A Data Parallel Scientific Computing Introduction

Serge G. Petiton* and Nahid Emad†

* Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
Cité Scientifique, 59655 Villeneuve d'Ascq Cedex, France
petiton@lifl.fr

† Laboratoire PRiSM
Université de Versailles
45 Avenue des États Unis, 78000 Versailles, France
emad@prism.uvsq.fr

Abstract. We first present data parallel algorithms for classical linear algebra methods. We analyze some of the main problems that a user has to solve. As examples, we propose data parallel algorithms for the Gauss and Gauss-Jordan methods. Thus, we introduce some criteria, such as the average data parallel computation ratio, to evaluate and compare data parallel algorithms. Our studies include both dense and sparse matrix computations. We describe in detail a data parallel structure to map general sparse matrices and we present data parallel sparse matrix-vector multiplication. Then, we propose a data parallel preconditioned conjugate gradient algorithm using these matrix vector operations.

1 Introduction

Many scientific problems lead to linear algebra problems such as linear system resolution or eigenvalue computation of linear operators [3, 4, 7]. Then, efficient computation of these linear algebra problems which manipulate regular or irregular data structures on data parallel computers is an important goal for the future of scientific computing.

Data structures are often matrices or graphs which come from very large meshes. Applications which justify the computing power and memory sizes of the larger available parallel computers often manipulate very large sparse matrices or meshes with a lot of nodes.

Since less than a decade, data parallel computation has been used to solve large applications [23, 14]. Nevertheless, there does not exist any general data parallel programming methodology. We will introduce some hypothesis and our own parameter definitions to be able to specify and evaluate data parallel algorithms.

We study in this paper only problems from very large applications which call for the most powerful parallel computers with the larger memory sizes.

2 The data parallel programming model

The data parallel programming model was often associated with the massively parallel computing and with SIMD machines. The recent evolution of parallel machines, languages and compilers illustrates how it was a premature assumption. The number of physical processors of the recent larger parallel machines is not very large. Thus, the definition of the data parallelism need to be independent of target machines and compilers.

A data parallel algorithm is defined by a unique control flow of instructions. Each of them is executed on a set of data, structured onto a virtual geometry. In this paper, we consider only two or one dimensional virtual geometries. Some data exchanges between virtual processors can be explicitly done. They are characterized as follows:

local data exchanges between neighboring virtual processors on the geometry.

logarithmic data exchanges along a predefined dimension of the geometry; generated for example by the sum of elements mapped on one column or row of a two-dimensional virtual geometry. The number of data parallel elementary operation step and basic communication operations is logarithmic.

general data exchanges between any two virtual processors.

Data parallel algorithms will be presented using an algorithmic language. The virtual geometry indexes will be called `ig` and `jg`. Each data parallel instruction will be associated with its data subset; for example `(ig > k and jg < k)`.

We assume in this paper that the number of virtual processors is, at least, of the same order as the data or of the order of the total memory size of a potential target machine. Nevertheless, the data size is often really larger that the number of physical processors of the more massively parallel computer available on the supercomputing market. Then, on these machines, the number of virtual processors compared to the number of physical processors is large. We have to simulate many virtual processors on each physical processor. In this case, the parallelism between operations done on the same physical processor is destroyed. In this paper, we consider that the best data parallel algorithm will be the one will have the fewest data parallel operations, independent of the global time obtained on a target machine. The reader should remember that all the algorithms and discussions on this paper will be under these hypothesis. We will discuss this point on the following sections after the study of a few data parallel algorithms and complexity.

In this paper, we will present only algorithms. We will not propose any codes. The translation into HPF, CM-Fortran , C* or DPCE would generate the definition of some layout or special templates with respect to the target machine. We do not consider the exact number of physical processors to define a data parallel algorithm in this paper.

3 Data parallel algorithm examples

We present in this section a few well-known linear algebra examples and their data parallel adaptations.

APλA: matrix + (scalar $\times$ matrix) The sum of two matrices, A and B, of the same size $(n \times n)$ is an easy example. Each element (i, j) of the matrices is mapped on a two dimensional virtual geometry $n \times n$ so that the elements $A(i, j)$ are aligned with the elements $B(i, j)$; i.e. each virtual processor (i_g, j_g) of the geometry store $A(i_g, j_g)$ and $B(i_g, j_g)$. In just one data parallel addition the matrix addition is done. The result $C(i, j) = A(i, j) + B(i, j)$ is aligned with the two input matrices. The computation of $C = A + \lambda B$, λ is a scalar, is also very easy; we map λ on each virtual processor and we compute a triadic data parallel operation $(+, \times)$.

AXPX: (matrix $\times$ vector) + vector The matrix-vector multiplication is a key-example on data parallel numerical analysis computing. In this paper, we study the data parallel algorithm to compute $Ax+x$, and then, $A(Ax+x)+x$. These computations are similar to those of many iterative methods [17].

We map each element $A(i, j)$ on a two-dimensional virtual geometry and we align the vector x on each row of the geometry. Thus, the vector is redundantly row-mapped. We remark that on each virtual processor (i_g, j_g) we have $A(i_g, j_g)$ and $x(j_g)$. Then, in just one data parallel operation, we compute all the necessary multiplications, i.e. $A(i, j) * x(j)$ for $i = 1, n$ and $j = 1, n$. The sum of all these partial results mapped on each row of the virtual geometry give one element of the matrix-vector per row, in $log_2(n)$ elementary data parallel operations. The i^{th} row of the geometry store the i^{th} component of Ax. To compute $Ax + x$, we now want to add this element with the i^{th} element of x. We remark that this one is mapped on each column of the geometry. Then, only the virtual processor of the diagonal of the virtual geometry have the correct two elements to add. We already remark that the results $Ax + x$ will not be aligned with the row of the virtual geometry. Then, to be able to compute $A(Ax + x) + x$, we have to re-mapped these results. We have to align each result $Ax + x$ calculated on the diagonal with each row of the virtual geometry, see Figure 1. These operations can be optimized with respect to the target machines and available languages.

To compute $A(A(x))$, and then, $A^m(x) = A(A(...A(x)))$, we also have a re-mapping problem. The vector $A(x)$ is column mapped and have to be re-mapped before the following matrix vector multiplication.

Gaussian Elimination: The sequential Gaussian Elimination without pivoting to triangularize a linear system is

```
do k=1,n-1
  do i=k+1,n
    a(i,k)=a(i,k)/a(k,k)
    do j=k+1,n
      a(i,j)=a(i,j)-a(i,k)*a(k,j)
    end do
  end do
```

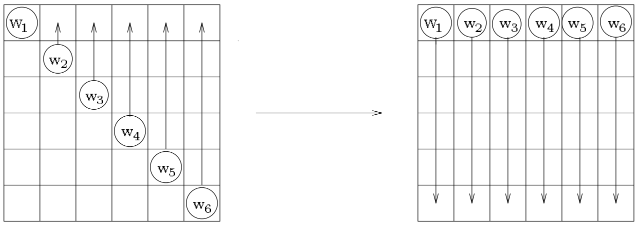

Fig. 1. Re-mapping example of the vector $w = Ax + x$ to compute Aw, $n = 6$; when broadcast from the diagonal is not possible.

```
end do
```

Then, a parallel algorithm is:

```
do k=1,n-1
  doall i=k+1,n
    a(i,k) = a(i,k)/a(k,k)
  end doall
  doall j=k+1,n and i=k+1,j
    a(i,j)=a(i,j)-a(i,k)*a(k,j)
  end doall
end do
```

Let a two-dimensional virtual geometry of size $(n \times n)$, we map the matrix as done for the matrix addition. Let `a`, `w` and `u` be some data parallel variables, the data parallel Gauss method is:

```
do k=1,n-1
 spread a {ig=k;jg=k,n} along the first dimension
       to w {ig=k+1,n;jg=k,n}.
 u=a/w {ig=k+1,n;jg=k}.
 spread u {ig=k+1,n;jg=k}along the second dimension
       to u {ig=k+1,n;jg=k+1,n}.
 a = a - u * w (ig=k+1,n;jg=k+1,n).
end do
```

Figure 2 illustrates the data mapping at the k^{th} step of the Gaussian elimination. The elements $\circ$ are spread to the elements $\bullet$ and $\times$ along the first dimension. Then, one data parallel division is done on each elements $\times$ and the result is spread to the elements $\bullet$ along the second dimension. The data parallel triadic $(+, \times)$ is computed on each element $\bullet$. We remark that the number of affected data elements decreases after each step. At the k^{th} step of the Gaussian elimination, the division and the triadic data parallel operations involve only, respectively, $(n - k + 1)$ and $(n - k)^2$ data. As the number decreases, we say that the Gaussian elimination is **semi data parallel**. Half of the data parallel operations are spread operations along a geometry axis. Only the triadic data parallel operations involve a large part of the subset of data concerned at a fixed step of the Gaussian elimination.

$$\left(\begin{array}{c|c|c}
U_{1,1}^{k-1} & U_{1,2}^{k-1} & U_{1,3}^{k-1} \\
\hline
0 & \otimes & \circ\ \circ\ \circ\ \circ\ \circ\ \circ \\
\hline
 & \times & \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet \\
 & \times & \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet \\
0 & \times & \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet \\
 & \times & \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet \\
 & \times & \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet \\
 & \times & \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet
\end{array}\right)$$

Fig. 2. Read and wrote elements of the matrix at the step k of the Gaussian Elimination.

Back substitution The obtained triangular linear system is solved using the classical back substitution. We suppose that the right hand side of the linear system was updated during the Gaussian elimination. We do not develop in details the data parallel back substitution because less than n data are concerned for each data parallel operation. Then, only, at least, one n^{th} of the number of virtual processor are selected; if we use a two-dimensional geometry to respect the hypothesis. We conclude that this algorithm is not data parallel. At each step of the back substitution we exchange data between virtual processors. Figure 3, we present as an example, the generated communications between the computation of x_6 and x_5, for $n = 8$.

$$\left(\begin{array}{cccccccc}
\cdot & \cdot & \circ \rightarrow & \circ \rightarrow & \circ \rightarrow & \circ \rightarrow & \circ \rightarrow & \bullet \\
\cdot & \cdot & \cdot & \circ \rightarrow & \circ \rightarrow & \circ \rightarrow & \circ \rightarrow & \bullet \\
\cdot & \cdot & \cdot & \cdot & \circ \rightarrow & \circ \rightarrow & \circ \rightarrow & \bullet \\
\cdot & \cdot & \cdot & \cdot & \cdot & \circ \rightarrow & \circ \rightarrow & \bullet \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \circ \rightarrow & \bullet \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & x_6 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}\right)$$

Fig. 3. Back substitution; data exchanges between the computation of x_6 and x_5, for $n = 8$.

Gauss-Jordan Diagonalization The Gauss-Jordan method to diagonalize a linear system is:

```
do k=1,n
  do i=k+1,n if k < n
    a(i,k)=a(i,k)/a(k,k)
    do j=k+1,n
      a(i,j)=a(i,j)-a(i,k)*a(k,j)
    end do
  end do
  do i=1,k-1 if k > 0
    a(i,k)=a(i,k)/a(k,k)
    do j=k+1,n
      a(i,j)=a(i,j)-a(i,k)*a(k,j)
    end do
  end do
end do
```

We map the data as proposed for the Gaussian elimination; the data parallel algorithm is adapted from the Gaussian one:

```
do k=1,n-1
  spread a {ig=k;jg=k,n}
      along the first dimension
      to w {ig=k+1,n;jg= k,n and ig=1,k-1;jg=k,n}.
  u=a/w {ig=k+1,n;jg=k and ig=1,k-1;jg=k}.
  spread u {ig=k+1,n;jg=k and ig=1,k-1;jg=k}
      along the second dimension
      to u {ig=k+1,n;jg=k+1,n and ig=1,k-1;jg=k+1,n}.
  a = a - u * w {ig=k+1,n;jg=k+1,n and ig=1,k-1;jg=k+1,n}
end do
```

The data subsets of the virtual geometry involved in each operation are different from those of the Gaussian elimination. This is the unique variation of the data parallel algorithm. Figure 4 presents the elements of the k^{th} step of the Gauss-Jordan Method. The $\circ$ elements are spread to the $\bullet$ or $\times$ elements. The data parallel division is done on the $\times$ subset and the results sent to the $\bullet$. The $(+, *)$ data parallel triadic operation is compute on all the $\bullet$. Then, we always have less than $(n-k)(n-1)$ data concerned.

We diagonalize the matrix and we have no back substitution to compute. This is the main advantage of using the Gauss-Jordan method instead of the Gaussian one. The Gauss-Jordan diagonalization have the same number of data parallel operations that the Gaussian elimination. The back substitution bottleneck is eliminated. The Gauss-Jordan rehabilitation, started with the vector machines [22] and large granularity MIMD machines [15], is completed here.

The QR method This iterative method is really often used to compute all the eigenvalues of a non-Hermitian Hessenberg form matrix; any matrix can

$$
\left(\begin{array}{c|c|c}
 & \times & \bullet\bullet\bullet\bullet\bullet\bullet \\
U_{1,1}^{k-1} & \times & \bullet\bullet\bullet\bullet\bullet\bullet \\
 & \times & \bullet\bullet\bullet\bullet\bullet\bullet \\
\hline
0 & \otimes & \circ\circ\circ\circ\circ\circ \\
\hline
 & \times & \bullet\bullet\bullet\bullet\bullet\bullet \\
 & \times & \bullet\bullet\bullet\bullet\bullet\bullet \\
0 & \times & \bullet\bullet\bullet\bullet\bullet\bullet \\
 & \times & \bullet\bullet\bullet\bullet\bullet\bullet \\
 & \times & n\bullet\bullet\bullet\bullet\bullet\bullet \\
 & \times & \bullet\bullet\bullet\bullet\bullet\bullet
\end{array}\right)
$$

Fig. 4. Read and wrote data at the k^{th} step of the Gauss-Jordan method.

be transformed to such a form. Shift techniques are utilized to accelerate the convergence of this method in practical implementation. Let λ_k be the Wilkinson shift of the k^{th} iteration [7]. The k^{th} iteration of the QR method computes (with Q^k satisfying $A^k = Q^k R^k$ where R^k is triangular):

$$A^k = Q^{k-1}(A^{k-1} - \lambda_k I)Q^{k-1} + \lambda_k I.$$

The computation of each iteration is mainly composed of $(n-1)$ steps in sequence. Figure 5 shows the subset of elements of the virtual geometry concerned at a given step. Some multiplication of two sub-rows by a 2×2 Housolder matrix h^k and the multiplication of two sub-columns by the same matrix are the important computations. Only four elements at the intersection of these two rows and two columns are multiplied twice per step by h^k, once on the left side and once on the right side. The $\times$ or $\circ$ elements can be computed in parallel but the $\otimes$ elements can not. The control flow is unique on data parallel computing model, then we have to sequentialize these operations. A in-depth study of this method shows that the 24 $n^2 + \mathcal{O}(n)$ operations of a QR iteration generate 40 $n + \mathcal{O}(1)$ data parallel operations [16, 17]. We always have $2n+6$ elements of the virtual geometry concerned by these operations at each step of a QR iterations. Then, only one n^{th} of the virtual processors are concerned, at least.

If we use a one-dimensional geometry with one column (resp. row) of the matrix on each virtual processor, cf Figure 6, we need $O(k)$ data parallel operations at each step, instead of $\mathcal{O}(1)$, because on each column (resp. row) the operations are sequentialized. Then, the global complexity will be of the same order that with a two dimensional virtual geometry.

There exist some optimized mapping strategies and parallel algorithms adapted from systolic ones to implement the QR method; using a one dimensional virtual geometry with $n+2$ virtual processors. We would have $O(n)$ data parallel operations for one iteration of the QR method with this algorithm. Nevertheless, the generated mapping of such methods is very different from those used for data parallel linear algebra and, then, create a lot of expensive re-mapping operations before and after each QR computation. We also remark that we do not respect

$$
\left(\begin{array}{c|c|c}
H^{k-1}_{1,1} & \begin{pmatrix} \times & \times \\ \times & \times \\ \times & \times \end{pmatrix} h^k & H^{k-1}_{1,3} \\
\hline
h^k \begin{pmatrix} & \circ & \circ \\ & & \circ \end{pmatrix} & h^k \begin{pmatrix} \otimes & \otimes \\ \otimes & \otimes \end{pmatrix} h^k & h^k \begin{pmatrix} \circ & \circ & \circ \\ \circ & \circ & \circ \end{pmatrix} \\
\hline
0 & \begin{pmatrix} \times \\ \\ \\ \end{pmatrix} h^k & H^{k-1}_{3,3}
\end{array}\right)
$$

Fig. 5. Read and wrote data at the step k of one iteration of the QR method; $m = 8$ and $k = 4$.

$$
\left(\begin{array}{cc|cc|cccc}
\bullet & \bullet & Col. & Col. & \bullet & \bullet & \bullet & \bullet \\
 & \bullet & k & k+1 & \bullet & \bullet & \bullet & \bullet
\end{array}\right)
$$

Fig. 6. One dimensional virtual geometry QR algorithm.

the hypothesis of the section 2; in this case. That is why we do not develop this algorithm further.

We can conclude that this method is not well-adapted to data parallel programming model.

Discussions These common linear algebra algorithms illustrate some of the data parallel programming problems for scientific computing. Some of them, as matrix addition, exploited the possibility to compute the same operation on a large number of data. If we have some matrix vector multiplications, we have to face the alignment problem between vectors mapped on the virtual geometry. This often generates important data exchanges during the computation; as for the computation of AXPX. The direct studied factorization methods would often be called **semi-data parallel** and some methods, as the back substitution and the QR method would be qualified as **non data parallel**.

Then, how can we determinate if an algorithm is a `good` data parallel one. Is the efficiency then guaranteed? Can we compare such algorithms? To be able to answer these questions, we will introduce, under some hypothesis, some parameters which will allow us to evaluate these algorithms. Nevertheless, we do not introduce a new theoretical model but just would like to help users to write efficient data parallel programs.

We set two hypothesis:

Hypothesis I We limit our study to data parallel algorithms using a unique virtual geometry into which we map and align data.

A canonic data mapping is adopted such as matrices are stored on the two dimensional virtual geometry, the vectors are mapped and aligned redundantly into rows or columns of the virtual geometry. Scalars are mapped on all virtual processors.

Hypothesis II The complexity of prefix operations such as the sum of elements of one row (resp. one column) of a data parallel variable mapped on the virtual geometry, of size n, with a result per row (resp. per column) is the same as if we have the results on each virtual processor of the row (resp. of the column): i.e. $log_2(n)$.

Thus, we define the following parameters, defined for a data parallel algorithm and a virtual geometry. Let:

p be the number of virtual processor of the virtual geometry,
α be the number of virtual processors concerned by a data parallel operation,
cx be the number of data parallel operations (other than communications).

We also define some parameters at the physical machine level. Let

$\mathcal{P}$ be the number of physical processors of the target parallel machine ($p \gg \mathcal{P}$ by hypothesis),
vpr be the *virtual processor ratio*, such as $vpr = \frac{p}{\mathcal{P}}$.
$\aleph$ be the number of running physical processors for a given data parallel operation,
CX be the number of parallel physical operations done on the target parallel machine to compute the cx data parallel operations; in general we have $CX \geq vpr\ cx$.

The computation ratio for a data operation ops is defined as follows:

$$\varphi_{ops} = \frac{\alpha_{ops}}{p} \leq 1,$$

Then, the average computation ratio is:

$$\overline{\varphi} = \frac{1}{cx} \sum_{ops=1}^{ops=cx} \varphi_{ops} \leq 1,$$

which only depends of the data parallel algorithm.

The maximum number of data involved in a data parallel operation of a given data parallel algorithm will be called the degree of parallelism. let $\mathcal{D}$ be this parameter. Then we remark that

$$\mathcal{D} = \max_{ops=1,cx} \alpha_{ops}$$

For the matrix addition example we have: $\overline{\phi} = \overline{\varphi} = \phi = \varphi = 1$, $cx = 1$, $\mathcal{D} = p$.

Table 1 gives values of the these parameters for the algorithms studied above. A data parallel triadic operation $(+, *)$ is count for one data parallel operation.

	data parallel		semi data parallel		non data parallel	
	A + λ A	Av + v	Gaussian Elimination	Gauss Jordan	Back substitution	QR method
cx	1 $(+, \times)$	$2 + log_2(n)$	2n -2	2n -2	2n -2	40 n
$\overline{\varphi}$	1	1	$\simeq \frac{1}{6}$	$\simeq \frac{1}{4}$	$\simeq \frac{3}{4n^3} + \frac{1}{4n^2}$	$\simeq \frac{3}{20n^3} + \frac{1}{20n^2}$
$\mathcal{D}$	n^2	n^2	$(n-1)^2$	$(n-1)^2$	$(n-1)$	$2(n-k+2)$
p	n^2	n^2	n^2	n^2	n^2	n^2

Table 1. Numerical results for the studied data parallel algorithms.

We remark that we have $\overline{\varphi}$ and $\frac{p}{\mathcal{D}}$ close to 1; while the semi data parallel ones have $\overline{\varphi}$ equal to a constant range between 1 and $\frac{1}{k}$, k = 2 to a small finite constant (but the ratio $\frac{p}{\mathcal{D}}$ is still always close to 1). Let us remark, that the non data parallel ones have $\overline{\varphi}$ in order of $O(\frac{1}{n^k})$, $k = 2$ for the studied examples. The ratio $\frac{p}{\mathcal{D}}$ is, then, of order $O(n)$.

4 First Data Parallel Programming Rules

The best data parallel algorithm to solve a scientific problem is the one with the smallest cx (the numerical aspect is supposed correct). For a fixed virtual geometry, we can have several possible mappings, alignments or numerical methods (for example the difference between Gaussian elimination and Gauss-Jordan is only the number of virtual processors selected for the data parallel operations). Thus, we often choose the data parallel algorithm which maximize $\overline{\varphi}$. In general, it generates less communications in the remaining part of the algorithm.

Experimentation on data parallel machines already focus on the importance of the communication problem. General communications between processors can be very time consuming. Nevertheless, it is very difficult to evaluate the communication cost at the algorithm level. Communications depend a lot of the target network, the data mapping, the virtual processor ratio *vpr* (and, then, of the number of physical processor and the data size). This problem especially occurs when communication optimization techniques increase cx and modify $\overline{\varphi}$. We choose to not directly consider communications at the programming model level, even if we will prefer reduction operations to general communications. We first compare and analyze data parallel algorithms without any communication cost evaluations.

However, we have to consider these communication problems as soon as we know what the target machine is. Thus, we can have some `good` data parallel algorithm which will not be efficient at all; because it will generate very long communications for a fixed target machine.

During this study, we respect the hypothesis of section 2; i.e. the choice to minimize the global number of data parallel operations, independent of the execution time on a target machine. Without this hypothesis, a solution is to directly distribute the operations onto parallel tasks (as many as the number of physical processors) and to optimize the computation on each physical processor; and using message passing programming model. One of the present problem is to be able to write data parallel programs from data parallel algorithms and to execute them on machines with a few physical processors. The criteria for developing a data parallel algorithm and to evaluate its complexity are often opposite to these for writing efficient code for a target machine with a few processors. For example, the Gauss-Jordan method has a smaller data parallel complexity that the Gaussian elimination one to solve a linear system, but on a parallel machine with a few processors the Gauss method will be faster. A smart compiler would generate less operations on each physical processor for the Gauss method. This comparison result will change with respect to the number of physical processors. The problem of evaluating and comparing data parallel algorithms is very different from that of compiling data parallel programs to parallel machines with a moderate number of physical processors. If we want to obtain the faster executable code on a parallel machine with a few processors, the best is to use message passing programming model. The problem is that the more data parallel algorithm will not be always the faster on many parallel machines. It will depend of the number of virtual processors and compilers. Then, the choice of an efficient data parallel algorithm will depend of many criteria. The work present in this paper is the first part of a process to develop efficient parallel codes. Nevertheless, the knowledge obtained with such analyses is necessary to choose a good parallel method to develop data parallel program for a fixed target computer.

Very Large Sparse Matrix Case When data structures are irregular, for example if we manipulate sparse matrices, the problem is very different. How it is very different? If we apply the above methodology, we map the sparse matrices after compression, i.e. we map only non zero elements, on a two dimensional virtual geometry. Vectors are aligned on the larger dimension (i.e. on the non compressed dimension). If we have to compute a sparse matrix vector multiplication, all the data parallel operations have a good computation ratio and the data parallel complexity is $cx = 1 + log_2 C$, cf [20]; where C is the number of non zero elements by row and column. Nevertheless, the general communication number is very large. We have to optimize the ones with available tools such as communication compilers or communication vectorizer [2, 13, 8], for a fixed target machine. This choice to parallelize the operations, and, then, optimize the communications is again set.

5 Data Parallel Sparse Matrix Vector Multiplication.

Let A be a sparse matrix of size n with C non zero elements on each row and/or column; let $(a_{i,j})(i = 1, n$ and $j = 1, n)$ represent the j^{th} element of the i^{th} row, zero or not. Let x, w be two vectors of size n, the goal is the computation of

$w = Ax$ followed by $A(Ax + x) + x$ or $A^m(x)$ computation.

$$A = \begin{pmatrix} & & 13 & & & & 17 & 18 \\ & & & 24 & & 26 & & \\ 31 & 32 & & & 35 & & & \\ 41 & & & & 45 & 46 & & \\ & & & 54 & & 56 & & \\ & 62 & & & 65 & & 67 & \\ 71 & & & 74 & & & & 78 \\ & 82 & 83 & & & & 87 & \end{pmatrix}, \; Acc = \begin{bmatrix} 13 & 17 & 18 \\ 24 & 26 & - \\ 31 & 32 & 35 \\ 41 & 45 & 46 \\ 54 & 56 & - \\ 62 & 65 & 67 \\ 71 & 74 & 78 \\ 82 & 83 & 87 \end{bmatrix}$$

$$Acr = \begin{bmatrix} 31 & 32 & 13 & 24 & 35 & 26 & 17 & 18 \\ 41 & 62 & 83 & 54 & 45 & 46 & 67 & 78 \\ 71 & 82 & - & 74 & 65 & 56 & 87 & - \end{bmatrix}$$

$$Ai = \begin{bmatrix} 3 & 3 & 1 & 2 & 3 & 2 & 1 & 1 \\ 4 & 6 & 8 & 5 & 4 & 4 & 6 & 7 \\ 7 & 8 & - & 7 & 6 & 5 & 8 & - \end{bmatrix}, \qquad Aj = \begin{bmatrix} 3 & 7 & 8 \\ 4 & 6 & - \\ 1 & 2 & 5 \\ 1 & 5 & 6 \\ 4 & 6 & - \\ 2 & 5 & 7 \\ 1 & 4 & 8 \\ 2 & 3 & 7 \end{bmatrix}$$

$$Jc = \begin{bmatrix} 1 & 2 & 1 & 1 & 3 & 2 & 2 & 3 \\ 1 & 1 & 2 & 1 & 2 & 3 & 3 & 3 \\ 1 & 1 & - & 2 & 2 & 2 & 3 & - \end{bmatrix}, \qquad Ic = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & - \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 2 & 3 & - \\ 2 & 3 & 2 \\ 3 & 3 & 2 \\ 3 & 2 & 3 \end{bmatrix}$$

Fig. 7. Data Parallel Mapping of the Sparse Matrix A; SGP format using a column compression (Acr, Ai and Jc) or a row compression (Acc, Aj and Ic); $n = 8$ and $C = 3$.

We suppose that we are able to implicitly or explicitly organize the virtual processors as a $n \times C$ or $C \times n$ geometry. We map the sparse matrix and the vector on this geometry. We introduced the *Sparse General Pattern* (SGP), see [20, 21] and Figure 7: i.e. we compressed each column or row, i.e. we remove zeros from each column or row, before mapping them to the C processors of the small dimension of the geometry. When the number of non zero elements by rows or columns is not equal, there exist another pattern, see the S^3 pattern in [20]. The SGP pattern also use a data parallel variable to store the index of the compressed dimension, see arrays Ai or Aj of the Figure 7. These two data parallel variables determine a given sparse matrix. It is a data parallel adaptation of the Ellpack format [25] used on a sequential machines. Nevertheless, when we manipulate sparse matrices for scientific computing, we have to often change from a row compressed pattern to a column compressed one; on the same virtual geometry. Then, for each non zero element $a_{i,j}$, $\forall\ (i,j) \in [1,n]^2$, we also have to know the location of each non zero element corresponding to a column compressed pattern (let (i_c, j) be this location) or corresponding to a row compressed pattern (let (i, j_c) be this location). We have to map on each virtual processor the quintuple $(a_{i,j}, i, j, i_c, j_c)$. That is why we add another data parallel variable to the SGP pattern (see Figure 7, Jc for column compressed pattern and Ic for the other compressed pattern) which allow to know on each virtual processor all these informations. We note that it would generate many general communications to re-calculate this information if we do not know Ic or Jc on each virtual processor of the geometry.

When we change the compression orientation of the sparse matrix during data parallel computation, using SGP data parallel pattern, we keep only a allocated two-dimensional virtual geometry $C \times N$ or $N \times C$. One of the mappings have to be seen with a $\frac{\pi}{2}$ permutation.

Each element of the vector x is redundantly mapped along the larger dimension of the virtual geometry to compute $w = Ax$, see Figure 8, i.e. all the C virtual processors where a compressed column is mapped have the expected element for the data parallel multiplication operation. Then, each non zero element of the matrix is mapped on only one virtual processor and each element of the vector is mapped on C virtual processeurs.

Our version of the data parallel sparse matrix vector multiplication is based on the idea of computing all the elementary multiplications with only one data parallel operation and, then, to sum the partial results on $O(log_2(C))$ data parallel addition; along the C virtual processors of the smaller dimension of the geometry. Thus, we compute $a_{i,j}\ x_j = t_{i,j}$ in just one data parallel operation, only for non zero elements. Then, we have to sum the elements $t_{i,j}$, such as $\sum_{j=1}^{j=n} t_{i,j} = w_i$, for only non zero elements; done with a $log_2(n)$ data parallel reduction with addition operation.

The first mapping, see G_2 in Figure 8, is the best one for the data parallel multiplications and the second, see G_3 on Figure 8, is well-adapted for the reduction with addition operation. Then, we have to re-map the data between the multiplication and the reduction data parallel operations. We have to re-map the data from a column compressed mapping to a row compressed one. We have to send data from virtual processor (i_c, j) of the geometry to (j_c, i), $\forall (i_c, j)$ and $(j_c, i) \in [1, C] \times [1, n]$,

After the reduction operation, $w = Ax$ is mapped on each row of the virtual geometry, see $G4$ in Figure 8. We observe that the results are mapped as the initial data. The result vector is well mapped and aligned on the sparse case to directly compute $Ax + x$, $A(Ax + x) + x$ or $A^m(x)$ without any re-mapping process.

$$A = \begin{pmatrix} & & 13 & & & & 17 & 18 \\ & & & 24 & & 26 & & \\ 31 & 32 & & & 35 & & & \\ 41 & & & & 45 & 46 & & \\ & & & 54 & & 56 & & \\ & 62 & & & 65 & & 67 & \\ 71 & & & 74 & & & & 78 \\ & 82 & 83 & & & & 87 & \end{pmatrix}, \quad x = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \end{pmatrix},$$

$$X = \left(\begin{array}{c|c|c|c|c|c|c|c} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ \hline x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ \hline x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \end{array}\right), \quad Acr = \left[\begin{array}{c|c|c|c|c|c|c|c} 31 & 32 & 13 & 24 & 35 & 26 & 17 & 18 \\ \hline 41 & 62 & 83 & 54 & 45 & 46 & 67 & 78 \\ \hline 71 & 82 & - & 74 & 65 & 56 & 87 & - \end{array}\right],$$

$$Ai = \left[\begin{array}{c|c|c|c|c|c|c|c} 3 & 3 & 1 & 2 & 3 & 2 & 1 & 1 \\ \hline 4 & 6 & 8 & 5 & 4 & 4 & 6 & 7 \\ \hline 7 & 8 & - & 7 & 6 & 5 & 8 & - \end{array}\right], \quad Jc = \left[\begin{array}{c|c|c|c|c|c|c|c} 1 & 2 & 1 & 1 & 3 & 2 & 2 & 3 \\ \hline 1 & 1 & 2 & 1 & 2 & 3 & 3 & 3 \\ \hline 1 & 1 & - & 2 & 2 & 2 & 3 & - \end{array}\right],$$

$$G_1 = \left[\begin{array}{c|c|c|c|c|c|c|c} 31 & 32 & 13 & 24 & 35 & 26 & 17 & 18 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ 1 & 2 & 1 & 1 & 3 & 2 & 2 & 3 \\ 3 & 3 & 1 & 2 & 3 & 2 & 1 & 1 \\ \hline 41 & 62 & 83 & 54 & 45 & 46 & 67 & 78 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ 1 & 1 & 2 & 1 & 2 & 3 & 3 & 3 \\ 4 & 6 & 8 & 5 & 4 & 4 & 6 & 7 \\ \hline 71 & 82 & - & 74 & 65 & 56 & 87 & - \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ 1 & 1 & - & 2 & 2 & 2 & 3 & - \\ 7 & 8 & - & 7 & 6 & 5 & 8 & - \end{array}\right], \quad G_2 = \left[\begin{array}{c|c|c|c|c|c|c|c} t_{1,1} & t_{1,2} & t_{1,3} & t_{1,4} & t_{1,5} & t_{1,6} & t_{1,7} & t_{1,8} \\ 1 & 2 & 1 & 1 & 3 & 2 & 2 & 3 \\ 3 & 3 & 1 & 2 & 3 & 2 & 1 & 1 \\ \hline t_{2,1} & t_{2,2} & t_{2,3} & t_{2,4} & t_{2,5} & t_{2,6} & t_{2,7} & t_{2,8} \\ 1 & 1 & 2 & 1 & 2 & 3 & 3 & 3 \\ 4 & 6 & 8 & 5 & 4 & 4 & 6 & 7 \\ \hline t_{3,1} & t_{3,2} & - & t_{3,4} & t_{3,5} & t_{3,6} & t_{3,7} & - \\ 1 & 1 & - & 2 & 2 & 2 & 3 & - \\ 7 & 8 & - & 7 & 6 & 5 & 8 & - \end{array}\right],$$

$$G_3 = \left[\begin{array}{c|c|c|c|c|c|c|c} t_{1,3} & t_{1,4} & t_{1,1} & t_{2,1} & t_{2,4} & t_{2,2} & t_{3,1} & t_{3,2} \\ \hline t_{1,7} & t_{1,6} & t_{1,2} & t_{2,5} & t_{3,6} & t_{3,5} & t_{3,4} & t_{2,3} \\ \hline t_{1,8} & - & t_{1,5} & t_{2,6} & - & t_{2,7} & t_{2,8} & t_{3,7} \end{array}\right], G_4 = \left[\begin{array}{c|c|c|c|c|c|c|c} w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 \\ \hline w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 \\ \hline w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 \end{array}\right].$$

Fig. 8. Data Parallel Sparse Matrix Vector Multiplication Ax, $n = 8$ and $C = 3$.

The explicit data parallel algorithm is:

Geometry: two dimensional virtual geometry. Let G be the virtual grid: 1 to C (first dimension) by 1 to n (second dimension).

Initial Mapping: On each virtual processor $pv(i_c, j); i_c = 1, C; j = 1, n$, mapped in the grid G:

- x_j, the j^{th} element of vector x . Let !!X be the corresponding data parallel variable .

– i, such as $(a_{i,j})$ is mapped to this virtual processor (Ai in Figure 7).
– The i_c^{th} non zero element of the compressed column j of the matrix A (i.e. $a_{i,j}$). Let !!A be the corresponding data parallel variable (Acr in Figure 7).
– j_c, such as the j_c^{th} non zero element of the compressed row is mapped to the virtual processor (Jc on Figure 7).

Data Parallel Algorithm : On each processor virtual $pv(i_c, j); i_c = 1, C; j = 1, n$ do:

1. !!W = !!X * !!A
2. *Send* !!W to processeurs $pv(i, j_c)$
3. Reduction with addition along the small dimension of all received messages on these processors $pv(i_c, j), i_c = 1, C; \forall j$. Results are redundantly mapped on virtual processeurs $pv(i_c, j)$, $\forall i_c = 1, C$.

Final mapping: On each virtual processor $pv(i_c, j); i_c = 1, C; j = 1, n$, mapped on the virtual grid G:

– !!X and !!A as initially mapped,
– the j^{th} element of the vector Ax.

The data parallel complexity is $1 + log_2(C)$ and a *one-to-one* general communication. There also exists a version with *get* operation, see [6], where the rows are first compressed and the communications are *many-to-one*.

The general communications are from virtual processors $pv(i_c, j)$ toward virtual processors $pv(j_c, i)$. Then, if we remark that i_c and j_c are less larger or equal to C, and small compared with n, we can observe that the distance are of the order of $|j - i|$.

6 Preconditioned Conjugate Gradient Method

The conjugate gradient method is often used in scientific computing to solve very large applications. Like many iterative methods to solve sparse linear systems, this method only reads sparse matrices to compute one or several matrix vector multiplications; at each iteration. The matrices are not changed during the iterations and, as we will show, the computation scheme is comparable to those described above (i.e. $A(Ax + x) + x$ for the conjugate gradient part and A^m for the polynomial precondionning part).

We want to solve the following linear algebra problem: $Ay = b$, where A is a very large symmetric positive definite sparse matrix of order n, b a given vector and y the solution vector.

The conjugate gradient method with a efficient preconditioner (PCG) is a classical method to solve these very large linear systems [5]. The convergence is obtained with less that n iterations, then it is necessary to accelerate the convergence. This is the goal of the preconditioning part of the PCG method.

A method is to find a matrix M which approximates A, with the same sparse structure and such that the linear system $Mz = r$ is easy to solve; M is called the preconditioning matrix. The convergence speed really depends of the choice of M.

Polynomial Preconditioner

Several possibilities are available to choose M or M^{-1}, solving $Mz = r$ without greatly increasing the number of operations.

In the case of polynomial preconditioner, we have to find a A polynomial which will approximated M^{-1}; see [12, 24] for details. The simplest of all candidate polynomials to approximate A^{-1} is the one obtained with the truncated Newman series:

$s(A) = I + B + B^2 + \cdots + B^k$ with $B = I - A$ and $||B|| < 1$.

In the following polynomial preconditionned GCGHS (Generalized Conjugate Gradient of Hestenes and Stiefel) algorithm, we use the matrix S which is equal to the polynomial s, such that: $S = s(A)$. Then we have the algorithm:

Step 1: choose an initial guess y_0, then do

$$r_0 = S(b - Ay_0) \tag{1}$$
$$\rho_0 = (r_0, r_0) \tag{2}$$
$$v_0 = r_0$$

Step 2: do for $i = 0, 1, \ldots$ until convergence

$$w_i = SAv_i \tag{3}$$
$$\alpha_i = \rho_i/(w_i, v_i) \tag{4}$$
$$y_{i+1} = y_i + \alpha_i v_i \tag{5}$$
$$r_{i+1} = r_i - \alpha_i w_i \tag{6}$$
$$\rho_{i+1} = (r_{i+1}, r_{i+1}) \tag{7}$$
$$\beta_i = \rho_{i+1}/\rho_i \tag{8}$$
$$v_{i+1} = r_{i+1} + \beta_i v_i \tag{9}$$

Capital letters represent matrices, small letters represent vectors and Greek letters represent scalars.

Data mapping and data parallel algorithm

We use the sparse matrix vector multiplication studied above. Thus, we have a two dimensional virtual geometry $C \times n$. The matrix A and the vectors are mapped as described for matrix vector multiplication.

Thus, the mapping is:

- Scalar variables are redundantly mapped on each virtual processor P_{i_g,j_g} $\forall i_g, \forall j_g$.
- Vectors, which are of size n, are redundantly mapped along each row of the virtual geometry. Then, the same data is mapped on all processors P_{i_g,j_g} $\forall i_g$.
- Matrices are mapped as described for the matrix vector multiplication. Each non zero coefficient is mapped on one virtual processor.

We directly compute $s(A)x$ in each iteration, where x is a vector; we never explicitly compute the A polynomial. Thus, we have a few matrix vector multiplications to compute at each iteration (as many as the degree of the polynomial). The computation involving row-mapped vectors are redundantly done on each row of virtual processors; we optimize spread operations like this because the results of one iteration are on each row of the virtual geometry at the beginning of the next iteration. If we consider two iterations, we observe that we have computations just like those studied in above sections: $A(Ax + f(x)) + g(x)$, where the function f and g are just linear combinations of vectors including x) ; see [19, 1] for details.

7 Conclusion

The data parallel programming model is a general paradigm for scientific computing, and especially well-adapted for some important linear algebra applications, denses or sparses. We proposed some criteria to compare and evaluate data parallel algorithms, without communication considerations. When we try to integrate other criteria from communications, compilers, networks or I/O problems, for examples, we are not able to compare data parallel algorithms on the general case. For many iterative method in linear algebra, such as the Lanczos method or the Arnoldi and GMRES methods [11, 9, 10], we present algorithms and data structures to obtain a `good` data parallel program. The programming rules presented can be generalized to develop solutions to many industrial problems. Nevertheless, at this point, we have to adapt the generated data parallel programs to target machines and choose a well-adapted language and compiler.

The choice of a efficient data parallel algorithm, without considering the number of physical processors, is still very difficult for the available parallel computers. We can choose the more data parallel algorithm but it may not be efficient on these machines.

Although current trends indicate that the evolution of parallel architectures is not toward massively parallel machines, the future can be different.

Acknowledgment

The authors wish to thank Sanjay Rajopadhye for many helpful comments.

References

1. C. Tong, *The preconditioned conjugate gradient method on the connection machine*, in Scientific Applications of the Connection Machine, World Scientific, 1989.
2. D. Dahl, *Mapping and compiled communications on the connection machine system*, in Proceeding of the Fifth Distributed Memory Computing Conference, 1990.
3. F. Chatelin, *Valeurs propres de matrices*, Masson, 1989.
4. G. Golub and C. Van Loan, *Matrix Computation*, North Oxford Academic Oxford, second ed., 1989.

5. H. Golub and G. Meurant, *Résolution numérique des grands systèmes linéaires*, Eyrolles, 1983.
6. J. Saltz, S. Petiton, H. Berryman, and A. Rifkin, *Performance effects of irregular communications patterns on massively parallel multiprocessors*, Journal of Parallel and Distributed Computing, 8 (1991).
7. J. Wilkinson, *The Algebric Eigenvalue Problem*, Oxford University Press, 65.
8. J. Zdenek, *Data Parallel Finite Element Techniques for Large-Scale Computational Fuild Dynamics*, PhD thesis, Stanford University, 1992.
9. N. Emad, *Data parallel lanczos and padé-rayleigh-ritz on the cm-5*, in Science on the Connection Machine System, J.-M. A. et al., ed., Thinking Machine Coorp., 1995.
10. ———, *The padé-rayleigh-ritz method for solving large symetric eigenproblem*, Journal of Numerical Analysis, 11 (1996).
11. N. Emad and S. Petiton, *Numerical behavior of iterative arnoldi's method for sparse eigenproblems on massively parallel architectures*, in Iterative Methods in Linear Algebra, North Holland, 1992.
12. S. Asby, *Polynomial Preconditioning for Conjugate Gradient Methods*, PhD thesis, University of Illinois at Urbana Champaign, 1987.
13. S. Hammond, *Mapping unstructured grid computation to massively parallel computers*, Tech. Report 14, RIACS/NASA Ames, 1992.
14. S. L. Johnsson and K. Mathur, *Data structures and algorithms for the finite element method on a data parallel supercomputer*, International Journal for Numerical Methods in Ingineering, 29 (1990).
15. S. Petiton, *Du Développement de Logiciels Numériques en Environnements Parallèles*, PhD thesis, University P. and M. Curie, Paris VI, 1988.
16. ———, *Parallel qr algorithm for iterative subspace methods on the connection machine (cm2)*, in Parallel Processing for Scientific Computing, J. Dongarra, P. Messina, D. Sorensen, and R. Voigt, eds., SIAM, 1990.
17. ———, *Parallel subspace method for non-hermitian eigenproblems on the connection machine (cm2)*, Applied Numerical Mathematics, 9 (1992).
18. S. Petiton and C. Guillon, *Machine à réseau d'interconnexion pour l'analyse numérique*, Tech. Report 197, DRET (SINTRA/THOMSON-CSF), 1986.
19. S. Petiton and C. Weill-Duflos, *Very sparse preconditioned conjugate gradient on massively parallel architectures*, in Proceeding of 13th World Congress on Computation and Applied Mathematics, 1991.
20. S. Petiton and G. Edjlali, *Data parallel structures and algorithms for sparse matrix computation*, in Advances in Parallel Computing, North Holland, 1993.
21. S. Petiton, Y. Saad, K. Wu, and W. Ferng, *Basic sparse matrix computation on the cm-5*, International Journal of Modern Physics C, 4 (1993).
22. T. dekker and W. Hoffman, *Rehabilitation of the gauss-jordan algorithm*, tech. report, Instituut Universiteit van Amsterdam, 1986.
23. W.D. Hillis and G.L. Steele, *Data parallel algorithms*, Communication of ACM, 29 (1986).
24. Y. Saad, *Practical use of polynomial preconditionings for the conjugate gradient method*, SIAM J. SCI. STAT. COMP., 6 (1985).
25. ———, *Sparskit: a basic tool kit for sparse matrix computations*, Tech. Report 20, RIACS, NASA Ames Research Center, 90.