Oil & Gas Science and Technology – Rev. IFP Energies nouvelles (2016) **71**, 63

DOI: 10.2516/ogst/2016018

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

Analysis of IDR(*s*) Family of Solvers for Reservoir Simulations on Different Parallel Architectures

Vincent Seignole*, Jean-Frédéric Thebault and Raouf Nabil

Storengy, 12 rue Raoul Nordling, CS 7001, 92274 Bois-Colombes Cedex - France
e-mail: vincent.seignole@storengy.com - jean-frederic.thebault@storengy.com - raouf.nabil@storengy.com

* Corresponding author

Abstract — *The present contribution consists in providing a detailed analysis of several realizations of the IDR*(s) *family of solvers, under different facets: robustness, performance and implementation on different parallel environments in regards of sequential IDR*(s) *resolution implementation tested through several industrial geologically and structurally coherent 3D-field case reservoir models. This work is the result of continuous efforts towards time-response improvement of* Storengy's *reservoir three-dimensional simulator named* Multi, *dedicated to gas-storage applications.*

Résumé — **Analyse de la famille de solveurs IDR(s) pour la simulation de réservoir sur différentes architectures parallèles** — Nous présentons ici une analyse détaillée de différentes mises en œuvre de la famille de solveurs IDR(*s*), sous différentes perspectives : robustesse, performance et implémentation dans différents environnements parallèles au regard du solveur séquentiel IDR(*s*) avec divers tests sur des données de cas réels industriels grandeur nature, géologiquement et structuralement cohérents de modèles réservoir 3D. Ce travail intègre des travaux plus globaux d'optimisation du temps de réponse du simulateur de réservoir tri-dimensionnel de *Storengy*, nommé *Multi*, et spécialisé pour les problématiques de stockage de gaz naturel.

INTRODUCTION

The industrial problem we deal with is the management of geological natural gas storage utilities, composed of underground storage reservoir and its associated surface facilities. Efficiently managing and operating such utilities requires a thorough modeling of both underground two-phases flows and surface systems.

Regarding the reservoir simulation, many concerns are shared with the oil and gas industry, but several ones are emphasized in the context of natural gas storage:

- underground flows are coupled with highly-varying boundary conditions at well heads,
- long historical production data must be taken into account for the history matching phase (usually 50 years, with *e.g* weekly production data variability).

These points lead to long unitary computation times of the simulator: despite the use of moderate size meshes (considering 100 000 ~ 200 000 cells as order of magnitude), time steps must be maintained small enough in accordance with timed-data needed to be matched. This often yields, with a non-parallel version of our simulator, simulations spanning several days.

To enable accurate reservoir simulator predictions, finer quantified knowledge of reservoir attributes, known originally with a given level of uncertainties, must be obtained. This is achieved by history matching of reservoir models, allowing to refine our knowledge regarding distribution of porosities, permeabilities, position of geological faults. This is a very iterative task (manually or automatically realized), but in any case this is costly in simulation time.

The paper is organized as follows: we first provide some figures on the physical and numerical modeling implemented in the Multi reservoir simulator, as well as figures regarding typical simulations performed through sequentially resolution methods. Then, we describe how the solver component, main bottleneck in the runtime performance of simulator, could be considered for improvement, by considering improved parallel methods, in the continuity of the existing solvers in our simulator: the IDR(s) family of methods for solving large sparse linear systems. The next sections will focus on comparing and analyzing implementations of the solver implemented for CPU (Central Processing Unit), GPU (Graphical Processing Units), and cluster platform. We will emphasize the relevance of GPU for our main concerns: industrial 3D-field case reservoir simulation running time reduction. We will then conclude with some perspectives.

1 PHYSICAL AND NUMERICAL MODELING IN MULTI

Multi is a specialized reservoir simulator, taking into account aspects relevant to natural gas storage activities in deep porous aquifers. It supports two-phase, multi component miscible gas compositional, reactive flows in porous media (water and natural gas).

We present in this section a short highlight of Multi modeling features.

1.1 Problem Modeling

The problem we consider is modeled first by its geometry: the reservoir geometry, and the wells locations. The reservoir is characterized by: its top and bottom, sides, not necessary with a uniform elevation and including geological faults. The wells are characterized by their position at surface, and their intersection with the reservoir.

The other reservoir attributes are porosities, permeabilities, relative permeabilities (to water and gas) and capillary pressure ($P_c = P_g - P_w$).

For the sake of simplification, we do not detail the compositional modeling hereafter, and concentrate on the basic two-phase flow formulation. The porous transport of each phase is taken into account by one transport equation for each phase:

$$\frac{\partial}{\partial t}(\rho_w S_w \phi) + \mathrm{div}(\rho_w U_w) = q_w \quad (\text{for water})$$

and

$$\frac{\partial}{\partial t}(\rho_g S_g \phi) + \mathrm{div}(\rho_g U_g) = q_g \quad (\text{for gas})$$

where ϕ denotes the medium porosity, ρ_w, ρ_g the densities for the water and gas phases, S_g, S_w the saturations of the respective phases, which obey $S_g + S_w = 1$. U_g and U_w correspond to the phase velocities, and q_g and q_w are source terms. These velocities are in turn closed by Darcy's law for each phase, and the densities are closed via real thermodynamic state laws: $\rho_g = \rho_g(P_g, T)$, $\rho_w = \rho_w(P_w, T)$, where P_g, P_w denote each phase pressure (related by the capillary pressure), and T the temperature (supposed constant and common to the two phases).

This yields two primary unknowns: the gas saturation and its pressure (S_g, P_g).

1.2 Space/Time Discretisation

The reservoir is discretised by a mesh built from layers of elementary regular hexaedral volumes, which can be locally refined. Generally, our meshes are built using a 3D Geomodeler.

The space discretisation is based on a robust cell-centered finite-volume scheme.

The time discretisation is implicit, and handled by an implicit Euler time-marching scheme, and each time step is solved by a Newton-Raphson iterative process, involving a sparse linear system. The linear systems are (originally) solved by an Induced Dimension Reduction (IDR) [1] preconditioned by an incomplete LU factorization, which is ILU(k), k corresponding to a L and U filling level [2].

1.3 Sparse Matrix Characteristics and Profile

During the iterations described above, sparse linear systems arise, according to the problem discretisation: the matrices are obtained by integrating derivatives of fluxes between cells, source terms contributions. Cells are initially renumbered, and the matrix lines are normalized, yielding diagonal values constant equal to 1.

In Figure 1, which presents two matrices via their profiles (corresponding to two actual 3D-field case reservoir models).

The first one (issued from a standard two-phase model) has 136 710 lines, 918 840 non-zero coefficients, and the second one (issued from a compositional model simulation) has 730 493 lines, 18 million non-zero coefficients.

In the following, we will consider two leading linear systems based on these matrices described and we will denote them as system 1 (using the first matrix), and system 2 (using the second matrix).

These matrices are non-symmetric and not diagonally dominant. For typical simulations, the condition number of the matrices can be up to an order of magnitude of 10^5. This condition number depends upon several parameters: the iteration time-steps, the geometry, as well as the flow configuration.

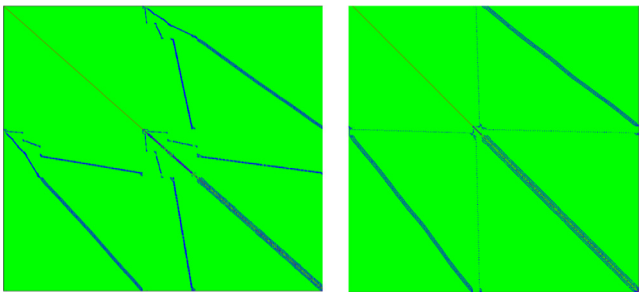

Figure 1
Example sparse matrices profiles.

In practice, we consider few reference meshes for reservoirs but coming from actual industrial 3D field-case data sets geologically consistent with more than hundreds km^2 global areal extension. Geological consistency of the reservoir gridding is a key point to ensure the simulation runs will present consistent and predictive results provided a good fit of the reservoir history production. Practically we did seek to improve computational performance in this context: hence the size of the problems we consider is directly the one defined and daily used by the practice of geologists and reservoir engineers. As the linear systems we are confronted to along our reservoir simulations are of moderate size, it is highly challenging to efficiently parallelize in this range of moderate data volumes and obtain significant running time reduction of simulations when comparing to sequential solver running time of the same reservoir data set conditions and constraints. Indeed, when decomposing the work to handle resolution in parallel, we easily reach low-granularity local computations, whose execution time reach the same order of magnitude than the communication and synchronization costs.

It is to be noted that having reached a certain ratio of performance on these industrial 3D field-case data sets, one may seek for the model size limit that shows no more running time performance reduction but still giving accurate and predictive results. However the main drawback of these downsizing scale of the reservoir dataset is the geological likelihood of the refined datasets. As petrophysical reservoir properties are obtained from interpolated well data measurements with the care of preserving the geostatistical coherence of the reservoir properties (geological facies correlation length). Highly refined grids are in general rather conceptual datasets far from a reservoir geological consistency.

2 SELECTED FAMILY OF SOLVERS

In order to improve our sequential linear solver component performance, we evaluated, as an evolution to the existing one, the family of IDR(s) solvers [3] to take over the sparse linear systems resolution in Multi, and realize them with parallel implementations.

More precisely, we considered both the IDR(s) [3] and IDR(s)-biortho [4] / IDR(s)-minsync [5] method for implementation.

Let us consider the following notations for the linear systems:

$$Ax = b$$

where A is a squared sparse matrix of size $n \times n$, x is the unknown (vector of size n), and b the right hand side.

2.1 IDR(s) Base Method

The IDR(s) method is an extension of the IDR method. We direct the reader to the original paper, but still provide a synopsis of the IDR(s) method hereafter.

It consists, in the scope of Krylov methods, to identify nested sub-spaces, with decreasing dimensions. Such sub-spaces, called Sonneveld spaces, are defined in the following manner:

Let G_0 be the total Krylov space generated by the matrix A, and the residual $r_0 = b - Ax_0$, x_0 being the initial start point of the algorithm. And let S be a vector subspace of R^n, such that $S \cap G_0$ does not contain a sub-space being invariant by A. Let also define $G_j = (\mathrm{I} - \omega_j A)(\mathrm{G}_{j-1} \cap S)$, where I is the identity matrix, and the ω_j's are non-zero coefficients (these coefficients are defined in [3]).

Then, these spaces have the following property:
- $G_j \subset G_{j-1}$ unless $G_{j-1} = \{0\}$,
- $G_j = \{0\}$ for some $j \leq n$.

Then, the IDR(s) method consists in establishing a Krylov-type method such that the sequence of residuals reside in the G_j subspaces.

2.2 IDR(s)-Biortho / IDR(s)-Minsync

The concept of IDR(s)-biortho is, by applying the same framework as IDR(s), to make this method more robust by applying a bi-orthogonalization of the basis vectors for Sonneveld spaces.

We direct the reader to the reference papers for the details.

A version of IDR(s)-biortho optimised to reduce the number of communications in case of parallel implementation is denoted by IDR(s)-minsync, as described in [5].

2.3 First Comparative Analysis (Robustness and Convergence Rate)

Before going further, we want to stress out that we have considered the use of these solvers without the conjunction

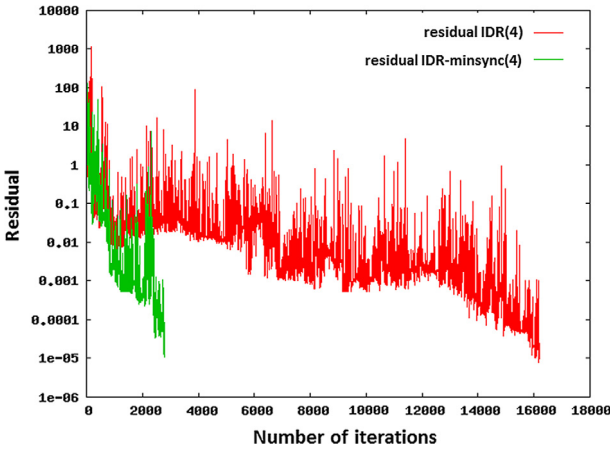

Figure 2

Comparison of residuals of IDR(4) and IDR(4)-biortho on system 1.

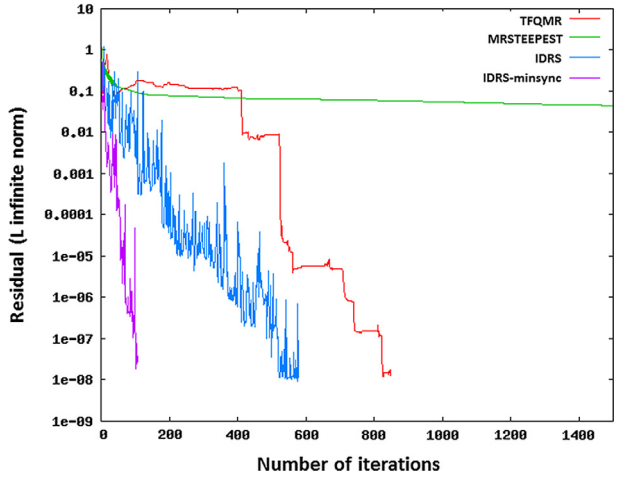

Figure 3

Comparison of residuals on system 2 (IDR(s) with s = 4).

of additional pre-conditioners (*i.e* in additional to the renumbering and re-normalization of the matrix described in Sect. 1.3). This means that no initial ILU preconditioner has been used, compared to the initial solver we sought to improve which was used in conjunction with such a preconditioner.

All the results we provide in the paper regarding the results of applying IDR(s) methods do not involve more preconditioning steps than renumbering and normalization of the linear system.

The IDR(s) basic method was described in the initial paper by Sonneveld and Van Gijzen [3] as sometimes suffering from residual stagnation. We experienced this, despite the application of the patches suggested in that paper to cure these issues, and this is illustrated in Figure 2.

The residual stagnation case of IDR(s, s = 4) is found here, concretized by a very large amount of iterations to achieve the prescribed convergence criterion (10^{-5}). In contrast, the bi-orthogonalization technique allows to the algorithm to converge in a much shorter number of iterations.

Also, on the system 2 described in Section 1.3, we observed (and also consistently on other linear systems in our simulations) that the IDR(s)-biortho/IDR(s)-minsync outperforms the initial IDR(s) algorithm.

In Figure 2, we observe that the IDR(s)/biortho-minsync residual convergence rate is far greater than the one of IDR(s), which in this case does not suffer from residual stagnation.

For the sake of additional information, we have also added in Figure 3 observed residuals of two other kinds of solvers: TFQMR and MR_STEEPEST [2] (a non-Krylov solver). The results regarding these solvers have been provided to position various types of solvers.

The IDR(s) authors compared IDR(s) with GMRES in the paper [3]. In terms of number of matrix-vectors product to reach convergence, GMRES performs better than IDR(s). But on one hand, GMRES is not a limited-memory algorithm, and on the other hand, is known to have time-costly iterations.

3 ANALYSIS OF THE RESULTS USING VARIOUS PARALLELIZED IMPLEMENTATIONS

In order to speed up the computations, we have evaluated four programming paradigms, and some of their variants. First, a short synopsis of these techniques is provided in Table 1.

We have implemented exactly the same algorithm with each of these techniques. This allowed us to target tangible comparisons between the parallel programming strategies. We also did not use black-box solver libraries, in order to precisely understand the performance of the solver in the considered various setups.

After giving some technical details on these implementation techniques, we will jump to the performance results on our two leading test cases, and provide some analyses.

3.1 Description of Implementation Techniques

3.1.1 Multi-Threaded Programming

Since almost 10 years, multi-core general purpose processors (multi-core CPU) have become ubiquitous, and offer some general-purpose parallelism with shared memory.

TABLE 1
A short synopsis of programming paradigms

Programming paradigm	Description
Sequential implementation	Direct mapping of algorithm description to code, with sequential linear algebra primitives applied on large vectors (simple loops), and sparse-matrix times vector operations.
Multi-threaded/Multi-core implementation	Linear algebra primitives implemented with Linux POSIX threads.
GPU implementation	Memory for iteration vectors, sparse matrix was allocated in the GPU accelerator, and linear algebra primitives done in the GPU, with a program on CPU acting as master.
Cluster-programming	Memory for iteration vectors is distributed in different processes spanning several machines, sparse matrix lines bands are also distributed, and communication is implemented between processes to consolidate needed data for each process to proceed.

Hence, our first implementation technique targeted the use of threads to parallelize the considered algorithms. Threads are concurrent flows of executions as part of a process, and each can use a different processor core (this can be imposed by the programmer, or we can let operating system scheduler handle it). Threads share the process data memory, which is handy for parallel programming.

Our algorithm parallelization work has been done by parallelizing elementary linear algebra operations on intermediate vectors of size n and on sparse matrix times vector product.

We dealt with a C-language based implementation of IDR(s) and IDR(s)-minsync, and used the Linux POSIX threads to activate several threads of execution in the scope of the solver.

In order to implement an optimized management of threads, we use on one hand a collection of pre-activated threads, and on the other hand a lock-free synchronization technique to get rid of the associated synchronizations costs implemented at the operating-system level. These lock-free techniques we have adopted rely on Linux spinlocks [6] and on full memory barriers [7] (as provided by the GCC compiler with the __sync_synchronize() operation).

3.1.2 GPU Programming

In the last few years, we have also seen the emergence of end-user programmable GPU, with two global providers: *NVIDIA* and *AMD*. GPU have a specific architecture, and are mainly used as accelerators that can be coupled with CPU-based software. They embed many processing units, and several Giga bytes internal shared memory. Their memory throughput is of a larger order of magnitude than current CPU.

We refer the reader to references [8, 9] for already work done these last years to build and evaluate GPU based solutions for implementing linear solvers [8, 9].

We implemented the IDR(s)-minsync algorithm as a combination of CPU code and calls to necessary linear algebra primitives on a GPU (thanks for NVIDIA Cuda [10], NVIDIA cuBlas [11], and cuSparse [12] libraries).

In Figure 4, we illustrate the "master-slave" programming model we have adopted. C-language based code implementing the solver logic is executed on the CPU. Vectors and Sparse matrix are allocated in the GPU internal (global) memory, and copied from the host central memory. The linear algebra computations primitives are delegated to implemented functions (on the GPU), thanks to the cuBlas and cuSparse primitives.

3.1.3 Cluster Programming

The traditional parallel programming setup consists of implementing communicating processes over a network of servers, with a distribution of input/intermediate/output data.

We selected the OpenSHMEM standard middleware [13] for data distribution, and in practice used the OpenSHMEM implementation part of OpenMPI [14]. OpenSHMEM implements a logically shared-distributed memory paradigm (Fig. 5).

We have used the METIS [15] library in order to decompose the sparse matrix, re-index it, and allocate the data onto the various processes distributed on a cluster.

Our implementation allows to use also multithreading for local parallel processing on each process data. This yields an hybrid distributed/multithreaded approach.

3.2 Discussion of Time and Scalability Results Obtained with these Techniques

3.2.1 Report on the Scalability Results Obtained with a Multi-Thread Approach

The reference hardware for these tests is made up of two 6-cores CPU Intel E5-2643 v2 (with 59 Go/s theoretical peak memory throughput), managed by Linux Centos 6.3.

In Figure 6, we considered the resolution of linear system system 1 and system 2 by our IDR(s)-minsync multi-threaded implementation, and increased the number of threads. We represent there: the duration time we would ideally expect for these computations, and the actually observed duration times.

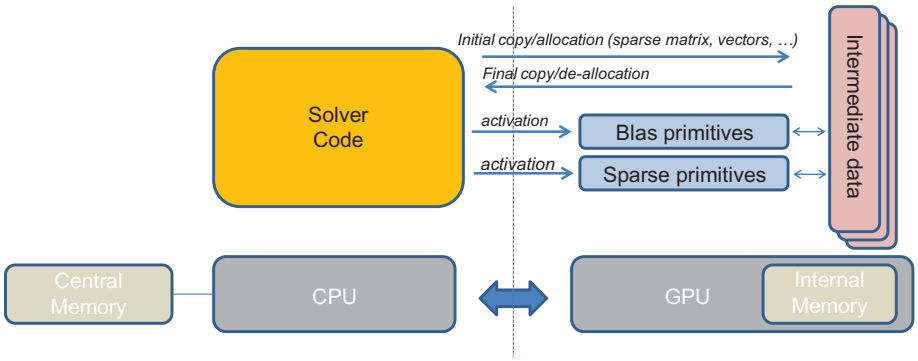

Figure 4

GPU acceleration of linear primitives.

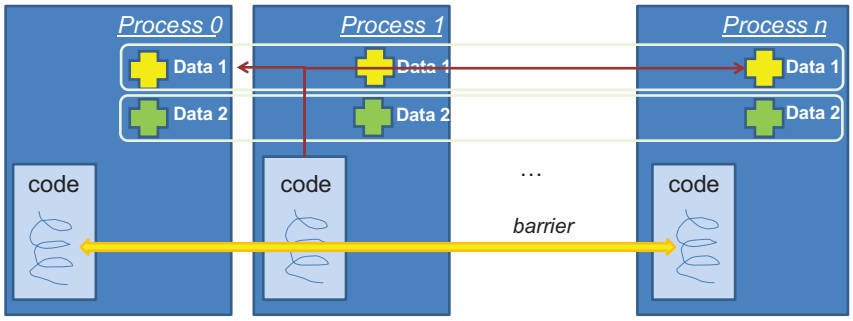

Figure 5

OpenSHMEM paradigm illustrated.

Our main statement is that there is a strong departure from ideal speedup, starting at 3 concurrent threads. In the case of system 1, the time decreases until 7 threads. In the case of system 2 (of larger size than system 1), the behavior appears even worse.

Best stated speedups in these cases are respectively 2.7 and 2.6.

For the sake of additional information, we also investigated multithreaded linear primitives available in Intel MKL library, as an alternative to our own multi-threading mechanisms implementation, which globally demonstrates the same behavior.

We observed that for our particular cases, we got a slight performance advantage with our multithreading technique over Intel-MKL ones. MKL relies on Intel compiler OpenMP support, which targets general setups, and is optimized globally for best performance, whereas our implementation is dedicated to our needs, and was slightly superior in the scope of our specific range of vectors sizes.

To understand these results, we devised a synthetic multithreaded benchmark implementing series of linear primitives on vectors, parameterized by a reference size n of vectors. We measured the computation times $d(k, n)$ of this synthetic benchmark by varying the number of threads k from 1 to 8, and n from 1 to 1 million. We identified that the ratio $r(k, n) = \frac{d(1,n)}{d(k,n)}$ when k is fixed, degrades with (except for $k = 1$) and with n large, it quickly reaches an upper limit far as k increases. In particular, we found out that $r(8 \times 10^6) \sim 2.25$, which is an order a magnitude consistent with the results presented in Figure 6.

As a consequence of this synthetic test, we deduce that the large departure from ideal speedup found here has its origin in memory throughput bottleneck to reach (shared) L3 cache and central memory.

When analyzing the implemented solver algorithm, we find that it is mainly memory-bound.

3.2.2 We Now Report on the Results Obtained by the GPU-Based Approach

The reference GPU used for this benchmark is a Fermi Quadro 6000 NVIDIA GPU with a core clock at 574 MHz,

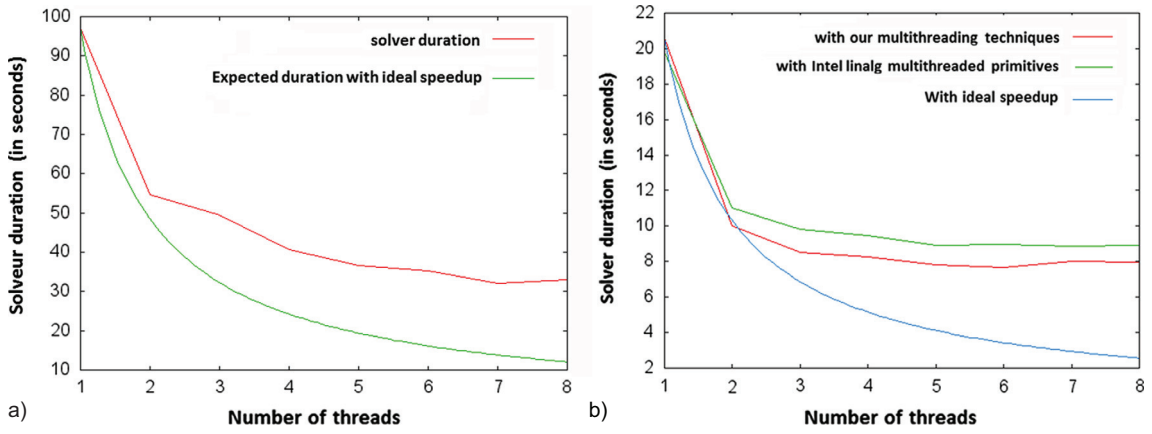

Figure 6

Duration *versus* number of threads for system 1 and system 2.

TABLE 2

Results obtained by comparing GPU implementation of IDR(*s*)-minsync and sequential implementation of IDR(*s*)

System	Number of iterations	IDR(4)-minsync sequential (CPU)	IDR(4)-minsync multithread best (CPU)	IDR(4)-minsync GPU
System 1	2803	50 s	27 s	12.84 s
System 2	62	12.02 s	5.97 s	2.08 s

6 Go internal memory, and having a peak memory bandwith of 144 Go/s.

The results of applying the IDR(*s*)-minsync GPU implementation, compared to the sequential implementation of IDR(*s*) are provided in Table 2.

These test cases reveal that the GPU implementation is shorter in term of execution delay: on one hand, there are more parallel computational resources on the GPU, and on the other hand, we benefit from a more integrated memory system architecture with a significantly higher throughput.

We cannot, with the implementation we have setup, provide scalability numbers regarding to the usage of GPU ressources *per se*, since the cuSparse and cuBlas NVIDIA libraries handle themselves an optimized allocation of its functions on the GPU: the programmer cannot configure these characteristics.

3.2.3 Results Obtained by the Cluster-Based Approach

Here, our experiments were performed on a cluster of machines, interconnected with an Infiniband QDR network.

We now deal with system 2 linear system case.

First of all, we show in the Figure 7 graphs regarding the obtained computation times, by increasing the number of processes in the parallel implementation of the solver. To avoid memory concurrency issues as for the multi-thread implementation, and in order to facilitate the understanding of the obtained performance, we allocated for this test only one process per cluster server, meaning that a single worker runs on each server.

Our first statement is that we manage to get better speedup than in the multi-threaded case, even when using the same number of logical resources, that is, the same number of computational cores (but allocated on one node *versus* several nodes). For example, for 8 processes, the obtained speedup here is 4.46, whereas in the best case in the multi-threaded setup, we managed a maximum speedup of 2.6.

Our implementation in the cluster case is not fully optimal yet. By an analysis of the communications graph, we identified that further work is needed in order to minimize the number of communications.

3.3 Comparison with Original MULTI Solver (IDR+ILU)

We provide here some comparison figures about our IDR(*s*)-minsync implementation for the GPU, and the initial IDR +ILU sequential implementation.

To do that, we consider the first 64 linear systems encountered in an real-world reservoir simulation, from which our leading test case system1 is extracted.

We represent in Table 3 the cumulated time spent in these respective solvers, and present the repartition of individual

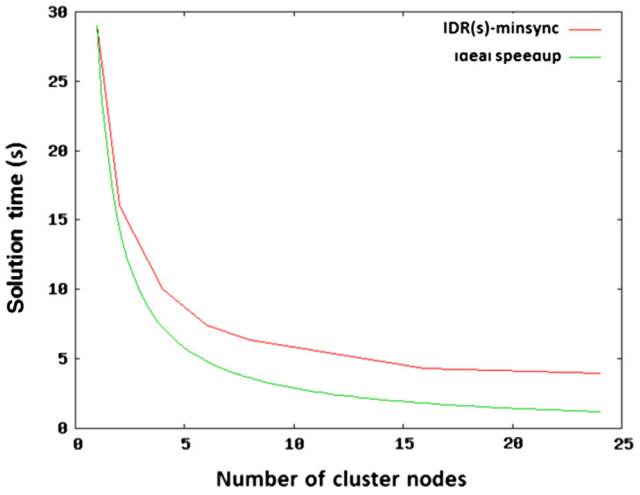

Figure 7

Resolution time with regard to number of cluster nodes used, 1 process per node.

TABLE 3

Cumulated time spent in the solvers

Cumulated time in IDR(4)-minsync on GPU	Cumulated time in sequential IDR+ILU (including factorization)
115 s	413.5 s

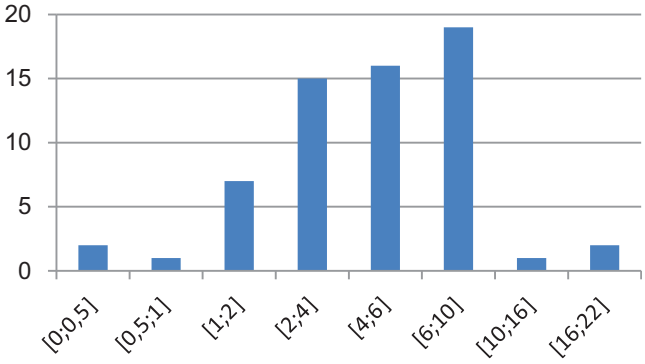

Figure 8

Repartition of individual speedups on 63 first linear systems.

execution times ratio in Figure 8: we present there, for the 64 considered linear systems, the population in different performance improvement ratio classes. In this figure: [0:0.5] and [0.5:1] are two classes in which our IDR(*s*)-minsync on GPU implementation does not outperform the original solver. But the majority of performance improvement ratio sits in the [2:10] interval.

CONCLUSION

In this paper, we have demonstrated the relevance of the IDR (*s*)-biortho (that we implemented in the IDR(*s*)-minsync version) solver for reservoir simulations.

Our study leads us to the following main conclusions.

First, we managed to use the IDR(*s*)-biortho (implemented as IDR(*s*)-minsync) algorithm on several of our actual industrial cases without the use of complex preconditioners. This demonstrates the high robustness of this method.

Second, with a GPU implementation of the solver, we managed to outperform the sequential IDR+ILU traditional preconditioner/solver combination of our reservoir Simulator Multi, and this allowed for overall reservoir simulation times reductions by a factor of about 2 or 3, the remaining (*i.e* external to the solver) sections of the software still being non-parallel.

Third, the GPU implementation reveals in practice a quite adequate tradeoff (for the problems sizes we currently consider) between obtained simulation times and economics of hardware.

Our perspectives lie in: a) the hybridization of the solver, to combine GPU approach with the cluster approach, b) the consideration of larger use cases with the same techniques, and in c) the end-to-end parallelization of the reservoir simulator.

ACKNOWLEDGMENTS

The authors thank Patrick Egermann for fruitful comments and suggestions.

REFERENCES

1 Wesseling P., Sonneveld P. (1980) Numerical experiments with a multiple grid and a preconditioned Lanczos type method, in Approximation Methods for Navier-Stokes Problems, in Rautmann R. (ed.), *Lecture Notes in Mathematics*, Springer, Berlin, Heidelberg, New-York, **771**, 543-562.

2 Saad Y. (2003) *Iterative methods for Sparse Linear systems*, SIAM, Philadelphia.

3 Sonneveld P., van Gijzen M. (2007) IDR(*s*), a family of simple and fast algorithms for solving large nonsymmetric linear systems, *Report 07-07*, Delft University of Technology, available at: http://ta.twi.tudelft.nl/nw/users/gijzen/idrs_report.pdf

4 van Gijzen M., Sonneveld P. (2011) Algorithm 913: An elegant IDR(*s*) Variant that efficiently exploits biorthogonality properties, *ACM Trans. Math. Soft.* **38**, 5, available at: http://ta.twi.tudelft.nl/nw/users/gijzen/idrs_toms.pdf

5 Collignon T.P., van Gijzen M. (2011) Minimizing synchronizations in IDR(*s*), *Numerical Linear Algebra with Applications* **18**, 805-825, available at: http://ta.twi.tudelft.nl/nw/users/gijzen/nlaa_idrs.pdf

6 Spinlocks, POSIX standard of the OpenGroup, available at http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_spin_lock.html

7 Memory barriers in the Gnu Compiler collection, available at https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html

8 Naumov M., Arsaev M., Castonguay P., Cohen J., Demouth J., Eaton J., Layton S., Markovskiy N., Reguly I., Sakharnykh N., Sellappan V., Strzodka R. (2014) AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods, *SIAM J. Sci. Comput.* **37**, 5, S602-S626.

9 Li R., Saad Y. (2013) GPU-accelerated preconditioned iterative linear solvers, *J. Supercomput.* **63**, 2013, 443-466.

10 NVIDIA Cuda, programming environment for NVIDIA GPUs, available at http://www.nvidia.fr/object/cuda-parallel-computing-fr.html

11 NVIDIA CuBlas, library for Blas routines on GPU, available at http://docs.nvidia.com/cuda/cublas

12 NVIDIA CuSparse, Library for Sparse linear algebra on GPU, available at http://docs.nvidia.com/cuda/cusparse/

13 OpenSHMEM standard, available at http://www.openshmem.org

14 Open-MPI, Open Message Passing Interface, available at: http://www.open-mpi.org

15 METIS, Graph and meshes decomposition library, available at http://glaros.dtc.umn.edu/gkhome/views/metis

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

Cite this article as: V. Seignole, J.-F. Thebault and R. Nabil (2016). Analysis of IDR(*s*) Family of Solvers for Reservoir Simulations on Different Parallel Architectures, *Oil Gas Sci. Technol* **71**, 63.