Parallel GMRES implementation for solving sparse linear systems on GPU clusters

Jacques M. Bahi, Raphaël Couturier, Lilia Ziane Khodja
University of Franche-Comte, LIFC laboratory, Rue Engel-Gros, BP 527, 90016 Belfort Cedex, France
{**jacques.bahi, raphael.couturier, lilia.ziane_khoja**}**@univ-fcomte.fr**

Keywords: sparse linear systems, GMRES, CUDA, GPU cluster

Abstract

In this paper, we propose an efficient parallel implementation of the GMRES method for GPU clusters. This implementation requires us to parallelize the GMRES algorithm between CPUs of the cluster. Hence, all parallel and intensive computations on local data are performed on GPUs and reduction operations to compute global results are carried out by CPUs. The performances of our parallel GMRES solver are evaluated on test matrices of sizes exceeding 10^7 rows. They show that solving large and sparse linear systems on a GPU cluster is faster than that performed on its CPU counterpart. It is noticed that a cluster of 12 GPUs is about 8 times faster than a cluster of 12 CPUs and about 5 times faster than a cluster of 24 CPUs.

1. INTRODUCTION

In numerous scientific and industrial applications, solving large sparse linear systems is often the most costly step, in both CPU time and memory space, in the computing process. In fact, solving such linear systems tends to be slowed down due to the large number of their unknowns and the irregularity of memory accesses to the few number of non-zero values of their matrices. Thus, any reduction in solving time of these systems will result in a significant saving in the total numerical computation time.

For the past few years, modern graphics processing units (GPUs) have become attractive tools to provide the high computing power and the large memory bandwidth required for sparse linear system solutions [6], [12]. Although initially they were designed to carry out intensive computations of graphic applications, GPUs have rapidly evolved to become high performance accelerators for data-parallel tasks and intensive arithmetic computations. To exploit the computational power of this architecture, Nvidia has released the CUDA platform (Compute Unified Device Architecture) [14] which provides a high level GPGPU-based programming language (General-Purpose computing on GPUs) allowing us to program GPUs not only for graphic application purposes but also for general purpose computations of non-graphic applications.

Nowadays, the relevant parallel architectures exploiting the high performances of GPUs are clusters equipped with GPUs. They have become very attractive for high performance computing, given their low cost compared to their computation power and their abilities to compute faster and to consume less energy than their pure CPU counterparts [2]. GPU clusters have already been used to accelerate numerical computations of sparse linear solvers, as [3], [7] where authors have parallelized the Conjugate Gradient solver (CG solver) on several GPU platforms and under different parameters affecting its performance. For example, Cevahir and al. [7] have presented the parallel implementation of a CG solver using hypergraph partitioning on CPU and GPU clusters. They show that double precision CG solution with 32 GPUs on 16 nodes TSUBAME is 17.4 times faster than the CPU implementation of the same amount of nodes and CPU cores. However, we have noticed in the literature that most parallel implementations of sparse linear solvers were performed on small sparse linear systems not exceeding $6 \cdot 10^6$ unknowns. Therefore our purpose in this paper is to solve more effectively large sparse linear systems whose sizes exceed 10^7 unknowns and to study the effect of these matrix sizes on the performances of our proposed sparse linear solver.

The target sparse linear solver that we explore on a GPU cluster is a GMRES solver (Generalized Minimal RESidual), since it is a generic and efficient method to solve linear systems. It gives better performances in most cases regardless of properties of associated matrices to linear systems (sparsity, symmetry, number of unknowns, type of values, ...). Moreover, it is an iterative method which is well-suited for linear systems of a very large order. In fact, an iterative method computes a sequence of approximate solutions converging to the exact solution. In contrast, a direct method determines the exact solution after a finite number of operations which leads to an expensive consumptions in computation time and memory space, and which therefore is not suited for large linear systems. GMRES parallelization has already been attempted on various parallel platforms using different approaches. Indeed, parallel GMRES algorithms are implemented on distributed memory architectures for large linear systems. For instance, Dias and al. [10] presented a parallel implementation of the restarted GMRES under the PVM message passing system on a MEiKO SPARC-based Computing Surface, and Couturier and al. [9] presented a parallel GMRES implementation on Grid'5000 using Java language and the MPJ library for com-

munication. Furthermore in recent years, several works like [8], [16] have proposed efficient GMRES solvers for GPUs. However, we can see that most parallel GMRES implementations proposed in the literature are either performed on pure CPU platforms or on architectures equipped with a single GPU. Hence in this paper, we propose a parallel GMRES solver for a mixed platform incorporating several GPUs and CPUs.

This paper is organized as follows. In section 2 is given a general description of the GMRES algorithm followed by that of GPU architectures in section 3. In section 4 are presented key points of our parallel implementation of GMRES solver on GPU cluster. In section 5, we show various relative gains of GMRES implementation on a GPU cluster compared to that implemented on a CPU cluster with different sizes of sparse matrices. Section 6 concludes this paper.

2. PRINCIPLE OF THE GMRES METHOD

GMRES is the generalization of the MINRES method (MInimal RESidual) to be applied to non-symmetric and non-Hermitian linear systems, in particular, and to all types of systems in general. It is an iterative method developed by Saad and Schultz [15] and it gives good results in most cases. Therefore, it is considered as a well-suited solution for solving linear systems with large and sparse matrices. Let the following system of equations:

$$Ax = b \tag{1}$$

where $x, b \in \mathcal{R}^n$, $A \in \mathcal{R}^{n \times n}$ is a nonsingular and sparse square matrix and n is the size of the system. The main idea of this method is to compute the solution x of the linear system (1) in the Krylov subspace $K(m; A; r_0)$ generated by $x_0 + K_m$, such that $r_0 = b - Ax_0$ is the residual associated with the initial guess x_0 of the system (1).

GMRES uses the Arnoldi process [4] to construct an orthonormal basis V_m for $K(m; A; r_0)$. This process allows it to produce a sequence of approximate solutions x_m converging to the exact solution x' in at most n iterations. Indeed, it minimizes the residual norm of the approximate solution over the Krylov subspace at each iteration until reaching the desired residual tolerance ε. In the case of GMRES with restarts, the Arnoldi process is restricted at $m \ll n$ iterations and restarted with the last iterate x_m as an initial guess to compute the new solution, in order to accelerate the method convergence and to avoid the storage of a large orthonormal basis V_m. Sometimes, GMRES does not converge or takes too many iterations to satisfy the convergence criterion. Therefore, in most cases, GMRES must contain a preconditioning step to improve its convergence. The preconditioning technique replaces the system (1) with the modified systems:

$$M^{-1}Ax = M^{-1}b \tag{2}$$

or

$$AM^{-1}\hat{x} = b, x = M^{-1}\hat{x} \tag{3}$$

where M is the preconditioning matrix. The main key points of the left-preconditioned GMRES method with restarts are represented in Algorithm 1.

Algorithm 1 Left-preconditioned GMRES with restarts

1: Set ε the tolerance for the residual norm r, $convergence = false$ and choose x_0
2: **while** *convergence* **do**
3: $r_0 = M^{-1}(b - Ax_0)$
4: $\beta = \|r_0\|_2$
5: $v_1 = r_0/\beta$
6: **for** $j = 1$ to m **do**
7: $w_j = M^{-1}Av_j$
8: **for** $i = 1$ to j **do**
9: $h_{i,j} = (w_j, v_i)$
10: $w_j = w_j - h_{i,j}v_i$
11: **end for**
12: $h_{j+1,j} = \|w_j\|_2$
13: $v_{j+1} = w_j/h_{j+1,j}$
14: **end for**
15: Set $V_m = [v_1, \cdots, v_m]$ and $\overline{H}_m = (h_{i,j})$ an upper Hessenberg matrix of order $(m+1) \times m$
16: Solve a least-square problem of size m: $min_{y \in \mathcal{R}^m} \|\beta e_1 - \overline{H}_m y\|_2$
17: $x_m = x_0 + V_m y_m$
18: **if** $\|M^{-1}(b - Ax_m)\|_2 < \varepsilon$ **then**
19: $convergence = true$
20: **end if**
21: $x_0 = x_m$
22: **end while**

3. OUR CUDA GPU CLUSTER

Our experimental platform is an Infiniband cluster with six CPUs Intel Xeon E5530 Nehalem. Each CPU is a Quad-Core processor running at 2.4GHz. It provides a RAM memory of 12GB and a memory bandwidth of 25.6GB/s, and it has two Nvidia Tesla C1060 GPUs.

Hardware architecture of Tesla GPU is based on Nvidia CUDA model which is also the name of the software for programming this architecture. Tesla C1060 GPU contains in total 240 cores organized in 30 *streaming multiprocessors* (MPs), each with 8 *streaming processors* (SPs) running at 1.3GHz. It provides 4GB of *device memory* with a memory bandwidth of 102GB/s, accessible by all its cores and also by the CPU through the PCI-Express 16x Gen 2.0 interface. Besides this global memory, each MP of this GPU has its own fast *shared memory* of 16KB shared among all its SPs.

In CUDA programming environment, the GPU is viewed as a co-processor to the CPU. All data-parallel and compute-

intensive portions of an application running on the CPU are off-loaded onto the GPU. In fact, a CUDA program is a C program running on the CPU with a minimal set of extensions to the C programming language to define the C functions to be performed on the GPU, called *kernels*. Then on the GPU, the same kernel is executed by a high number of parallel threads in SIMD fashion (Single Instruction Multiple Data). Threads are grouped together as a grid of thread blocks, such that each MP executes one or more thread blocks and each of its SP runs one or more threads within a block. At any given clock cycle, threads execute the same instruction of a kernel, but each of them operates on different data. Moreover, threads within the same block can coordinate their execution through a synchronization point. In contrast, within a grid of blocks, there is no synchronization at all between thread blocks. Kernels work only with data in the GPU device memory and their final results must be communicated to the CPU. Hence, the data must be transferred *in* and *out* of the GPU. However, transferring data *from* or *to* the GPU memory takes a significant amount of the global computation time. The speed of copy memory between the CPU and the GPU gives about 5GB/s which is much slower than the memory copy speed of the GPU memory, 102GB/s. Accordingly, it is necessary to limit the transfer of data between the GPU and CPU.

4. GMRES PARALLELIZATION ON A GPU CLUSTER

In this paper, we choose to parallelize the GMRES algorithm of [8] between CPUs of the cluster. It is an algorithm adapted to GPUs with restarts and a very basic left-preconditioning. Nevertheless, its parallelization requires to study three important points which are data partitioning, data dependencies and communications between the various components of the cluster.

Before starting computations, our parallel GMRES solver splits input data of the linear system between MPI processes on the cluster. From Algorithm 1, the input data are the sparse matrix A, the right-hand side b, the initial guess x_0 and the preconditioning matrix M. Let p denote the number of MPI processes on the cluster and n the size of the linear system to be solved. The algorithm performs a simple data partitioning by creating p portions, of at most $\lceil n/p \rceil$ rows per MPI process, for each element mentioned above. Consequently, each MPI process k will have its own sparse matrix $A_k(n/p \times n)$, preconditioning matrix $M_k^{-1}(n/p \times n/p)$ and right-hand side b_k of size n/p. To avoid the storage of the entire solution vector x by all MPI processes, this vector is also split into p sub-vectors x^k. As shown in Figure 1, the solution vector x^k managed by the MPI process k is composed of two parts: the local computed sub-vector x_k of size n/p and the shared sub-vectors x_{Left} and x_{Right} computed by neighbor processes. Let

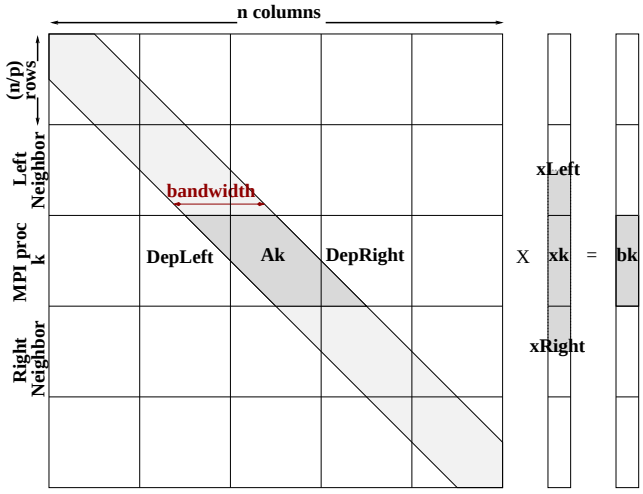


Figure 1. Data partitioning of the sparse matrix A, the solution vector x and the right-hand side b into p portions required locally by a CPU

$bw > 0$ be the bandwidth of A, then the size of shared data is $|x_{Left}| + |x_{Right}| \leq bw$. The intermediate computed vectors in Algorithm 1: r_0, $v_i\,(i = 1, \cdots, m+1)$, $w_j\,(j = 1, \cdots, m)$ and x_m are also split among all processes, such that vectors v are split as the solution vector x in v_k, v_{Left} and v_{Right}. After the data splitting, our algorithm must compute the data dependency pattern between MPI processes on the cluster. This operation consists in computing for each process all its dependencies with its neighbors, which can fulfill its sub-vectors x_{Left}, x_{Right}, v_{Left} and v_{Right}. For that, a MPI process computes what data it shares with each of its neighbors and then exchanges this information with them.

Since a GPU works only on data of its memory, all local input data, A_k, x_0^k, b_k and M_k^{-1}, must be transferred from CPU memories to the corresponding GPU memories. Sparse sub-matrices A_k are stored on GPU memories in HYB format (Hybrid) which stores only non-zero values in these matrices. This storage format is the combination of two sparse storage formats: Ellpack format (ELL) and Coordinate format (COO). It stores a typical number of non-zero values per row in ELL format and remaining entries of exceptional rows in COO format. It combines the *efficiency* of ELL due to the regularity of its memory accessing and the *flexibility* of COO which is insensitive to the matrix structure. Therefore, the HYB format gives on average the best performance in sparse matrix operations on GPUs [5].

Afterward, the same GMRES algorithm (Algorithm 1) is run by all processes but on different sub-systems $A_k x^k = b_k$, $k = 1, \ldots, p$. Each MPI process acts as a controller of the main loop of GMRES iterations and, all data-parallel operations inside this loop will be performed by its GPU. In Algorithm 1, we can see that typical data-parallel operations of

GMRES are those of sparse matrix-vector products (SpMV) (lines 3 and 7), scalar-vector products (lines 5 and 13), dot products (line 9), euclidean norms (lines 4, 12 and 18) and AXPY operations (line 10). They are implemented as kernels using CUDA programming language. We used the efficient HYB kernel of Nvidia for the SpMV [5] and fastest kernels of basic linear algebra subprograms of CUDA (cublas) [13] for vector operations (cublasDdot, cublasDnrm2, cublasDaxpy). For the rest of the parallel operations, we wrote their function codes in CUDA as kernels. We developed a kernel for the scalar-vector products (lines 5 and 13), a kernel to solve the least-square problem (line 16) and a kernel for the solution vector update (line 17).

Therefore, we have written the overall of our parallel GMRES code in C, using the extensions of CUDA for the GPU programming. In this code, all kernels to be performed on the GPU are defined as seperate functions, from those of the CPU, by assigning it a function type qualifier *__device__*. Then, each kernel inside the main loop of GMRES will be called by the MPI process and executed by the GPU of this last. Once on the GPU, each kernel will be executed in parallel by multiple thread blocks, whose number is provided by the host (MPI process) when calling the kernel. In our GMRES implementation, we set the thread block size to the maximum block size supported by the GPU architecture, 512 threads. Then, each MPI process computes the number of thread blocks, *Blocks*, to be involved in a kernel execution on its GPU as follows:

$$Blocks = \frac{(Num\_Rows + Threads - 1)}{Threads} \quad (4)$$

where *Num_Rows* is the number of local matrix rows and *Threads* is the thread block size. In this way, we will have one thread per matrix row and/or vector element.

Besides these local computations, communications and synchronizations between GPUs must be performed to assure the solving of the complete sparse linear system $Ax = b$. In every GMRES iteration of our parallel algorithm, there are two types of synchronizations: before SpMV and after vector operations. Before computing lines 3 and 7, it is mandatory to construct complete vectors x_0 and v_j, respectively, required to perform the full SpMV at each GPU. For that, each MPI process holds a global array tmp of size $\left(|x_{Left}| + |x_k| + |x_{Right}|\right)$, where its GPU *writes to* or *reads from* for communicating vector entries. Then, each GPU copies its x_k (resp. v_{jk}) subvector entries, required by other GPUs, to corresponding indices of tmp vector of its host. Once these copies are held, the MPI process performs required exchanges of tmp vector entries with its neighbors using *MPI_Alltoallv()* communication function. After that, each GPU reads its x_{Left} and x_{Right} (resp. v_{jLeft} and v_{jRight}) entries from corresponding indices of tmp vector of its MPI process host. After each of the following lines 4, 9, 12 and 18, MPI processes perform a reduction operation on local scalars computed by GPUs at this line using *MPI_Allreduce()* function, in order to compute the global scalar of the complete linear system.

Scalar-vector products (lines 5 and 13) and the AXPY operation (line 10) are performed locally by each GPU without any synchronization. They involve only local data of vectors and the global scalar computed before with a reduction operation. Line 16 solves a least-square problem of size m, where m is typically small. It is an inexpensive operation which can be computed sequentially on local data by one thread in each GPU. Line 17 computes, at each GMRES iteration, the approximate solution x_m for the linear system. It performs simple updates to the local part of the solution vector computed at the previous iteration. It is executed as a kernel in parallel by all GPUs because each of them is in charge of its own parts of x_m, V_m and the previous solution vector x_0.

We can see that there are two types of communications in a GPU cluster. First, we have those occurring between MPI processes on CPUs and GPUs. Indeed, MPI process requests its GPU to give it the local computed scalars and, a GPU requests its host to give it the global tmp vector required for a SpMV operation. These communications are implemented using copy functions of cublas: *cublasGetVector()* for GPU→CPU communications and *cublasSetVector()* for GPU←CPU communications. The second type of communications occurs between MPI processes using MPI functions to perform reduction operations or to exchange shared data of the global array tmp.

The whole code of our parallel GMRES implementation for a CPU cluster is written in C programming language, using MPI functions to perform communications between CPU cores. We have developed our own CPU kernels for the parallel operations of the GMRES method: SpMV product, vector operations, solving the least-square and solution vector update. With these developed kernels, we did not require to use any CPU libreries sine the authors of [9] showed that there is not a large difference between the use of our functions or those of PETSc library.

5. EXPERIMENTAL RESULTS

5.1. Test sparse matrices

Our aim is to test our parallel GMRES algorithm on large sparse linear systems whose sizes exceed 10^7 unknowns. Moreover, target sparse matrices are those with *band* structure, since they arise in many numerical computations. We chose to perform our tests on the real-world sparse matrices of the university of Florida collection (UF collection) [11]. However, matrices in this collection are very small compared to our desired matrix sizes. Therefore, we developed in C a generator of sparse matrices which takes one real matrix of the UF collection as an initial matrix to build large matrices.

This generator is executed in parallel by all MPI processes

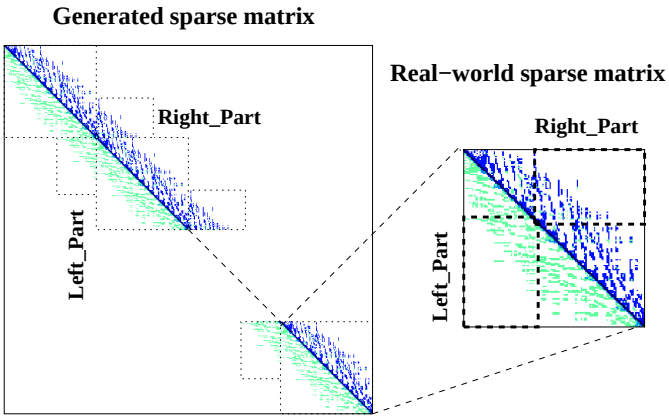


Figure 2. A generated large and sparse matrix from a real matrix of UF collection

on the cluster before any execution of our GMRES algorithm. It allows each MPI process k to build its sub-matrix A_k by performing several copies of the same real matrix B of the UF collection and, such that all these copies in the cluster build the large matrix of the sparse linear system. Let us denote by N the desired size of the sparse linear system to be solved and, thus, the size of the associated sparse matrix. At first, all MPI processes load the same matrix B from its storage file. Each of them computes the number of copies Nb_k to perform on B, such that:

$$Nb_k = \frac{\lceil N/size_B \rceil}{p} \tag{5}$$

where $size_B$ is the size of the matrix B and p the number of MPI processes on the cluster. After that, each process k performs Nb_k copies of the matrix B on which it constructs its sub-matrix A_k. In order to build matrices with band structure, each process places its copies on its part of the main diagonal of the global generated matrix, as shown in Figure 2. Besides these full copies, other sub-copies *Left_Part* and *Right_Part* of the matrix B are performed to fulfill the empty spaces between two consecutive copies on the main diagonal of the generated matrix. These sub-copies *Left_Part* and *Right_Part* are performed from the widest row of the matrix B, which defines the bandwidth of this matrix.

Table 1 shows 12 square matrices that are positive definite with real values entries, chosen from the UF collection to build our large test matrices. The columns of Table 1 give the main characteristics of each matrix: the number of rows, the number of non-zero values and the bandwidth of the generated matrices from these matrices. We can also see in Figure 3 the structures of these matrices.

5.2. Test platform

In our experiments, we have evaluated the speed of the parallel GMRES solver implemented on a GPU cluster against

Table 1. Description of the chosen matrices from UF collection

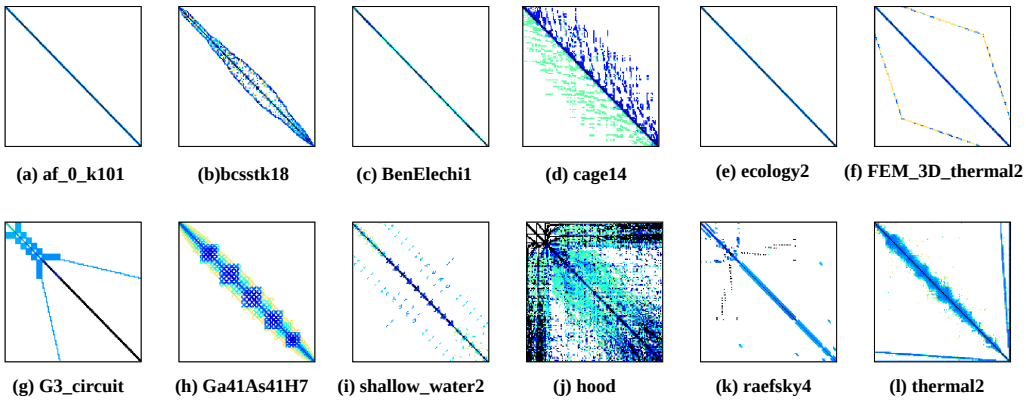
Matrix	Nb. rows	Nb. nonzeros	Bandwidth
af_0_k101	503,625	9,027,150	1,716
bcsstk18	11,948	80,519	1,650
BenElechi1	245,874	6,698,185	1,171
cage14	1,505,785	27,130,349	1,492,443
ecology2	999,999	2,997,995	2,002
FEM_3D_thermal2	147,900	3,489,300	206,126
G3_circuit	1,585,478	4,623,152	1,584,984
Ga41As41H72	268,096	9,378,286	64,512
shallow_water2	81,920	204,800	61,441
hood	220,542	5,494,489	220,543
raefsky4	19,779	674,195	17,820
thermal2	1,228,045	4,904,179	1,228,046

that implemented on a CPU cluster. As mentioned in section 3, the experiments are performed on an Infiniband cluster of six Xeon E5530 nodes. However in our GPU cluster of tests, we exploit only two CPU cores of each node, such that each CPU core manages one Tesla C1060 GPU since this last is passive device. Therefore, we have compared the performances of the GMRES solver implemented on a cluster of 12 GPUs with those obtained on clusters of 12 CPU cores and those obtained on cluster of 24 CPU cores. Linux cluster version 2.6.18 OS is installed on nodes. C programming language is used for coding the GMRES algorithm on both GPU cluster and CPU cluster. CUDA version 3.1.1 [14] is used for programming GPUs, using CUBLAS 3.1 to deal with vector operations and CUSP library [1] to perform a HYB SpMV product in GPUs, and finally MPI functions are used to carry out communications between CPU cores.

We tested our solver on sparse matrices generated from the sparse matrices of Table 1. The sizes of generated matrices vary from $2 \cdot 10^7$ to $6 \cdot 10^7$ rows and manageable numbers of non-zeros values vary from 59,925,452 to 1,634,561,260. The number of non-zeros values of these generated matrices is reported in the second column of Tables 2, 3, and 4. Our tests are made in double precision data. All results obtained from the performance evaluation of our GMRES solver are for a residual tolerance threshold $\varepsilon = 10^{-10}$, a restart limit $m = 16$ and, a right-hand side b filled with 1 and an initial guess x_0 filled with 12. For the sake of simplicity, we chose a preconditioning matrix M easy to compute and to inverse, without seeking for the best efficiency for some particular cases. For that, we took M as the A diagonal which provides a relatively good preconditioning in most cases.

5.3. Results

Comparison results between GPU cluster and CPU clusters are given in Tables 2, 3, and 4 for sparse linear systems of sizes $2 \cdot 10^7$, $4 \cdot 10^7$, and $6 \cdot 10^7$ unknowns, respectively. We



(a) af_0_k101 (b)bcsstk18 (c) BenElechi1 (d) cage14 (e) ecology2 (f) FEM_3D_thermal2

(g) G3_circuit (h) Ga41As41H7 (i) shallow_water2 (j) hood (k) raefsky4 (l) thermal2

Figure 3. Structures of the matrices chosen from UF collection

Table 2. Comparison between GPU cluster and CPU clusters for size $2 \cdot 10^7$ unknowns

Matrix	#Nonzeros	T_{GPU}	T_{12CPUs}	$Ratio_{12CPUs}$	T_{24CPUs}	$Ratio_{24CPUs}$	#Iter	Prec	Error
af_0_k101	$361 \cdot 10^6$	1.99s	15.45s	7.74	10.52s	5.29	62	1.52e-14	6.60e-17
bcsstk18	$137 \cdot 10^6$	0.61s	4.98s	7.88	3.29s	5.39	30	4.90e-10	3.09e-12
BenElechi1	$543 \cdot 10^6$	2.15s	17.36s	7.90	12.68s	5.90	70	1.50e-11	6.94e-17
cage14	$395 \cdot 10^6$	1.75s	11.80s	6.80	10.75s	6.14	33	1.34e-08	2.67e-10
ecology2	$60 \cdot 10^6$	0.71s	6.19s	8.59	4.29s	6.04	36	5.42e-10	6.76e-13
FEM_3D_thermal2	$516 \cdot 10^6$	2.15s	15.15s	7.09	10.38s	4.83	61	3.56e-07	2.41e-09
G3_circuit	$71 \cdot 10^6$	1.69s	11.17s	6.59	10.12s	5.99	45	6.77e-10	9.30e-14
Ga41As41H72	$706 \cdot 10^6$	7.20s	46.51s	6.48	32.27s	4.48	149	7.32e-11	7.66e-15
shallow_water2	$60 \cdot 10^6$	0.63s	5.49s	9.06	3.78s	5.99	33	1.81e-13	1.12e-15
hood	$520 \cdot 10^6$	2.30s	14.76s	6.30	10.51s	4.57	56	8.11e-10	2.66e-16
raefsky4	$693 \cdot 10^6$	3.37s	23.56s	6.93	16.22s	4.81	87	1.32e-12	1.47e-17
thermal2	$97 \cdot 10^6$	1.36s	6.74s	4.94	4.46s	3.28	27	7.64e-10	1.40e-12

Table 3. Comparison between GPU cluster and CPU clusters for size $4 \cdot 10^7$ unknowns

Matrix	#Nonzeros	T_{GPU}	T_{12CPUs}	$Ratio_{12CPUs}$	T_{24CPUs}	$Ratio_{24CPUs}$	#Iter	Prec	Error
af_0_k101	$713 \cdot 10^6$	3.52s	27.15s	7.78	18.40s	5.23	62	1.40e-14	4.50e-15
bcsstk18	$273 \cdot 10^6$	1.24s	9.90s	8.14	6.53s	5.26	30	4.90e-10	1.76e-11
BenElechi1	10^9	4.31s	34.74s	8.14	24.58s	5.70	70	1.50e-11	4.16e-17
cage14	$823 \cdot 10^6$	3.08s	19.75s	6.49	17.99s	5.84	32	4.95e-08	2.12e-09
ecology2	$120 \cdot 10^6$	1.35s	12.43s	9.14	8.54s	6.32	36	5.34e-10	6.54e-13
FEM_3D_thermal2	10^9	4.03s	28.99s	7.30	20.48s	5.08	61	3.56e-07	1.81e-09
G3_circuit	$138 \cdot 10^6$	2.40s	17.23s	7.34	15.62s	6.51	45	6.75e-10	6.97e-13
Ga41As41H72	$1.5 \cdot 10^9$	13.19s	86.02s	6.56	63.76s	4.83	149	5.68e-11	1.37e-14
shallow_water2	$120 \cdot 10^6$	1.11s	10.71s	9.60	7.21s	6.50	33	1.81e-13	7.78e-16
hood	10^9	4.16s	27.83s	6.72	19.49s	4.68	56	8.11e-10	1.78e-16
raefsky4	$1.3 \cdot 10^9$	6.72s	47.11s	6.91	31.88s	4.74	87	1.29e-12	1.65e-17
thermal2	$201 \cdot 10^6$	1.67s	9.46s	5.58	6.26s	3.75	27	1.90e-09	2.64e-10

report the performances of our parallel GMRES solver for different sparse matrices generated from matrices mentioned in the first column of each table. The results presented are obtained from the mean value over 10 executions of the same algorithm and for the same input data. The dashes (-) in some cells of Table 4 indicate that there were GPU memory overflows for such data sizes and, thus, we could not solve the linear system for these sparse matrices on the GPU cluster.

In the third, fourth and sixth columns, we report execution times in seconds of our solver on: a cluster of 12 GPUs, a cluster of 12 CPU cores and a cluster of 24 CPU cores, respectively. The number of iterations required to reach the residual tolerance threshold $\varepsilon = 10^{-10}$ is reported in the eighth column of these tables. For the same sparse linear system, it is identical for both implementations (GPU and CPU clustes) of our parallel solver. The execution time includes

Table 4. Comparison between GPU cluster and CPU clusters for size $6 \cdot 10^7$ unknowns

Matrix	#Nonzeros	T_{GPU}	T_{12CPUs}	$Ratio_{12CPUs}$	T_{24CPUs}	$Ratio_{24CPUs}$	#Iter	Prec	Error
af_0_k101	10^9	4.84s	38.69s	7.87	26.13s	5.40	62	1.33e-14	2.54e-15
bcsstk18	$410 \cdot 10^6$	1.78s	14.89s	8.23	9.78s	5.49	30	4.90e-10	4.30e-13
BenElechi1	$1.6 \cdot 10^9$	6.48s	52.14s	8.14	37.08s	5.72	70	1.50e-11	6.94e-17
cage14	$1.2 \cdot 10^9$	3.77s	26.26s	6.82	18.14s	4.81	32	5.70e-08	4.35e-10
ecology2	$180 \cdot 10^6$	1.74s	15.48s	8.86	10.66s	6.13	36	5.32e-10	1.67e-13
FEM_3D_thermal2	$1.5 \cdot 10^9$	5.75s	42.92s	7.55	30.06s	5.23	61	3.56e-07	2.04e-09
G3_circuit	$211 \cdot 10^6$	3.36s	25.97s	7.72	17.88s	5.32	45	6.74e-10	1.46e-13
Ga41As41H72	$2.1 \cdot 10^9$	-	-	-	-	-	-	-	-
shallow_water2	$180 \cdot 10^6$	1.65s	15.93s	9.75	10.53s	6.38	33	1.81e-13	2.53e-15
hood	$1.5 \cdot 10^9$	-	-	-	-	-	-	-	-
raefsky4	$2.1 \cdot 10^9$	-	-	-	-	-	-	-	-
thermal2	$299 \cdot 10^6$	2.13s	14.46s	6.81	11.27s	5.29	27	6.77e-10	1.60e-10

only the solving time of the linear system without the matrix generation time, the data partitioning time and the construction time of the data shared scheme. Besides the computation time T_{comput} of arithmetic operations, the execution times T_{GPU} on GPU cluster and T_{CPU} on CPU cluster include the time of CPU↔CPU communications $T_{CPU \leftrightarrow CPU}$ and the time of GPU↔CPU communications $T_{GPU \leftrightarrow CPU}$ on the GPU cluster, as shown in the two following formula (6) and (7):

$$T_{GPU} = T_{comput} + T_{GPU \leftrightarrow CPU} + T_{CPU \leftrightarrow CPU} \quad (6)$$

$$T_{CPU} = T_{comput} + T_{CPU \leftrightarrow CPU} \quad (7)$$

In order to validate our results, we have computed two parameters allowing us to verify the solution accuracy of the solved linear system; which are the solution precision computed on the GPU cluster *prec* and the error computation between solutions of the two implementations *error*. They are reported in the ninth and tenth columns of the tables of results. The solution precision *prec* is the maximum value among the components of the residue vector as shown in the equation (8), where X^{GPU} is the solution vector computed by the GPU cluster. The parameter *error* allows us to ensure that both versions of the parallel solver implemented on the GPU cluster and the CPU cluster give more or less the same solution. It is the difference between the two solution vectors, X^{CPU} and X^{GPU}, of the CPU cluster and the GPU cluster, see the equation (9).

$$prec = max\left(M^{-1} \cdot \left(b - A \cdot X^{GPU}\right)\right) \quad (8)$$

$$error = max\left|X^{CPU} - X^{GPU}\right| \quad (9)$$

The relative gains $\frac{T_{CPU}}{T_{GPU}}$ in the execution times of the GPU cluster for different matrix sizes are shown in the fifth and seventh columns compared to the cluster of 12 CPU cores and the cluster of 24 CPU cores, respectively. In all conducted experiments, we can see that the GPU cluster version is faster than that of the CPU clusters. With the GPU cluster, we gained in the execution time 5 to 10 times more than with 12 CPU cores and 4 to 6.5 times more than with 24 CPU cores. Moreover, we can notice that relative gains of some types of matrices increase with the increasing of matrix sizes and the decreasing of the matrix bandwidths (please see the matrix *thermal2*). The GPUs work best on very large sizes of matrices due to their data-parallel nature and their use of the SIMD paradigm in a program execution, where each matrix row is attributed to a single thread. In contrast, the matrix bandwidth represents the size of the shared sub-vectors x_{Left} and x_{Right} to exchange with neighbors and, thus the number of sub-vector entries that a GPU must *write to* or *read from* the *tmp* vector of its host (see section 4). Therefore a large matrix bandwidth leads to the transfer of many sub-vector entries between a GPU and its host, whereas this type of data transfers is the slowest communication in a GPU cluster and, hence, it takes a significant amount of the global solving time.

From the tables of results, we can see that the precisions *prec* of our results are sufficient, varying from *3.56e-7* to *1.33e-14* and in general they are around *1e-10*. We can also see that the difference between the CPU cluster solution X^{CPU} and the GPU cluster solution X^{GPU} computed for the same sparse linear system is always quite low for the same precision of solving. This difference varies from *2.41e-9* to *1.47e-17* which allows us to conclude that, for a sparse linear system, our parallel GMRES solver on a GPU cluster almost computes the same solution as that computed on a CPU cluster.

6. CONCLUSION AND PERSPECTIVES

In this paper, we have presented a parallel GMRES algorithm for solving large and sparse linear systems on a GPU cluster. We have aimed to exploit the high power computing and the tremendous memory bandwidth of several GPUs, required for solving such linear systems. We have parallelized a GMRES algorithm with restarts and a basic left-preconditioning adapted to GPUs between CPU cores of the cluster.

The efficiency and performance of our parallel GMRES solver for sparse linear systems is demonstrated by numerical experimental results carried out on a GPU cluster. We have compared the performances of this solver on a cluster of 12 Tesla C1060 GPUs against those obtained on a cluster of 12 E5530 CPU cores and those obtained on a cluster of 24 E5530 CPU cores. The experiments have been performed on large sparse matrices with band structure and varying from $2 \cdot 10^7$ to $6 \cdot 10^7$ rows. The experimental results clearly show that the solving of large and sparse linear systems on the GPU cluster is faster than on the CPU cluster. The relative gains of the GPU cluster range from 5 up-to 10 compared to the cluster of 12 CPU cores and 4 up-to 6.5 compared to the cluster of 24 CPU cores, for the same precision. We have also noticed that the GPU cluster is more efficient for large sparse matrix sizes, due to the high data-parallel nature of the GPUs, provided that these large matrices do not exceed the limited memory capabilities of the GPUs.

In future work, we will evaluate our parallel GMRES solver on other structures of sparse matrices to see its performance behavior on a GPU cluster according to these different structures. We will also study the different methods of data partitioning according to the sparse matrix structures. This will allow us to minimize data sharing and dependencies between CPUs of the cluster and, thus, to deal with the slow data transfers between GPUs and CPUs. In addition, we will work on the parallel implementation of GMRES solver on a grid computing equipped with GPU cards for solving large and sparse linear systems whose sizes are in order of a billion unknowns. However this type of parallel platform requires us to take into account another parameter which is the asynchronism of its different distant clusters.

REFERENCES

[1] *CUSP library*. http://code.google.com/p/cusp-library/.

[2] ABBAS-TURKI, L. A., VIALLE, S., LAPEYRE, B., AND MERCIER, P. High dimensional pricing of exotic european contracts on a gpu cluster, and comparison to a cpu cluster. In *IPDPS'09* (2009), IEEE Computer Society, pp. 1–8.

[3] AMENT, M., KNITTEL, G., WEISKOPF, D., AND STRASSER, W. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. In *PDP'10* (2010), IEEE Computer Society, pp. 583–592.

[4] ARNOLDI, W. The principle of minimized iteration in the solution of the matrix eigenvalue problem. *Quart, Appl.Math. 9* (1951), 17–29.

[5] BELL, N., AND GARLAND, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC'09* (Portland, Oregon, USA, 2009), ACM, pp. 1–11.

[6] BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *SIGGRAPH'03* (San Diego, California, USA, 2003), ACM, pp. 917–924.

[7] CEVAHIR, A., NUKADA, A., AND MATSUOKA, S. High performance conjugate gradient solver on multi-gpu clusters using hypergraph partitioning. *Computer Science - Research and Development 25* (2010), 83–91.

[8] COUTURIER, R., AND DOMAS, S. Sparse systems solving on gpus with gmres. *The journal of Supercomputing* (2010). Accepted manuscript. To appear.

[9] COUTURIER, R., AND JÉZÉQUEL, F. Solving large sparse linear systems in a grid environment using java. In *PDSEC'10, 11-th IEEE Int., joint to IPDPS'10, ACM/IEEE Int.* (Atlanta, USA, 2010), IEEE Computer Society Press, pp. 1–7.

[10] DA CUNHA, R., AND HOPKINS, T. A parallel implementation of the restarted gmres iterative algorithm for nonsymmetric systems of linear equations. *Advances in Computational Mathematics 2* (1994), 261–277.

[11] DAVIS, T., AND HU, Y. The university of florida sparse matrix collection, 1997. NA Digest, http://www.cise.ufl.edu/research/sparse/matrices/.

[12] JOST, T., CONTASSOT-VIVIER, S., AND VIALLE, S. An efficient multi-algorithms sparse linear solver for GPUs. In *EuroGPU mini-symposium of the International Conference on Parallel Computing, ParCo2009* (Lyon, Sept. 2009), pp. 546–553.

[13] NVIDIA. *Cuda cublas library*, 2010. Version 3.1.

[14] NVIDIA. *NVIDIA CUDA C Programming Guide*, 2010. Version 3.1.1.

[15] SAAD, Y., AND SCHULTZ, M. H. Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput. 7*, 3 (1986), 856–869.

[16] WANG, M., KLIE, H., PARASHAR, M., AND SUDAN, H. Solving sparse linear systems on nvidia tesla gpus. In *ICCS'09* (Baton Rouge, Louisiana, USA, 2009), Springer-Verlag, pp. 864–873.