Parallel sparse linear solver with GMRES method using minimization techniques of communications for GPU clusters

LILIA ZIANE KHODJA   RAPHAËL COUTURIER*   ARNAUD GIERSCH   JACQUES M. BAHI

FEMTO-ST Institute, University of Franche-Comte
IUT Belfort-Montbéliard
19 Av. du Marchal Juin, BP 527, 90016 Belfort, France

{lilia.ziane_khoja, raphael.couturier, arnaud.giersch, jacques.bahi}@univ-fcomte.fr

Abstract

In this paper, we aim at exploiting the power computing of a GPU cluster for solving large sparse linear systems. We implement the parallel algorithm of the GMRES iterative method using the CUDA programming language and the MPI parallel environment. The experiments show that a GPU cluster is more efficient than a CPU cluster. In order to optimize the performances, we use a compressed storage format for the sparse vectors and the hypergraph partitioning. These solutions improve the spatial and temporal localization of the shared data between the computing nodes of the GPU cluster.

1   Introduction

Large sparse linear systems arise in most numerical scientific or industrial simulations. They model numerous complex problems in different areas of applications such as mathematics, engineering, biology or physics [6]. However, solving these systems of equations is often an expensive operation in terms of execution time and memory space consumption. Indeed, the linear systems arising in most applications are very large and have many zero coefficients, and this sparse nature leads to irregular accesses to load the nonzero coefficients from the memory.

Parallel computing has become a key issue for solving sparse linear systems of large sizes. This is due to the computing power and the storage capacity of the current parallel computers as well as the availability of different parallel programming languages and environments such as the MPI communication standard. Nowadays, graphics processing units (GPUs) are the most commonly used hardware accelerators in high performance computing. They are equipped with a massively parallel architecture allowing them to compute faster than CPUs. However, the parallel computers equipped with GPUs introduce new programming difficulties to adapt parallel algorithms to their architectures.

In this paper, we use the GMRES iterative method for solving large sparse linear systems on a cluster of GPUs. The parallel algorithm of this method is implemented using the CUDA programming language for the GPUs and the MPI parallel environment to distribute the computations between the different GPU nodes of the cluster. Particularly, we focus on improving the performances of the parallel sparse matrix-vector multiplication. Indeed, this operation is not only very time-consuming but it also requires communications between the GPU nodes. These communications are needed to build the global vector involved in the parallel sparse matrix-vector multiplication. It should be noted that a communication between two GPU nodes involves data transfers between the GPU and CPU memories in the same node and the MPI communications between the CPUs of the GPU nodes. For performance purposes, we propose to use a compressed storage format to reduce the size of the vectors to be exchanged between the GPU nodes and a hypergraph partitioning of the sparse matrix to reduce the total communication volume.

The present paper is organized as follows. In Section 2 some previous works about solving sparse linear systems on GPUs are presented. In Section 3 is given a general overview of the GPU architectures, followed by that the GMRES method in Section 4. In Section 5 the main key points of the parallel implementation of the GMRES method on a GPU cluster are described. Finally, in Section 6 is presented the performance improvements of the parallel GMRES algorithm on a GPU cluster.

---

*Contact author

2   Related work

Numerous works have shown the efficiency of GPUs for solving sparse linear systems compared to their CPUs counterpart. Different iterative methods are implemented on one GPU, for example Jacobi and Gauss-Seidel in [27], conjugate and biconjugate gradients in [23, 35, 10, 36] and GMRES in [34, 19, 29, 24]. In addition, some iterative methods are implemented on shared memory multi-GPUs machines as [13, 4, 22, 26]. A limited set of studies are devoted to the parallel implementation of the iterative methods on distributed memory GPU clusters as [25, 7, 30].

Traditionally, the parallel iterative algorithms do not often scale well on GPU clusters due to the significant cost of the communications between the computing nodes. Some authors have already studied how to reduce these communications. In [14], the authors used a hypergraph partitioning as a preprocessing to the parallel conjugate gradient algorithm in order to reduce the inter-GPU communications over a GPU cluster. The sequential hypergraph partitioning method provided by the PaToH tool [12] is used because of the small sizes of the sparse symmetric linear systems to be solved. In [8], a compression and decompression technique is proposed to reduce the communication overheads. This technique is performed on the shared vectors to be exchanged between the computing nodes. In [16], the authors studied the impact of asynchronism on parallel iterative algorithms on local GPU clusters. Asynchronous communication primitives suppress some synchronization barriers and allow overlap of communication and computation. In [15], a communication reduction method is used for implementing finite element methods (FEM) on GPU clusters. This method firstly uses the Reverse Cuthill-McKee reordering to reduce the total communication volume. In addition, the performances of the parallel FEM algorithm are improved by overlapping the communication with computation.

Our main contribution in this work is to show the difficulties of implementing the GMRES method to solve sparse linear systems on a cluster of GPUs. First, we show the main key points of the parallel GMRES algorithm on a GPU cluster. Then, we discuss the improvements of the algorithm which are mainly performed on the sparse matrix-vector multiplication when the matrix is distributed on several GPUs. In fact, on a cluster of GPUs the influence of the communications is greater than on clusters of CPUs due to the CPU/GPU communications between two GPUs that are not on the same machines. We propose to perform a hypergraph partitioning on the problem to be solved, then we reorder the matrix columns according to the partitioning scheme, and we use a compressed format to store the vectors in order to minimize the communication overheads between two GPUs.

3   GPU architectures

A GPU (Graphics processing unit) is a hardware accelerator for high performance computing. Its hardware architecture is composed of hundreds of cores organized in several blocks called *streaming multiprocessors*. It is also equipped with a memory hierarchy. It has a set of registers and a private read-write *local memory* per core, a fast *shared memory*, read-only *constant* and *texture* caches per multiprocessor and a read-write *global memory* shared by all its multiprocessors. The new architectures (Fermi, Kepler, etc) have also L1 and L2 caches to improve the accesses to the global memory.

NVIDIA has released the CUDA platform (Compute Unified Device Architecture) [31] which provides a high level GPGPU-based programming language (General-Purpose computing on GPUs), allowing to program GPUs for general purpose computations. In CUDA programming environment, all data-parallel and compute intensive portions of an application running on the CPU are off-loaded onto the GPU. Indeed, an application developed in CUDA is a program written in C language (or Fortran) with a minimal set of extensions to define the parallel functions to be executed by the GPU, called *kernels*. We define kernels, as separate functions from those of the CPU, by assigning them a function type qualifiers `__global__` or `__device__`.

At the GPU level, the same kernel is executed by a large number of parallel CUDA threads grouped together as a grid of thread blocks. Each multiprocessor of the GPU executes one or more thread blocks in SIMD fashion (Single Instruction, Multiple Data) and in turn each core of a GPU multiprocessor runs one or more threads within a block in SIMT fashion (Single Instruction, Multiple threads). In order to maximize the occupation of the GPU cores, the number of CUDA threads to be involved in a kernel execution is computed according to the size of the problem to be solved. In contrast, the block size is restricted by the limited memory resources of a core. On current GPUs, a thread block may contain up-to $1,024$ concurrent threads. At any given clock cycle, the threads execute the same instruction of a kernel, but each of them operates on different data. Moreover, threads within a block can cooperate by sharing data through the fast shared memory and coordinate their execution through synchronization points. In contrast, within a grid of thread blocks, there is no synchronization at all between blocks.

GPUs only work on data filled in their global memory and the final results of their kernel executions must be

communicated to their hosts (CPUs). Hence, the data must be transferred *in* and *out* of the GPU. However, the speed of memory copy between the CPU and the GPU is slower than the memory copy speed of GPUs. Accordingly, it is necessary to limit the transfer of data between the GPU and its host.

4 GMRES method

The generalized minimal residual method (GMRES) is an iterative method designed by Saad and Schultz in 1986 [33]. It is a generalization of the minimal residual method (MNRES) [32] to deal with asymmetric and non Hermitian problems and indefinite symmetric problems.

Let us consider the following sparse linear system of n equations:

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

where $A \in \mathbb{R}^{n \times n}$ is a sparse square and nonsingular matrix, $x \in \mathbb{R}^n$ is the solution vector and $b \in \mathbb{R}^n$ is the right-hand side vector. The main idea of the GMRES method is to find a sequence of solutions $\{x_k\}_{k \in \mathbb{N}}$ which minimizes at best the residual $r_k = b - Ax_k$. The solution x_k is computed in a Krylov sub-space $\mathcal{K}_k(A, v_1)$:

$$\mathcal{K}_k(A, v_1) \equiv \text{span}\{v_1, Av_1, A^2v_1, ..., A^{k-1}v_1\}, \quad v_1 = \frac{r_0}{\|r_0\|_2}, \tag{2}$$

such that the Petrov-Galerkin condition is satisfied:

$$r_k \perp A\mathcal{K}_k(A, v_1). \tag{3}$$

Algorithm 1 illustrates the main key points of the GMRES method with restarts. The linear system to be solved in this algorithm is left-preconditioned where M is the preconditioning matrix. The Arnoldi process [5] is used (from line 7 to line 17 of algorithm 1) to construct an orthonormal basis V_m and a Hessenberg matrix $\bar{H}_m$ of order $(m+1) \times m$ such that $m \ll n$. Then, the least-squares problem is solved (line 18) to find the vector $y \in \mathbb{R}^m$ which minimizes the residual. Finally, the solution x_m is computed in the Krylov sub-space spanned by V_m (line 19). In practice, the GMRES algorithm stops when the Euclidean norm of the residual is small enough and/or the maximum number of iterations is reached.

5 Parallel GMRES method on GPU clusters

5.1 Parallel implementation on a GPU cluster

The implementation of the GMRES algorithm on a GPU cluster is performed by using a parallel heterogeneous programming. We use the programming language CUDA for the GPUs and the parallel environment MPI for the distribution of the computations between the GPU computing nodes. In this work, a GPU computing node is composed of a GPU and a CPU core managed by a MPI process.

Let us consider a cluster composed of p GPU computing nodes. First, the sparse linear system (1) is split into p sub-linear systems, each is attributed to a GPU computing node. We partition row-by-row the sparse matrix A and both vectors x and b in p parts (see Figure 1). The data issued from the partitioning operation are off-loaded on the GPU global memories to be proceeded by the GPUs. Then, all the computing nodes of the GPU cluster execute the same GMRES iterative algorithm but on different data. Finally, the GPU computing nodes synchronize their computations by using MPI communication routines to solve the global sparse linear system. In what follows, the computing nodes sharing data are called the neighboring nodes.

In order to exploit the computing power of the GPUs, we have to execute maximum computations on GPUs to avoid the data transfers between the GPU and its host (CPU), and to maximize the GPU cores utilization to hide global memory access latency. The implementation of the GMRES algorithm is performed by executing the functions operating on vectors and matrices as kernels on GPUs. These operations are often easy to parallelize and more efficient on parallel architectures when they operate on large vectors. We use the fastest routines of the CUBLAS library (CUDA Basic Linear Algebra Subroutines) to implement the dot product (`cublasDdot()`), the Euclidean norm (`cublasDnrm2()`) and the AXPY operation (`cublasDaxpy()`). In addition, we have coded in CUDA a kernel for the scalar-vector product (lines 7 and 15 of Algorithm 1), a kernel for solving the least-squares problem (line 18) and a kernel for solution vector updates (line 19).

The solution of the least-squares problem in the GMRES algorithm is based on:

- a QR factorization of the Hessenberg matrix $\bar{H}$ by using plane rotations and,

---

Algorithm 1: Left-preconditioned GMRES algorithm with restarts

Input: A (matrix), b (vector), M (preconditioning matrix), x_0 (initial guess), ε (tolerance threshold), max (maximum number of iterations), m (number of iterations of the Arnoldi process)

Output: x (solution vector)

1 $r_0 \leftarrow M^{-1}(b - Ax_0)$;
2 $\beta \leftarrow \|r_0\|_2$;
3 $\alpha \leftarrow \|M^{-1}b\|_2$;
4 $convergence \leftarrow false$;
5 $k \leftarrow 1$;

6 **while** $(\neg convergence)$ **do**
7 $v_1 \leftarrow r_0/\beta$;
8   **for** $j = 1$ **to** m **do**
9 $w_j \leftarrow M^{-1}Av_j$;
10   **for** $i = 1$ **to** j **do**
11 $h_{i,j} \leftarrow (w_j, v_i)$;
12 $w_j \leftarrow w_j - h_{i,j} \times v_i$;
13   **end**
14 $h_{j+1,j} \leftarrow \|w_j\|_2$;
15 $v_{j+1} \leftarrow w_j/h_{j+1,j}$;
16   **end**

17   Put $V_m = \{v_j\}_{1 \leq j \leq m}$ and $\bar{H}_m = (h_{i,j})$ Hessenberg matrix of order $(m+1) \times m$;
18   Solve the least-squares problem of size m: $\min\limits_{y \in \mathbb{R}^m} \|\beta e_1 - \bar{H}_m y\|_2$;

19 $x_m \leftarrow x_0 + V_m y$;
20 $r_m \leftarrow M^{-1}(b - Ax_m)$;
21 $\beta \leftarrow \|r_m\|_2$;

22   **if** $(\frac{\beta}{\alpha} < \varepsilon)$ **or** $(k \geq max)$ **then**
23 $convergence \leftarrow true$;
24   **else**
25 $x_0 \leftarrow x_m$;
26 $r_0 \leftarrow r_m$;
27 $k \leftarrow k + 1$;
28   **end**
29 **end**

- backward-substitution method to compute the vector y minimizing the residual.

This operation is not easy to parallelize and it is not interesting to implement it on GPUs. However, the size m of the linear least-squares problem to solve in the GMRES method with restarts is very small. So, this problem is solved in sequential by one GPU thread.

The most important operation in the GMRES method is the sparse matrix-vector multiplication. It is quite expensive for large size matrices in terms of execution time and memory space. In addition, it performs irregular memory accesses to read the nonzero values of the sparse matrix, implying non coalescent accesses to the GPU global memory which slow down the performances of the GPUs. So we use the HYB kernel developed and optimized by NVIDIA [1] which gives on average the best performance in sparse matrix-vector multiplications on GPUs [9]. The HYB (Hybrid) storage format is the combination of two sparse storage formats: Ellpack format (ELL) and Coordinate format (COO). It stores a typical number of nonzero 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.

In the parallel GMRES algorithm, the GPU computing nodes must exchange between them their shared data in order to construct the global vector necessary to compute the parallel sparse matrix-vector multiplication (SpMV). In fact, each computing node has locally the vector elements corresponding to the rows of its sparse sub-matrix and, in order to compute its part of the SpMV, it also requires the vector elements of its neighboring nodes corresponding to
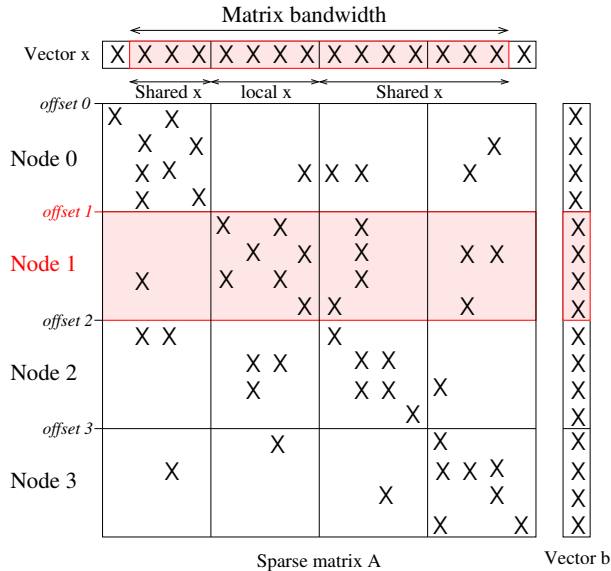
Figure 1: Data partitioning of the sparse matrix A, the solution vector x and the right-hand side b in 4 partitions

the column indices in which its local sub-matrix has nonzero values. Consequently, each computing node manages a global vector composed of a local vector of size $\frac{n}{p}$ and a shared vector of size S:

$$S = bw - \frac{n}{p}, \tag{4}$$

where $\frac{n}{p}$ is the size of the local vector and bw is the bandwidth of the local sparse sub-matrix which represents the number of columns between the minimum and the maximum column indices (see Figure 1). In order to improve memory accesses, we use the texture memory to cache elements of the global vector.

On a GPU cluster, the exchanges of the shared vectors elements between the neighboring nodes are performed as follows:

- at the level of the sending node: data transfers of the shared data from the GPU global memory to the CPU memory by using the CUBLAS communication routine `cublasGetVector()`,

- data exchanges between the CPUs by the MPI communication routine `MPI_Alltoallv()` and,

- at the level of the receiving node: data transfers of the received shared data from the CPU memory to the GPU global memory by using CUBLAS communication routine `cublasSetVector()`.

5.2 Experimentations

The experiments are done on a cluster composed of six machines interconnected by an Infiniband network of 20 GB/s. Each machine is a Xeon E5530 Quad-Core running at 2.4 GHz. It provides 12 GB of RAM memory with a memory bandwidth of 25.6 GB/s and it is equipped with two Tesla C1060 GPUs. Each GPU is composed of 240 cores running at 1.3 GHz and has 4 GB of global memory with a memory bandwidth of 102 GB/s. The GPU is connected to the CPU via a PCI-Express 16x Gen2.0 with a throughput of 8 GB/s. Figure 2 shows the general scheme of the GPU cluster.

Scientific Linux 5.10, with Linux version 2.6.18, is installed on the six machines. The C programming language is used for coding the GMRES algorithm on both the CPU and the GPU versions. CUDA version 4.0 [18] is used for programming the GPUs, using CUBLAS library [17] to deal with the functions operating on vectors. Finally, MPI routines of OpenMPI 1.3.3 are used to carry out the communication between the CPU cores.

The experiments are done on linear systems associated to sparse matrices chosen from the Davis collection of the University of Florida [20]. They are matrices arising in real-world applications. Table 1 shows the main characteristics of these sparse matrices and Figure 3 shows their sparse structures. For each matrix, we give the number of rows (column 3 in Table 1), the number of nonzero values (column 4) and the bandwidth (column 5).

All the experiments are performed on double-precision data. The parameters of the parallel GMRES algorithm are as follows: the tolerance threshold $\varepsilon = 10^{-12}$, the maximum number of iterations $max = 500$, the Arnoldi process is limited to $m = 16$ iterations, the elements of the guess solution x_0 is initialized to 0 and those of the right-hand
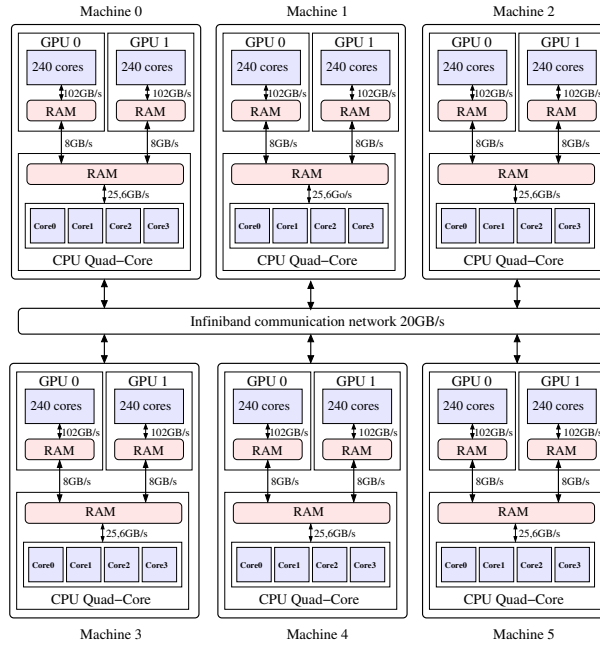
Figure 2: A cluster composed of six machines, each equipped with two Tesla C1060 GPUs

Matrix type	Name	# Rows	# Nonzeros	Bandwidth
Symmetric	2cubes_sphere	101 492	1 647 264	100 464
	ecology2	999 999	4 995 991	2 001
	finan512	74 752	596 992	74 725
	G3_circuit	1 585 478	7 660 826	1 219 059
	shallow_water2	81 920	327 680	58 710
	thermal2	1 228 045	8 580 313	1 226 629
Asymmetric	cage13	445 315	7 479 343	318 788
	crashbasis	160 000	1 750 416	120 202
	FEM_3D_thermal2	147 900	3 489 300	117 827
	language	399 130	1 216 334	398 622
	poli_large	15 575	33 074	15 575
	torso3	259 156	4 429 042	216 854

Table 1: Main characteristics of the sparse matrices chosen from the Davis collection

side vector are initialized to 1. For simplicity's sake, we chose the matrix preconditioning M as the main diagonal of the sparse matrix A. Indeed, it allows us to easily compute the required inverse matrix M^{-1} and it provides relatively good preconditioning in most cases. Finally, we set the size of a thread-block in GPUs to 512 threads. It should be noted that the same optimizations are performed on the CPU version and on the GPU version of the parallel GMRES algorithm.

In Table 2, we give the performances of the parallel GMRES algorithm for solving the linear systems associated to the sparse matrices shown in Table 1. The second and third columns show the execution times in seconds obtained on a cluster of 24 CPU cores and on a cluster of 12 GPUs, respectively. The fourth column shows the ratio τ between the CPU time $Time_{cpu}$ and the GPU time $Time_{gpu}$ that is computed as follows:

$$\tau = \frac{Time_{cpu}}{Time_{gpu}}. \tag{5}$$

From these ratios, we can notice that the use of many GPUs is not interesting to solve small sparse linear systems. Solving these sparse linear systems on a cluster of 12 GPUs is as fast as on a cluster of 24 CPU cores. Indeed, the small sizes of the sparse matrices do not allow to maximize the utilization of the GPU cores of the cluster. The fifth, sixth and seventh columns show, respectively, the number of iterations performed by the parallel GMRES algorithm to converge, the precision of the solution, $prec$, computed on the GPU cluster and the difference, Δ, between the solutions computed on the GPU and the GPU clusters. The last two parameters are used to validate the results
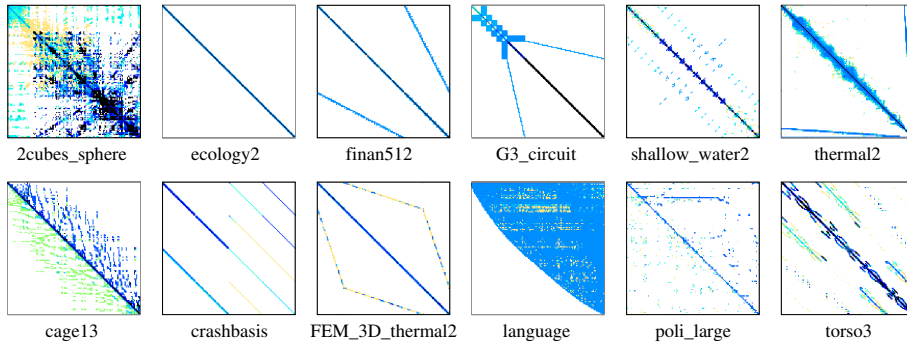
Figure 3: Structures of the sparse matrices chosen from the Davis collection

Matrix	$Time_{cpu}$	$Time_{gpu}$	τ	# iter	prec	Δ
2cubes_sphere	0.234 s	0.124 s	1.88	21	2.10e-14	3.47e-18
ecology2	0.076 s	0.035 s	2.15	21	4.30e-13	4.38e-15
finan512	0.073 s	0.052 s	1.40	17	3.21e-12	5.00e-16
G3_circuit	1.016 s	0.649 s	1.56	22	1.04e-12	2.00e-15
shallow_water2	0.061 s	0.044 s	1.38	17	5.42e-22	2.71e-25
thermal2	1.666 s	0.880 s	1.89	21	6.58e-12	2.77e-16
cage13	0.721 s	0.338 s	2.13	26	3.37e-11	2.66e-15
crashbasis	1.349 s	0.830 s	1.62	121	9.10e-12	6.90e-12
FEM_3D_thermal2	0.797 s	0.419 s	1.90	64	3.87e-09	9.09e-13
language	2.252 s	1.204 s	1.87	90	1.18e-10	8.00e-11
poli_large	0.097 s	0.095 s	1.02	69	4.98e-11	1.14e-12
torso3	4.242 s	2.030 s	2.09	175	2.69e-10	1.78e-14

Table 2: Performances of the parallel GMRES algorithm on a cluster of 24 CPU cores vs. a cluster of 12 GPUs

obtained on the GPU cluster and they are computed as follows:

$$prec = \|M^{-1}(b - Ax^{gpu})\|_\infty,$$
$$\Delta = \|x^{cpu} - x^{gpu}\|_\infty, \qquad (6)$$

where x^{cpu} and x^{gpu} are the solutions computed, respectively, on the CPU cluster and on the GPU cluster. We can see that the precision of the solutions computed on the GPU cluster are sufficient, they are about 10^{-10}, and the parallel GMRES algorithm computes almost the same solutions in both CPU and GPU clusters, with Δ varying from 10^{-11} to 10^{-25}.

Afterwards, we evaluate the performances of the parallel GMRES algorithm for solving large linear systems. We have developed in C programming language a generator of large sparse matrices having a band structure which arises in most numerical problems. This generator uses the sparse matrices of the Davis collection as the initial matrices to build the large band matrices. It is executed in parallel by all the MPI processes of the cluster so that each process constructs its own sub-matrix as a rectangular block of the global sparse matrix. Each process i computes the size n_i and the offset $offset_i$ of its sub-matrix in the global sparse matrix according to the size n of the linear system to be solved and the number of the GPU computing nodes p as follows:

$$n_i = \frac{n}{p}, \qquad (7)$$

$$offset_i = \begin{cases} 0 & \text{if } i = 0, \\ offset_{i-1} + n_{i-1} & \text{otherwise.} \end{cases} \qquad (8)$$

So each process i performs several copies of the same initial matrix chosen from the Davis collection and it puts all these copies on the main diagonal of the global matrix in order to construct a band matrix. Moreover, it fulfills the empty spaces between two successive copies by small copies, *lower_copy* and *upper_copy*, of the same initial matrix. Figure 4 shows a generation of a sparse bended matrix by four computing nodes.
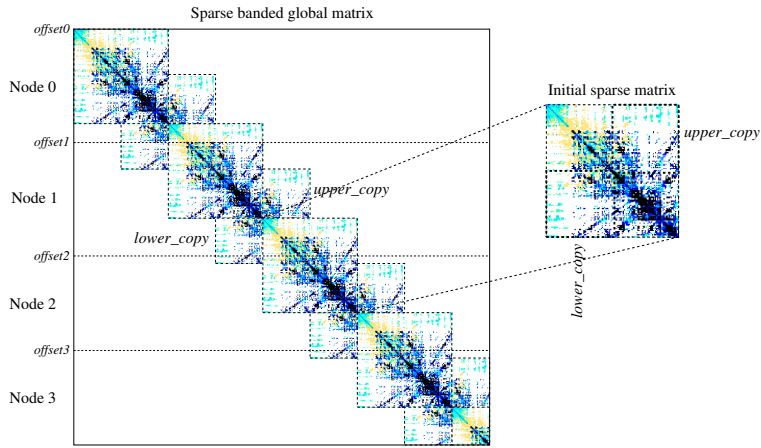
7

Figure 4: Example of the generation of a large sparse and band matrix by four computing nodes.

Table 3 shows the main characteristics (the number of nonzero values and the bandwidth) of the large sparse matrices generated from those of the Davis collection. These matrices are associated to the linear systems of 25 million of unknown values (each generated sparse matrix has 25 million rows). In Table 4 we show the performances of the parallel GMRES algorithm for solving large linear systems associated to the sparse band matrices of Table 3. The fourth column gives the ratio between the execution time spent on a cluster of 24 CPU cores and that spent on a cluster of 12 GPUs. We can notice from these ratios that for solving large sparse matrices the GPU cluster is more efficient (about 5 times faster) than the CPU cluster. The computing power of the GPUs allows to accelerate the computation of the functions operating on large vectors of the parallel GMRES algorithm.

Matrix type	Name	# nonzeros	Bandwidth
Symmetric	2cubes_sphere	413 703 602	198 836
	ecology2	124 948 019	2 002
	finan512	278 175 945	123 900
	G3_circuit	125 262 292	1 891 887
	shallow_water2	100 235 292	62 806
	thermal2	175 300 284	2 421 285
Asymmetric	cage13	435 770 480	352 566
	crashbasis	409 291 236	200 203
	FEM_3D_thermal2	595 266 787	206 029
	language	76 912 824	398 626
	poli_large	53 322 580	15 576
	torso3	433 795 264	328 757

Table 3: Main characteristics of the sparse and band matrices generated from the sparse matrices of the Davis collection.

6   Minimization of communications

The parallel sparse matrix-vector multiplication requires data exchanges between the GPU computing nodes to construct the global vector. However, a GPU cluster requires communications between the GPU nodes and the data transfers between the GPUs and their hosts CPUs. In fact, a communication between two GPU nodes implies: a data transfer from the GPU memory to the CPU memory at the sending node, a MPI communication between the CPUs of two GPU nodes, and a data transfer from the CPU memory to the GPU memory at the receiving node. Moreover, the data transfers between a GPU and a CPU are considered as the most expensive communications on a GPU cluster. For example in our GPU cluster, the data throughput between a GPU and a CPU is of 8 GB/s which is about twice lower than the data transfer rate between CPUs (20 GB/s) and 12 times lower than the memory bandwidth of the GPU global memory (102 GB/s). In this section, we propose two solutions to improve the execution time of the parallel GMRES algorithm on GPU clusters.

Matrix	$Time_{cpu}$	$Time_{gpu}$	τ	# $iter$	$prec$	Δ
2cubes_sphere	3.683 s	0.870 s	4.23	21	2.11e-14	8.67e-18
ecology2	2.570 s	0.424 s	6.06	21	4.88e-13	2.08e-14
finan512	2.727 s	0.533 s	5.11	17	3.22e-12	8.82e-14
G3_circuit	4.656 s	1.024 s	4.54	22	1.04e-12	5.00e-15
shallow_water2	2.268 s	0.384 s	5.91	17	5.54e-21	7.92e-24
thermal2	4.650 s	1.130 s	4.11	21	8.89e-12	3.33e-16
cage13	6.068 s	1.054 s	5.75	26	3.29e-11	1.59e-14
crashbasis	25.906 s	4.569 s	5.67	135	6.81e-11	4.61e-15
FEM_3D_thermal2	13.555 s	2.654 s	5.11	64	3.88e-09	1.82e-12
language	13.538 s	2.621 s	5.16	89	2.11e-10	1.60e-10
poli_large	8.619 s	1.474 s	5.85	69	5.05e-11	6.59e-12
torso3	35.213 s	6.763 s	5.21	175	2.69e-10	2.66e-14

Table 4: Performances of the parallel GMRES algorithm for solving large sparse linear systems associated to band matrices on a cluster of 24 CPU cores vs. a cluster of 12 GPUs.

6.1 Compressed storage format of the sparse vectors

In Section 5.1, the SpMV multiplication uses a global vector having a size equivalent to the matrix bandwidth (see Formula 4). However, we can notice that a SpMV multiplication does not often need all the vector elements of the global vector composed of the local and shared sub-vectors. For example in Figure 1, node 1 only needs a single vector element from node 0 (element 1), two elements from node 2 (elements 8 and 9) and two elements from node 3 (elements 13 and 14). Therefore to reduce the communication overheads of the unused vector elements, the GPU computing nodes must exchange between them only the vector elements necessary to perform their local sparse matrix-vector multiplications.
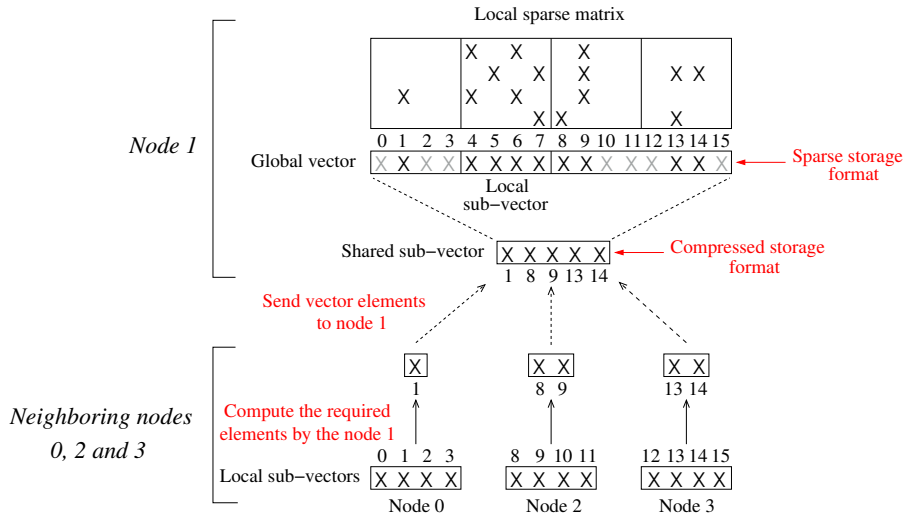

Figure 5: Example of data exchanges between node 1 and its neighbors 0, 2 and 3.

We propose to use a compressed storage format of the sparse global vector. In Figure 5, we show an example of the data exchanges between node 1 and its neighbors to construct the compressed global vector. First, the neighboring nodes 0, 2 and 3 determine the vector elements needed by node 1 and, then, they send only these elements to it. Node 1 receives these shared elements in a compressed vector. However to compute the sparse matrix-vector multiplication, it must first copy the received elements to the corresponding indices in the global vector. In order to avoid this process at each iteration, we propose to reorder the columns of the local sub-matrices so as to use the shared vectors in their compressed storage format (see Figure 6). For performance purposes, the computation of the shared data to send to the neighboring nodes is performed by the GPU as a kernel. In addition, we use the MPI point-to-point communication routines: a blocking send routine `MPI_Send()` and a nonblocking receive routine `MPI_Irecv()`.

Table 5 shows the performances of the parallel GMRES algorithm using the compressed storage format of the sparse global vector. The results are obtained from solving large linear systems associated to the sparse band matrices
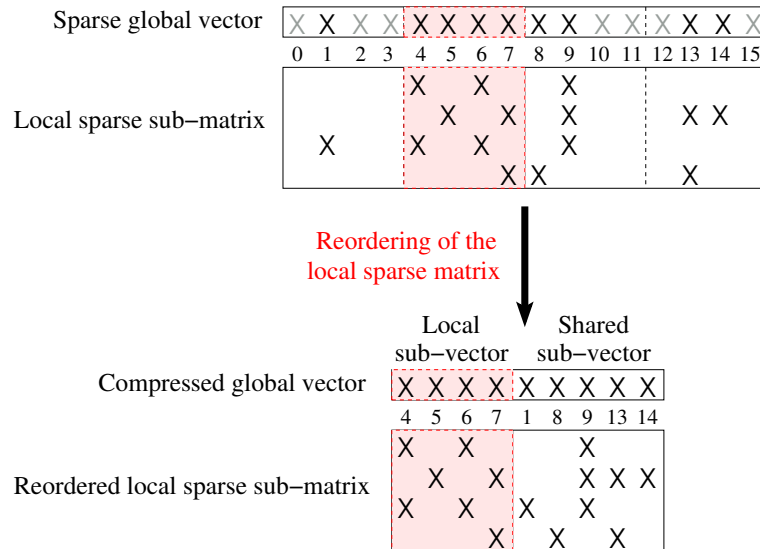
Figure 6: Reordering of the columns of a local sparse matrix.

presented in Table 3. We can see from Table 5 that the execution times of the parallel GMRES algorithm on a cluster of 12 GPUs are improved by about 38% compared to those presented in Table 4. In addition, the ratios between the execution times spent on the cluster of 24 CPU cores and those spent on the cluster of 12 GPUs have increased. Indeed, the reordering of the sparse sub-matrices and the use of a compressed storage format for the sparse vectors minimize the communication overheads between the GPU computing nodes.

Matrix	$Time_{cpu}$	$Time_{gpu}$	τ	# $iter$	$prec$	Δ
2cubes_sphere	3.597 s	0.514 s	6.99	21	2.11e-14	8.67e-18
ecology2	2.549 s	0.288 s	8.83	21	4.88e-13	2.08e-14
finan512	2.660 s	0.377 s	7.05	17	3.22e-12	8.82e-14
G3_circuit	3.139 s	0.480 s	6.53	22	1.04e-12	5.00e-15
shallow_water2	2.195 s	0.253 s	8.68	17	5.54e-21	7.92e-24
thermal2	3.206 s	0.463 s	6.93	21	8.89e-12	3.33e-16
cage13	5.560 s	0.663 s	8.39	26	3.29e-11	1.59e-14
crashbasis	25.802 s	3.511 s	7.35	135	6.81e-11	4.61e-15
FEM_3D_thermal2	13.281 s	1.572 s	8.45	64	3.88e-09	1.82e-12
language	12.553 s	1.760 s	7.13	89	2.11e-10	1.60e-10
poli_large	8.515 s	1.053 s	8.09	69	5.05e-11	6.59e-12
torso3	31.463 s	3.681 s	8.55	175	2.69e-10	2.66e-14

Table 5: Performances of the parallel GMRES algorithm using a compressed storage format of the sparse vectors for solving large sparse linear systems associated to band matrices on a cluster of 24 CPU cores vs. a cluster of 12 GPUs.

6.2 Hypergraph partitioning

In this section, we use another structure of the sparse matrices. We are interested in sparse matrices whose nonzero values are distributed along their large bandwidths. We developed in C programming language a generator of sparse matrices having five bands (see Figure 7). The principle of this generator is the same as the one presented in Section 5.2. However, the copies made from the initial sparse matrix, chosen from the Davis collection, are placed on the main diagonal and on two off-diagonals on the left and right of the main diagonal. Table 6 shows the main characteristics of sparse matrices of size 25 million of rows and generated from those of the Davis collection. We can see in the fourth column that the bandwidths of these matrices are as large as their sizes.

In Table 7 we give the performances of the parallel GMRES algorithm on the CPU and GPU clusters for solving large linear systems associated to the sparse matrices shown in Table 6. We can notice from the ratios given in the fourth column that solving sparse linear systems associated to matrices having large bandwidth on the GPU cluster is as fast as on the CPU cluster. This is due to the large total communication volume necessary to synchronize
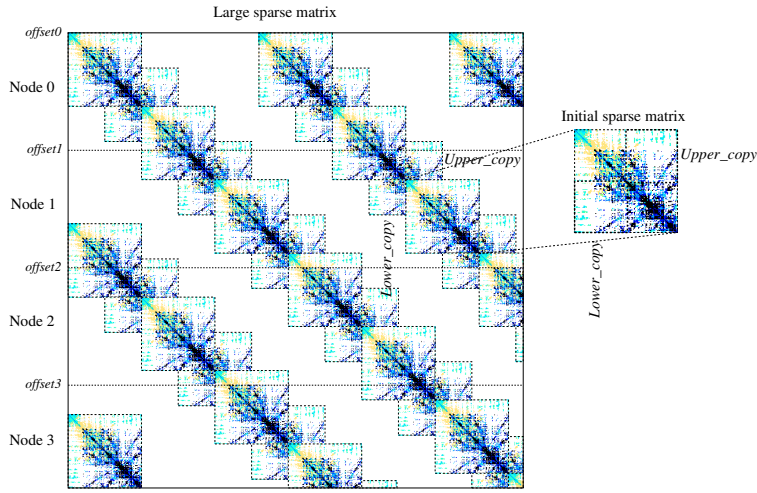
Figure 7: Example of the generation of a large sparse matrix having five bands by four computing nodes.

Matrix type	Name	# nonzeros	Bandwidth
Symmetric	2cubes_sphere	829 082 728	24 999 999
	ecology2	254 892 056	25 000 000
	finan512	556 982 339	24 999 973
	G3_circuit	257 982 646	25 000 000
	shallow_water2	200 798 268	25 000 000
	thermal2	359 340 179	24 999 998
Asymmetric	cage13	879 063 379	24 999 998
	crashbasis	820 373 286	24 999 803
	FEM_3D_thermal2	1 194 012 703	24 999 998
	language	155 261 826	24 999 492
	poli_large	106 680 819	25 000 000
	torso3	872 029 998	25 000 000

Table 6: Main characteristics of the sparse matrices having five band and generated from the sparse matrices of the Davis collection.

the computations over the cluster. In fact, the naive partitioning row-by-row or column-by-column of this type of sparse matrices links a GPU node to many neighboring nodes and produces a significant number of data dependencies between the different GPU nodes.

Matrix	$Time_{cpu}$	$Time_{gpu}$	τ	# iter	prec	Δ
2cubes_sphere	15.963 s	7.250 s	2.20	58	6.23e-16	3.25e-19
ecology2	3.549 s	2.176 s	1.63	21	4.78e-15	1.06e-15
finan512	3.862 s	1.934 s	1.99	17	3.21e-14	8.43e-17
G3_circuit	4.636 s	2.811 s	1.65	22	1.08e-14	1.77e-16
shallow_water2	2.738 s	1.539 s	1.78	17	5.54e-23	3.82e-26
thermal2	5.017 s	2.587 s	1.94	21	8.25e-14	4.34e-18
cage13	9.315 s	3.227 s	2.89	26	3.38e-13	2.08e-16
crashbasis	35.980 s	14.770 s	2.43	127	1.17e-12	1.56e-17
FEM_3D_thermal2	24.611 s	7.749 s	3.17	64	3.87e-11	2.84e-14
language	16.859 s	9.697 s	1.74	89	2.17e-12	1.70e-12
poli_large	10.200 s	6.534 s	1.56	69	5.14e-13	1.63e-13
torso3	49.074 s	19.397 s	2.53	175	2.69e-12	2.77e-16

Table 7: Performances of the parallel GMRES algorithm using a compressed storage format of the sparse vectors for solving large sparse linear systems associated to matrices having five bands on a cluster of 24 CPU cores vs. a cluster of 12 GPUs.
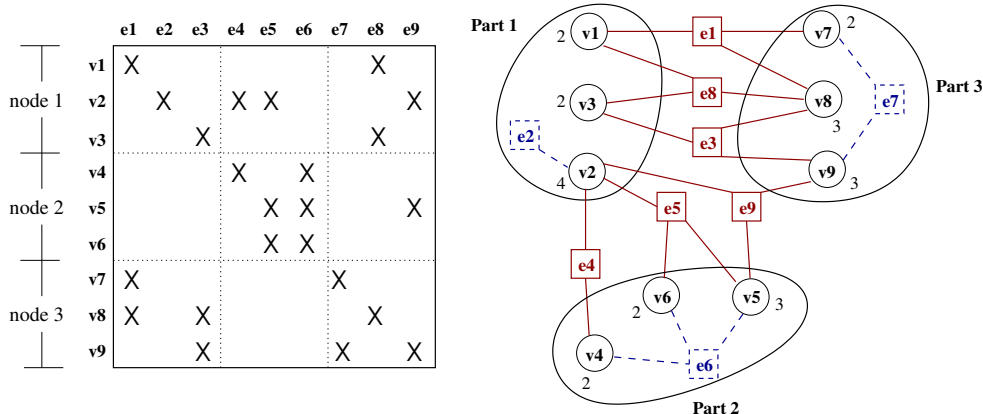
Figure 8: A hypergraph partitioning of a sparse matrix between three computing nodes.

We propose to use a hypergraph partitioning method which is well adapted to numerous structures of sparse matrices [11]. Indeed, it can well model the communications between the computing nodes especially for the asymmetric and rectangular matrices. It gives in most cases good reductions of the total communication volume. Nevertheless, it is more expensive in terms of execution time and memory space consumption than the partitioning method based on graphs.

The sparse matrix A of the linear system to be solved is modelled as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ as follows:

- each matrix row i ($0 \leq i < n$) corresponds to a vertex $v_i \in \mathcal{V}$,

- each matrix column j ($0 \leq j < n$) corresponds to a hyperedge $e_j \in \mathcal{E}$, such that: $\forall a_{ij}$ is a nonzero value of the matrix A, $v_i \in pins[e_j]$,

- w_i is the weight of vertex v_i,

- c_j is the cost of hyperedge e_j.

A K-way partitioning of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is defined as a set of K pairwise disjoint non-empty subsets (or parts) of the vertex set $\mathcal{V}$: $\mathcal{P} = \{\mathcal{V}_1, \ldots, \mathcal{V}_k\}$, such that $\mathcal{V} = \cup_{k=1}^{K} \mathcal{V}_k$. Each computing node is in charge of a vertex subset. Figure 8 shows an example of a hypergraph partitioning of a sparse matrix of size (9×9) into three parts. The circles and squares correspond, respectively, to the vertices and hyperedges of the hypergraph. The solid squares define the cut hyperedges connecting at least two different parts. The connectivity λ_j denotes the number of different parts spanned by the cut hyperedge e_j.

The cut hyperedges model the communications between the different GPU computing nodes in the cluster, necessary to perform the SpMV multiplication. Indeed, each hyperedge e_j defines a set of atomic computations $b_i \leftarrow b_i + a_{ij}x_j$ of the SpMV multiplication which needs the jth element of vector x. Therefore pins of hyperedge e_j ($pins[e_j]$) denote the set of matrix rows requiring the same vector element x_j. For example in Figure 8, hyperedge e_9 whose pins are: $pins[e_9] = \{v_2, v_5, v_9\}$ represents matrix rows 2, 5 and 9 requiring the vector element x_9 to compute in parallel the atomic operations: $b_2 \leftarrow b_2 + a_{29}x_9$, $b_5 \leftarrow b_5 + a_{59}x_9$ and $b_9 \leftarrow b_9 + a_{99}x_9$. However, x_9 is a vector element of the computing node 3 and it must be sent to the neighboring nodes 1 and 2.

The hypergraph partitioning allows to reduce the total communication volume while maintaining the computational load balance between the computing nodes. Indeed, it minimizes at best the following sum:

$$\mathcal{X}(\mathcal{P}) = \sum_{e_j \in \mathcal{E}_C} c_j(\lambda_j - 1), \tag{9}$$

where $\mathcal{E}_C$ is the set of the cut hyperedges issued from the partitioning $\mathcal{P}$, c_j and λ_j are, respectively, the cost and the connectivity of the cut hyperedge e_j. In addition, the hypergraph partitioning is constrained to maintain the load balance between the K parts:

$$W_k \leq (1 + \epsilon)W_{avg}, \ (1 \leq k \leq K) \text{ and } (0 < \epsilon < 1), \tag{10}$$

where W_k is the sum of the vertex weights in the subset $\mathcal{V}_k$, W_{avg} is the average part's weight and ϵ is the maximum allowed imbalanced ratio.

The hypergraph partitioning is a NP-complete problem but software tools using heuristics are developed, for example: hMETIS [28], PaToH [12] and Zoltan [21]. Due to the large sizes of the linear systems to be solved, we use a parallel hypergraph partitioning which must be performed by at least two MPI processes. The hypergraph model $\mathcal{H}$ of the sparse matrix is split into p (number of computing nodes) sub-hypergraphs $\mathcal{H}_k = (\mathcal{V}_k, \mathcal{E}_k)$, $0 \le k < p$, then the parallel partitioning is applied by using the MPI communication routines.

Table 8 shows the performances of the parallel GMRES algorithm for solving the linear systems associated to the sparse matrices presented in Table 6. In the experiments, we have used the compressed storage format of the sparse vectors and the parallel hypergraph partitioning developed in the Zoltan tool [2, 3]. The parameters of the hypergraph partitioning are initialized as follows:

- The weight w_i of each vertex v_i is set to the number of the nonzero values on the matrix row i,

- For simplicity's sake, the cost c_j of each hyperedge e_j is set to 1,

- The maximum imbalanced ratio ϵ is limited to 10%.

We can notice from Table 8 that the execution times on the cluster of 12 GPUs are significantly improved compared to those presented in Table 7. The hypergraph partitioning applied on the large sparse matrices having large bandwidths have improved the execution times on the GPU cluster by about 65%.

Matrix	$Time_{cpu}$	$Time_{gpu}$	τ	# $iter$	$prec$	Δ
2cubes_sphere	16.430 s	2.840 s	5.78	58	6.23e-16	3.25e-19
ecology2	3.152 s	0.367 s	8.59	21	4.78e-15	1.06e-15
finan512	3.672 s	0.723 s	5.08	17	3.21e-14	8.43e-17
G3_circuit	4.468 s	0.971 s	4.60	22	1.08e-14	1.77e-16
shallow_water2	2.647 s	0.312 s	8.48	17	5.54e-23	3.82e-26
thermal2	4.190 s	0.666 s	6.29	21	8.25e-14	4.34e-18
cage13	8.077 s	1.584 s	5.10	26	3.38e-13	2.08e-16
crashbasis	35.173 s	5.546 s	6.34	127	1.17e-12	1.56e-17
FEM_3D_thermal2	24.825 s	3.113 s	7.97	64	3.87e-11	2.84e-14
language	16.706 s	2.522 s	6.62	89	2.17e-12	1.70e-12
poli_large	12.715 s	3.989 s	3.19	69	5.14e-13	1.63e-13
torso3	48.459 s	6.234 s	7.77	175	2.69e-12	2.77e-16

Table 8: Performances of the parallel GMRES algorithm using a compressed storage format of the sparse vectors and a hypergraph partitioning method for solving large sparse linear systems associated to matrices having five bands on a cluster of 24 CPU cores vs. a cluster of 12 GPUs.

Table 9 shows in the second, third and fourth columns the total communication volume on a cluster of 12 GPUs by using row-by-row partitioning or hypergraph partitioning and compressed format. The total communication volume defines the total number of the vector elements exchanged between the 12 GPUs. From these columns we can see that the two heuristics, compressed format for the vectors and the hypergraph partitioning, minimize the number of vector elements to be exchanged over the GPU cluster. The compressed format allows the GPUs to exchange the needed vector elements without any communication overheads. The hypergraph partitioning allows to split the large sparse matrices so as to minimize data dependencies between the GPU computing nodes. However, we can notice in the fifth column that the hypergraph partitioning takes longer than the computation times. As we have mentioned before, the hypergraph partitioning method is less efficient in terms of memory consumption and partitioning time than its graph counterpart. So for the applications which often use the same sparse matrices, we can perform the hypergraph partitioning only once and, then, we save the traces in files to be reused several times. Therefore, this allows us to avoid the partitioning of the sparse matrices at each resolution of the linear systems.

Hereafter, we show the influence of the communications on a GPU cluster compared to a CPU cluster. In Tables 10, 11 and 12, we compute the ratios between the computation time over the communication time of three versions of the parallel GMRES algorithm to solve sparse linear systems associated to matrices of Table 6. These tables show that the hypergraph partitioning and the compressed format of the vectors increase the ratios either on the GPU cluster or on the CPU cluster. That means that the two optimization techniques allow the minimization of the total communication volume between the computing nodes. However, we can notice that the ratios obtained on the GPU cluster are lower than those obtained on the CPU cluster. Indeed, GPUs compute faster than CPUs but

Matrix	Total comm. vol. using row-by row partitioning	Total comm. vol. using compressed format	Total comm. vol. using hypergraph partitioning and compressed format	Time of hypergraph partitioning in minutes
2cubes_sphere	182 061 791	25 360 543	240 679	68.98
ecology2	181 267 000	26 044 002	73 021	4.92
finan512	182 090 692	26 087 431	900 729	33.72
G3_circuit	192 244 835	31 912 003	5 366 774	11.63
shallow_water2	181 729 606	25 105 108	60 899	5.06
thermal2	191 350 306	30 012 846	1 077 921	17.88
cage13	183 970 606	28 254 282	3 845 440	196.45
crashbasis	182 931 818	29 020 060	2 401 876	33.39
FEM_3D_thermal2	182 503 894	25 263 767	250 105	49.89
language	183 055 017	27 291 486	1 537 835	9.07
poli_large	181 381 470	25 053 554	7 388 883	5.92
torso3	183 863 292	25 682 514	613 250	61.51

Table 9: Total communication volume on a cluster of 12 GPUs using row-by-row or hypergraph partitioning methods and compressed vectors. The total communication volume is defined as the total number of vector elements exchanged between all GPUs of the cluster.

with GPUs there are more communications due to CPU/GPU communications, so communications are more time-consuming while the computation time remains unchanged. Furthermore, we can notice that the GPU computation times on Tables 11 and 12 are about 10% lower than those on Table 10. Indeed, the compression of the vectors and the reordering of matrix columns allow to perform coalesced accesses to the GPU memory and thus accelerate the sparse matrix-vector multiplication.

Figure 9 presents the weak scaling of four versions of the parallel GMRES algorithm on a GPU cluster. We fixed the size of a sub-matrix to 5 million of rows per GPU computing node. We used matrices having five bands generated from the symmetric matrix thermal2. This figure shows that the parallel GMRES algorithm, in its naive version or using either the compression format for vectors or the hypergraph partitioning, is not scalable on a GPU cluster due to the large amount of communications between GPUs. In contrast, we can see that the algorithm using both optimization techniques is fairly scalable. That means that in this version the cost of communications is relatively constant regardless the number of computing nodes in the cluster.

Finally, as far as we are concerned, the parallel solving of a linear system can be easy to optimize when the associated matrix is regular. This is unfortunately not the case for many real-world applications. When the matrix has an irregular structure, the amount of communications between processors is not the same. Another important parameter is the size of the matrix bandwidth which has a huge influence on the amount of communications. In this work, we have generated different kinds of matrices in order to analyze several difficulties. With a bandwidth as large as possible, involving communications between all processors, which is the most difficult situation, we proposed to use two heuristics. Unfortunately, there is no fast method that optimizes the communications in any situation. For systems of non linear equations, there are different algorithms but most of them consist in linearizing the system of equations. In this case, a linear system needs to be solved. The big interest is that the matrix is the same at each step of the non linear system solving, so the partitioning method which is a time consuming step is performed only once.

Another very important issue, which might be ignored by too many people, is that the communications have a greater influence on a cluster of GPUs than on a cluster of CPUs. There are two reasons for that. The first one comes from the fact that with a cluster of GPUs, the CPU/GPU data transfers slow down communications between two GPUs that are not on the same machines. The second one is due to the fact that with GPUs the ratio of the computation time over the communication time decreases since the computation time is reduced. So the impact of the communications between GPUs might be a very important issue that can limit the scalability of parallel algorithms.

7 Conclusion and perspectives

In this paper, we have aimed at harnessing the computing power of a GPU cluster for solving large sparse linear systems. We have implemented the parallel algorithm of the GMRES iterative method. We have used a heteroge-

Matrix	GPU version			CPU version		
	$Time_{comput}$	$Time_{comm}$	$Ratio$	$Time_{comput}$	$Time_{comm}$	$Ratio$
2cubes_sphere	37.067 s	1434.512 s	**0.026**	312.061 s	3453.931 s	**0.090**
ecology2	4.116 s	501.327 s	**0.008**	60.776 s	1216.607 s	**0.050**
finan512	7.170 s	386.742 s	**0.019**	72.464 s	932.538 s	**0.078**
G3_circuit	4.797 s	537.343 s	**0.009**	66.011 s	1407.378 s	**0.047**
shallow_water2	3.620 s	411.208 s	**0.009**	51.294 s	973.446 s	**0.053**
thermal2	6.902 s	511.618 s	**0.013**	77.255 s	1281.979 s	**0.060**
cage13	12.837 s	625.175 s	**0.021**	139.178 s	1518.349 s	**0.092**
crashbasis	48.532 s	3195.183 s	**0.015**	623.686 s	7741.777 s	**0.081**
FEM_3D_thermal2	37.211 s	1584.650 s	**0.023**	370.297 s	3810.255 s	**0.097**
language	22.912 s	2242.897 s	**0.010**	286.682 s	5348.733 s	**0.054**
poli_large	13.618 s	1722.304 s	**0.008**	190.302 s	4059.642 s	**0.047**
torso3	74.194 s	4454.936 s	**0.017**	897.440 s	10800.787 s	**0.083**

Table 10: Ratios of the computation time over the communication time obtained from the parallel GMRES algorithm using row-by-row partitioning on 12 GPUs and 24 CPUs.

Matrix	GPU version			CPU version		
	$Time_{comput}$	$Time_{comm}$	$Ratio$	$Time_{comput}$	$Time_{comm}$	$Ratio$
2cubes_sphere	27.386 s	154.861 s	**0.177**	342.255 s	42.100 s	**8.130**
ecology2	3.822 s	53.131 s	**0.072**	69.956 s	15.019 s	**4.658**
finan512	6.366 s	41.155 s	**0.155**	79.592 s	8.604 s	**9.251**
G3_circuit	4.543 s	63.132 s	**0.072**	76.540 s	27.371 s	**2.796**
shallow_water2	3.282 s	43.080 s	**0.076**	58.348 s	8.088 s	**7.214**
thermal2	5.986 s	57.100 s	**0.105**	87.682 s	28.544 s	**3.072**
cage13	10.227 s	70.388 s	**0.145**	152.718 s	30.785 s	**4.961**
crashbasis	41.527 s	369.071 s	**0.113**	701.040 s	158.916 s	**4.411**
FEM_3D_thermal2	28.691 s	167.140 s	**0.172**	403.510 s	50.935 s	**7.922**
language	22.408 s	242.589 s	**0.092**	333.119 s	64.409 s	**5.172**
poli_large	13.710 s	179.208 s	**0.077**	215.934 s	30.903 s	**6.987**
torso3	58.455 s	480.315 s	**0.122**	993.609 s	152.173 s	**6.529**

Table 11: Ratios of the computation time over the communication time obtained from the parallel GMRES algorithm using row-by-row partitioning and compressed format for vectors on 12 GPUs and 24 CPUs.

Matrix	GPU version			CPU version		
	$Time_{comput}$	$Time_{comm}$	$Ratio$	$Time_{comput}$	$Time_{comm}$	$Ratio$
2cubes_sphere	28.440 s	7.768 s	**3.661**	327.109 s	63.788 s	**5.128**
ecology2	3.652 s	0.757 s	**4.823**	63.632 s	13.520 s	**4.707**
finan512	7.579 s	4.569 s	**1.659**	74.120 s	22.505 s	**3.294**
G3_circuit	4.876 s	8.745 s	**0.558**	72.280 s	28.395 s	**2.546**
shallow_water2	3.146 s	0.606 s	**5.191**	52.903 s	11.177 s	**4.733**
thermal2	6.473 s	4.325 s	**1.497**	81.171 s	20.907 s	**3.882**
cage13	11.676 s	7.723 s	**1.512**	145.755 s	46.547 s	**3.131**
crashbasis	42.799 s	29.399 s	**1.456**	650.386 s	203.918 s	**3.189**
FEM_3D_thermal2	29.875 s	8.915 s	**3.351**	382.887 s	93.252 s	**4.106**
language	20.991 s	11.197 s	**1.875**	310.679 s	82.480 s	**3.767**
poli_large	13.817 s	102.760 s	**0.134**	197.508 s	151.672 s	**1.302**
torso3	57.469 s	16.828 s	**3.415**	926.588 s	242.721 s	**3.817**

Table 12: Ratios of the computation time over the communication time obtained from the parallel GMRES algorithm using hypergraph partitioning and compressed format for vectors on 12 GPUs and 24 CPUs.
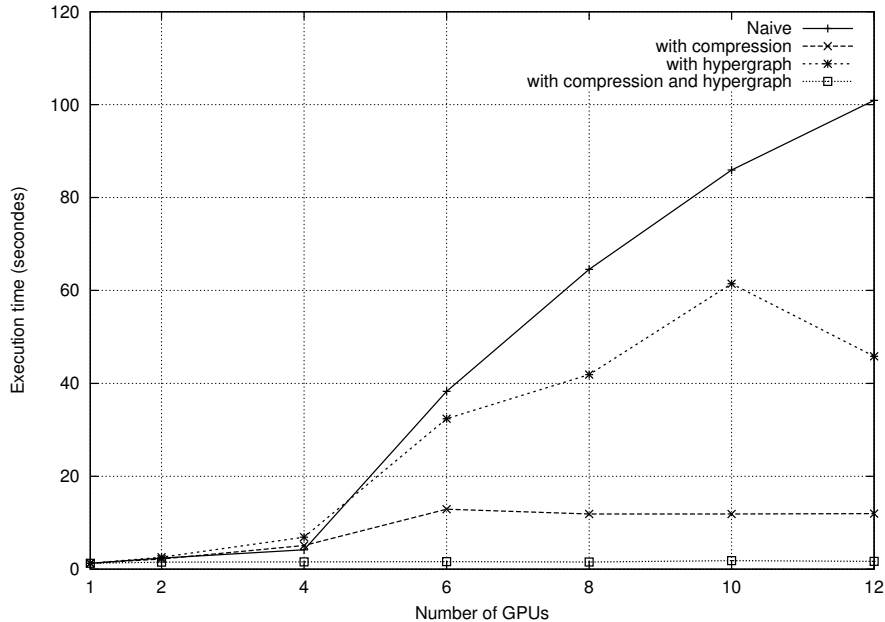
Figure 9: Weak scaling of the parallel GMRES algorithm on a GPU cluster.

neous parallel programming based on the CUDA language to program the GPUs and the MPI parallel environment to distribute the computations between the GPU nodes on the cluster.

The experiments have shown that solving large sparse linear systems is more efficient on a cluster of GPUs than on a cluster of CPUs. However, the efficiency of a GPU cluster is ensured as long as the spatial and temporal localization of the data is well managed. The data dependency scheme on a GPU cluster is related to the sparse structures of the matrices (positions of the nonzero values) and the number of the computing nodes. We have shown that a large number of communications between the GPU computing nodes affects considerably the performances of the parallel GMRES algorithm on the GPU cluster. Therefore, we have proposed to reorder the columns of the sparse local sub-matrices on each GPU node and to use a compressed storage format for the sparse vector involved in the parallel sparse matrix-vector multiplication. This solution allows to minimize the communication overheads. In addition, we have shown that it is interesting to choose a partitioning method according to the structure of the sparse matrix. This reduces the total communication volume between the GPU computing nodes.

In future works, it would be interesting to implement and study the scalability of the parallel GMRES algorithm on large GPU clusters (hundreds or thousands of GPUs) or on geographically distant GPU clusters. In this context, other methods might be used to reduce communication and to improve the performances of the parallel GMRES algorithm as the multisplitting methods. The recent GPU hardware and software architectures provide the GPU-Direct system which allows two GPUs, placed in the same machine or in two remote machines, to exchange data without using CPUs. This improves the data transfers between GPUs. Finally, it would be interesting to implement other iterative methods on GPU clusters for solving large sparse linear or non linear systems.

References

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

[2] PHG - Parallel hypergraph and graph partitioning with Zoltan. `http://www.cs.sandia.gov/Zoltan/ug_html/ug_alg_phg.html`.

[3] Zoltan : Parallel partitioning, load balancing and data-management services. user's guide. `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`.

[4] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform. *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 583–592, 2010.

[5] W. Arnoldi. The principle of minimized iteration in the solution of the matrix eigenvalue problem. *Quart, Appl.Math.*, 9:17–29, 1951.

[6] J. Bahi, S. Contassot-Vivier, and R. Couturier. Parallel iterative algorithms: from sequential to grid computing. *Chapman & Hall/CRC Numerical Analysis and Scientific Computing*, 2008.

[7] J. Bahi, R. Couturier, and L. Ziane Khodja. Parallel GMRES implementation for solving sparse linear systems on GPU clusters. pages 12–19, 2011.

[8] J. Bahi, R. Couturier, and L. Ziane Khodja. Parallel sparse linear solver gmres for gpu clusters with compression of exchanged data. *Euro-Par 2011: Parallel Processing Workshops*, 7155:471–480, 2012.

[9] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. pages 1–11, 2009.

[10] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[11] U. Catalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, 1999.

[12] U. Catalyürek and C. Aykanat. PaToH: Partitioning tool for hypergraphs. 1999.

[13] A. Cevahir, A. Nukada, and S. Matsuoka. Fast conjugate gradients with multiple GPUs. *Computational Science ICCS 2009*, 5544:893–903, 2009. .

[14] A. Cevahir, A. Nukada, and S. Matsuoka. High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. *Computer Science - Research and Development*, 25:83–91, 2010.

[15] C. Chen and T. Taha. A communication reduction approach to iteratively solve large sparse linear systems on a GPGPU cluster. *Cluster Computing*, pages 1–11, 2013.

[16] S. Contassot-Vivier, T. Jost, and S. Vialle. Impact of asynchronism on GPU accelerated parallel iterative computations. 7133:43–53, 2012.

[17] N. Corporation. CUDA Toolkit 4.2 CUBLAS Library. 2012. `http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf`.

[18] N. Corporation. NVIDIA CUDA C Programming Guide. *Version 4.2*, 2012. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.

[19] R. Couturier and S. Domas. Sparse systems solving on GPUs with GMRES. *The journal of Supercomputing*, 59(3):1504–1516, 2012.

[20] T. Davis and Y. Hu. The university of florida sparse matrix collection. 1997. Digest, http://www.cise.ufl.edu/research/sparse/matrices/.

[21] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 124–124. IEEE Computer Society, 2006.

[22] B. DeVries, J. Iannelli, C. Trefftz, K. O'Hearn, and G. Wolffe. Parallel implementations of {FGMRES} for solving large, sparse non-symmetric linear systems. *Procedia Computer Science*, 18(0):491–500, 2013.

[23] A. Gaikwad and I. Toke. Parallel iterative linear solvers on GPU: a financial engineering case. *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 607–614, 2010.

[24] N. Ghaemian, A. Abdollahzadeh, Z. Heinemann, A. Harrer, M. Sharifi, and G. Heinemann. Accelerating the GMRES iterative linear solver of an oil reservoir simulator using the multi-processing power of compute unified device architecture of graphics cards. 2010. .

[25] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing, Special issue: High-performance computing using accelerators*, 33(10–11):685–699, 2007.

[26] G. Haase, M. Liebmann, C. Douglas, and G. Plank. A parallel algebraic multigrid solver on graphics processing units. *High Performance Computing and Applications*, 5938:38–47, 2010. .

[27] T. Jost, S. Contassot-Vivier, and S. Vialle. An efficient multi-algorithms sparse linear solver for GPUs. *International conference on parallel computing, ParCo2009*, 2009. .

[28] G. Karypis and V. Kumar. hMETIS: A hypergraph partitioning package. 1998.

[29] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013. .

[30] A. Neic, M. Liebmann, G. Haase, and G. Plank. Algebraic multigrid solver on clusters of CPUs and GPUs. *Applied Parallel and Scientific Computing*, 7134:389–398, 2012. .

[31] Nvidia. NVIDIA CUDA C programming guide. 2010. Version 4.0.

[32] C. Paige and M. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12(4):617–629, 1975.

[33] Y. Saad and M. Schultz. GMRES : a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

[34] M. Wang, H. Klie, M. Parashar, and H. Sudan. Solving sparse linear systems on NVIDIA Tesla GPUs. *Computational Science ICCS 2009*, 5544:864–873, 2009.

[35] D. Weber, J. Bender, M. Schnoes, A. Stork, and D. Fellner. Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. *Computer Graphics Forum*, 32:16–26, 2013. .

[36] N. Zhao and X. Wang. A parallel preconditioned Bi-Conjugate Gradient stabilized solver for the Poisson problem. *Journal of Computers*, 7(12), 2012.