. Regardless of the chosen preconditioner, the simulation performs for each test case the same number of time steps and the same cumulative number of Newton steps. The tables present for an entire simulation the total cumulative number of solver iterations (Tab. 1-4) and total simulation time (Tab. 5) or total solver time (Tab. 6-8) for each used cores number.

4.4.1 Distributed Architecture

In the first step, we analyze the multi-core architecture with large numbers of cores. We choose a large but homogeneous model: case 3. The experimental results\(^6\) are shown in Tables 1 and 5. It can be seen that CPR-AMG offers a robust solution obtained in less time, using less cores. Figure 3 demonstrates that for this homogenous model, Block-Jacobi technique, used in PETSc and PMTL4, offers better scalability, since the links between domains are not crucial and the boundaries communication are avoided. This characteristic can be also seen in Tables 2 and 6.

For ill conditioned matrices – which is the case 1 and the case 4 situation – the proposed ILU(0) by IFPSolver and CPR-AMG offer a better solutions since the inner boundaries are not ignored. The experimentation results shown in Tables 3, 7, 9, 10 confirm that. We demonstrate in Figure 3 and Figure 4 that CPR-AMG offers a fast solution but it is not scalable.

4.4.2 Heterogenous Architecture

In this section, we analyze the performance gain obtained thanks to GPU accelerators. Since GPU

\[^6\] PETSc results are not coherent for 1 024 cores.
TABLE 4  
Case 1: cumulative number of solver iteration during whole simulation using hybrid parallelization

<table>
<thead>
<tr>
<th>Cores number</th>
<th>1c</th>
<th>8c (mpi)</th>
<th>1c/1GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFPS CPR-AMG</td>
<td>1 685</td>
<td>1 235</td>
<td>***</td>
</tr>
<tr>
<td>IFPS ILU(0)</td>
<td>7 749</td>
<td>7 192</td>
<td>***</td>
</tr>
<tr>
<td>PMTL4 ILU(0)</td>
<td>6 892</td>
<td>7 463</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(0)</td>
<td>7 749</td>
<td>8 088</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(1)</td>
<td>4 520</td>
<td>2 410</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(2)</td>
<td>3 809</td>
<td>4 641</td>
<td>***</td>
</tr>
<tr>
<td>MCGS color ILU(0)</td>
<td>12 393</td>
<td></td>
<td>***</td>
</tr>
<tr>
<td>MCGS BSSOR</td>
<td>22 362</td>
<td></td>
<td>***</td>
</tr>
<tr>
<td>MCGS PolyN</td>
<td>12 301</td>
<td>12 160</td>
<td>12 309</td>
</tr>
</tbody>
</table>

TABLE 5  
Case 3: total simulation time (s) using classical parallelization (MPI)

<table>
<thead>
<tr>
<th>Cores number</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1 024</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFPS CPR-AMG</td>
<td>4 346</td>
<td>2 506</td>
<td>2 360</td>
<td>2 477</td>
</tr>
<tr>
<td>IFPS ILU(0)</td>
<td>31 354</td>
<td>17 637</td>
<td>12 124</td>
<td>7 025</td>
</tr>
<tr>
<td>PMTL4 ILU(0)</td>
<td>38 346</td>
<td>15 946</td>
<td>9 373</td>
<td>6 048</td>
</tr>
<tr>
<td>PETSc ILU(0)</td>
<td>44 584</td>
<td>25 134</td>
<td>1 0749</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(1)</td>
<td>32 892</td>
<td>18 863</td>
<td>1 5447</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(2)</td>
<td>37 081</td>
<td>19 528</td>
<td>9 811</td>
<td>***</td>
</tr>
</tbody>
</table>

TABLE 6  
Case 2: total solver time (s) using classical parallelization (MPI)

<table>
<thead>
<tr>
<th>Cores number</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFPS CPR-AMG</td>
<td>84.6</td>
<td>38.2</td>
<td>21.6</td>
<td>21.3</td>
</tr>
<tr>
<td>IFPS ILU(0)</td>
<td>177.8</td>
<td>100.0</td>
<td>51.4</td>
<td>29.4</td>
</tr>
<tr>
<td>PMTL4 ILU(0)</td>
<td>167.8</td>
<td>86.2</td>
<td>48.6</td>
<td>26.6</td>
</tr>
<tr>
<td>PETSc ILU(0)</td>
<td>153.0</td>
<td>105.9</td>
<td>53.1</td>
<td>29.3</td>
</tr>
<tr>
<td>PETSc ILU(1)</td>
<td>103.2</td>
<td>59.8</td>
<td>38.4</td>
<td>23.9</td>
</tr>
<tr>
<td>PETSc ILU(2)</td>
<td>126.5</td>
<td>68.3</td>
<td>47.8</td>
<td>27.2</td>
</tr>
</tbody>
</table>

TABLE 7  
Case 1: total solver time (s) using classical parallelization (MPI)

<table>
<thead>
<tr>
<th>Cores number</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFPS CPR-AMG</td>
<td>151.6</td>
<td>79.7</td>
<td>41.9</td>
<td>49.4</td>
</tr>
<tr>
<td>IFPS ILU(0)</td>
<td>194.0</td>
<td>108.5</td>
<td>57.5</td>
<td>29.8</td>
</tr>
<tr>
<td>PMTL4 ILU(0)</td>
<td>223.4</td>
<td>117.8</td>
<td>63.5</td>
<td>36.3</td>
</tr>
<tr>
<td>PETSc ILU(0)</td>
<td>269.5</td>
<td>206.2</td>
<td>86.0</td>
<td>51.9</td>
</tr>
<tr>
<td>PETSc ILU(1)</td>
<td>233.4</td>
<td>138.0</td>
<td>87.4</td>
<td>51.4</td>
</tr>
<tr>
<td>PETSc ILU(2)</td>
<td>267.9</td>
<td>169.9</td>
<td>82.3</td>
<td>71.8</td>
</tr>
</tbody>
</table>

TABLE 8  
Case 1: performance results (total solver time) of hybrid parallelization (PolyN 8 cores use threads instead of mpi)

<table>
<thead>
<tr>
<th>Cores number</th>
<th>1c</th>
<th>8c (mpi)</th>
<th>1c/1GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFPS CPR-AMG</td>
<td>1 038</td>
<td>280</td>
<td>***</td>
</tr>
<tr>
<td>IFPS ILU(0)</td>
<td>2157</td>
<td>374</td>
<td>***</td>
</tr>
<tr>
<td>PMTL4 ILU(0)</td>
<td>1 796</td>
<td>294</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(0)</td>
<td>2 159</td>
<td>500</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(1)</td>
<td>1 586</td>
<td>205</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(2)</td>
<td>1 793</td>
<td>493</td>
<td>***</td>
</tr>
<tr>
<td>MCGS color ILU(0)</td>
<td>2 144</td>
<td></td>
<td>297</td>
</tr>
<tr>
<td>MCGS BSSOR</td>
<td>8 495</td>
<td></td>
<td>1 129</td>
</tr>
<tr>
<td>MCGS PolyN</td>
<td>2 786</td>
<td>1 035</td>
<td>367</td>
</tr>
</tbody>
</table>

TABLE 9  
Case 4: cumulative number of solver iteration during whole simulation using classical parallelization (MPI)

<table>
<thead>
<tr>
<th>Cores number</th>
<th>1c</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFPS CPR-AMG</td>
<td>20 898</td>
<td>22 759</td>
<td>24 413</td>
<td>25 052</td>
</tr>
<tr>
<td>IFPS ILU(0)</td>
<td>278 413</td>
<td>296 243</td>
<td>308 324</td>
<td>310 157</td>
</tr>
<tr>
<td>PMTL4 ILU(0)</td>
<td>155 643</td>
<td>205 361</td>
<td>217 348</td>
<td>239 380</td>
</tr>
<tr>
<td>PETSc ILU(0)</td>
<td>204 686</td>
<td>248 023</td>
<td>263 654</td>
<td>285 464</td>
</tr>
<tr>
<td>PETSc ILU(1)</td>
<td>133 978</td>
<td>229 705</td>
<td>258 613</td>
<td>283 661</td>
</tr>
<tr>
<td>PETSc ILU(2)</td>
<td>114 628</td>
<td>217 700</td>
<td>257 875</td>
<td>282 339</td>
</tr>
</tbody>
</table>
preconditioner are based on SpMV, we first present synthetic SpMV performances on CPU and GPU. These days, the cost of one CPU and one GPU is quite similar. Therefore the following question makes sense: “is it interesting, in term of performance, to have a GPU accelerator instead of an additional CPU?”

**SpMV performances**

In Figure 5, we compare the performance of SpMV developed at IFPEN and Nvidia’s cusparse SpMV for CSR, BSR, Ellpack and Hybrid formats running on Fermi C0270 and Kepler K20 boards against a CPU SpMV running on one Xeon Sandy-Bridge processor using between one and eight cores. The CPU SpMV implementations are block-CSR algorithms developed for the MCGSolver and the block-CSR from the Intel MKL library. We use matrices ranging from $2 \times 10^4$ to $2 \times 10^6$ equations extracted from IFPEN reservoir simulator. These results show the situations:

– where the matrix is small enough (Canta, $2 \times 10^4$ eq.) to fit into L3 or L2 processor caches. Then, the CPU performance is with 4 and 8 threads and is better than the best GPU version. Also, multi-core CPU SpMV offers good scaling with speed-up of 3.2 and 6.1;

– where the matrices are too large to fit into CPU caches. Then all GPU SpMV – except CSR – offer a significant performance boost over CPUs, even with 4 and 8 cores. Acceleration compared to 8 cores varies from 1.9 to 3 for the K20 GPU.

Also block CSR from either MKL or Cusparse does not improve performance against CSR except for Cusparse BSR when block size is $2 \times 2$ or $4 \times 4$ on K20 but never competes with Cusparse Ellpack or Hybrid format except for MSUR_9 matrix.

**Reservoir simulator results**

We present simulation results for Case 1 and Case 4 which show two different behaviours for GPU solver acceleration.

For Case 1, performance results presented in Tables 8 and 4 highlight the obtained benefit by using the hybrid CPU + GPU implementation of ILU(0) compared to ILU(0) using 8 cores of a CPU. This table shows that for the ill conditioned Spe10 system, the hybrid CPU + GPU offers a better performance, even if the number of solver iterations is more important, due to the graph-coloring, than the traditional version implemented in IFPSolver, PETSc and PMTL4. Note that CPU + GPU solver times include all data setup and transfer to/from GPU. For polynomial preconditioner, we have
a convergence rate equivalent to GPU color ILU(0) but with a higher algorithmic iteration cost: the polynomial degree is 2 which means 2 SpMV per preconditioner call. This cost is at least twice the cost of an ILU(0) solve; anyway GPU polynomial preconditioner is competitive against all one core CPU preconditioners and close to some eight cores CPU preconditioners. On the other side GPU BSSOR has higher iteration cost and also slower convergence rate thus provides small improvement against one core CPU preconditioner and no improvement against eight cores one. It is also interesting to note that CPU + GPU solver/preconditioner have an acceleration factor close to 7.5 against their CPU counterpart.

**Table 11**

Case 4: performance results (total solver time) of hybrid parallelization (PolyN 8 cores use threads instead of mpi)

<table>
<thead>
<tr>
<th>Cores number</th>
<th>1c</th>
<th>8c(MPI)</th>
<th>1c/1GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFPS CPR-AMG</td>
<td>858</td>
<td>185</td>
<td>***</td>
</tr>
<tr>
<td>IFPS ILU(0)</td>
<td>1489</td>
<td>226</td>
<td>***</td>
</tr>
<tr>
<td>PMTL4 ILU(0)</td>
<td>4842</td>
<td>791</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(0)</td>
<td>3348</td>
<td>575</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(1)</td>
<td>4139</td>
<td>772</td>
<td>***</td>
</tr>
<tr>
<td>PETSc ILU(2)</td>
<td>6054</td>
<td>1172</td>
<td>***</td>
</tr>
<tr>
<td>MCGS color ILU(0)</td>
<td>3376</td>
<td>738</td>
<td></td>
</tr>
<tr>
<td>MCGS BSSOR</td>
<td>6858</td>
<td>1370</td>
<td></td>
</tr>
<tr>
<td>MCGS PolyN</td>
<td>4207</td>
<td>878</td>
<td>788</td>
</tr>
</tbody>
</table>

**Table 12**

Case 4: cumulative number of solver iteration during whole simulation using hybrid parallelization

<table>
<thead>
<tr>
<th>Cores number</th>
<th>1c</th>
<th>8c(MPI)</th>
<th>1c/1GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFPS CPR-AMG</td>
<td>2089</td>
<td>2413</td>
<td>***</td>
</tr>
<tr>
<td>IFPS ILU(0)</td>
<td>2784</td>
<td>308</td>
<td>324 ***</td>
</tr>
<tr>
<td>PMTL4 ILU(0)</td>
<td>155</td>
<td>217</td>
<td>348 ***</td>
</tr>
<tr>
<td>PETSc ILU(0)</td>
<td>204</td>
<td>263</td>
<td>654 ***</td>
</tr>
<tr>
<td>PETSc ILU(1)</td>
<td>133</td>
<td>258</td>
<td>613 ***</td>
</tr>
<tr>
<td>PETSc ILU(2)</td>
<td>114</td>
<td>257</td>
<td>875 ***</td>
</tr>
<tr>
<td>MCGS color ILU(0)</td>
<td>362</td>
<td>361 ***</td>
<td></td>
</tr>
<tr>
<td>MCGS BSSOR</td>
<td>383</td>
<td>381</td>
<td>337 ***</td>
</tr>
<tr>
<td>MCGS PolyN</td>
<td>387</td>
<td>388</td>
<td>983 ***</td>
</tr>
</tbody>
</table>

Figure 5

SpMV on a) C2070 and b) K20 compared to Intel Sandy-Bridge XEON E5-2680@2.7 Gzh.
For Case 4, as shown in Table 7 and 8, CPU + GPU solvers offer an acceleration between 4.6 and 5.3 against their one core CPU counterpart but do not compete with best eight cores MPI solvers (IFPS CPR-AMG and IFPS ILU(0)). This is due to the relatively small case size (≈ 10^4 solver unknows) where the parallel CPU solvers gain efficiency as data set fits in processor caches but also to a worse numerical behaviour of GPU preconditioners. As an example for IFPS ILU(0)/ MCGS color ILU (0), we see a 1.6 increase factor for number of iteration in case 1 while the same factor is 2.3 in Case 4. For GPU polynomial and BSSOR preconditioner the observed behaviour is close to the one for Case 1: GPU polynomial is close to color ILU(0) and GPU BSSOR provides no performance improvement.

CONCLUSION AND FUTURE WORKS

The purpose of our article was to evaluate the performance of the linear solvers used in large parallel reservoir simulators on new hardware configurations. We have focused our study on two kinds of configurations: large memory distributed architectures with a large number of cores and heterogeneous nodes with multi-core CPUs accelerated with GPGPU cards. We evaluated the scalability of a BiCGStab solver with different preconditioners of the IFPS, PETSc and PMTL4 solver packages, with some runs on 128 up to 1 024 cores, on a 17 950 000 grid block study test case. For this kind of architecture, CPR-AMG is the most efficient preconditioner mainly due to its extensibility qualities even if its construction suffers of strong scalability problems. The preconditioners of the ILU family, well parallelized by means of renumbering techniques, have a good scalability behaviour although they numerically suffer of lack of extensibility.

With heterogeneous multi-core nodes accelerated with GPGPUs, we have tested new solvers and preconditioners, which are based on SpMV, taking advantage of GPGPU accelerators. These algorithms turn to be competitive with respect to the best CPU ones using one core and in some cases against eight cores even with worse convergence rates. Also with good performance of GPU SpMV, we expect that SPAI preconditioners [38] will provide better convergence rates with the same level of performance for solver iteration. Our results demonstrate that using GPGPUs becomes now a real alternative to multi-core nodes for solving ill-conditioned linear systems with Krylov solvers in reservoir simulation. Once we proved the efficiency of the GPU implementation, the next step will be to compare our solution with other libraries. Different implementations are available in different libraries like PETSc [25] and Trilions [27] (many-core implementation).

The integration of these libraries in our simulator is under study.

To handle Petascale and later Exascale architectures, we need to improve the scalability of the CPR-AMG preconditioner beyond 1 024 cores up to at least 10 000 cores in order to have both weak and strong scalability. Within nodes, the use of accelerators in heterogeneous architecture becomes a heavy trend, for energy and economical reasons. Right now, the preconditioners designed for GPGPU become competitive with respect to the best CPU one. Nevertheless, to really take advantage of the computation power offered by accelerators, we need to design new algorithms which are numerically more efficient to reduce the number of iterations, as for preconditioners like CPR-AMG. As this last one is not adapted for fine grain size parallelization, we plan to consider other new multi-level methods, based on domain decomposition algorithms which are promising as they present both numerical robustness and are potentially more easier to accelerate with GPGPU.

REFERENCES

4 http://www.khronos.org/opencl/.