Solving the Least Squares Problem for Sparse Matrices on the AP1000

R. Doallo J. Touriño
F.F. Rivera[†]
e-mail: {esdoallo,juan}@udc.es, elfran@usc.es
Dep. of Electrónica y Sistemas. University of La Coruña
[†]Dep. of Electrónica y Computación. University of Santiago de Compostela
SPAIN

Abstract

Least Squares is a standard problem in linear algebra. It can be solved by adapting the algorithm that carries out the QR decomposition. This approach is adequate due to the good numerical stability of the QR factorization. The algorithm has been developed for Fujitsu's AP1000 computer and it takes into account the sparse nature of the matrices to get memory savings and decrease the computation times. We will describe the parallel algorithm which solves this problem and show an evaluation of its efficiency using the Harwell-Boeing matrix collection.

1 Introduction

The least squares problem consists of calculating a vector x of length B that minimizes $\|Mx - z\|_2$, where M is a matrix $A * B$ ($A \geq B$) and z is a vector of length A. If the rank of M is maximum, the problem is going to have a unique solution (x_{LS}). In any other case, it is going to have an infinite number of solutions x_{SOL}, one of them with minimum norm and which we will also denote as x_{LS}: $x_{LS} = x_{SOL}/\|x_{SOL}\|_2$ *is minimum.*

QR factorization is a direct method in matrix algebra based on orthogonalization. It involves the decomposition of a matrix M into the product of two matrices: $Q{*}R$, where Q is an orthogonal matrix ($Q^T = Q^{-1}$) and R is upper triangular. The least squares problem is equivalent to solving the upper triangular system: $R\Pi^T x = Q^T z$ where Π is a permutation due to the pivoting we carry out in the algorithm. We will get the unique solution to the least squares problem when $rank(M) = B$. In the case where $rank(M) < B$, we get one of the infinite solutions: the one called basic solution, which will have a maximum of as many nonzero elements as the rank of *M*, and which in general will not coincide with the minimum norm solution x_{LS}.

There are several methods for finding this decomposition: Householder reflections, Givens rotations and Modified Gram-Schmidt [1, chap.3]. A hybrid algorithm can be found in [2]. Several parallel orthogonal factorization algorithms have been designed for various machines. We cite just a few: [3] for the *NCUBE/10*, [4] for the *AP1000*, all of them for dense matrices; and [5] (*CM-2*), [6] (*Cray X-MP/Y-MP*, vector computers) for sparse matrices.

Modified Gram-Schmidt is the algorithm we will implement. It is a rearrangement of the Classical Gram-Schmidt algorithm with better numerical properties. This method obtains matrices Q and R of dimensions $A * B$ and $B * B$, respectively.

2 Data storage and data distribution

As the problem is sparse, we will use a dynamic structure for the storage of the data in the cells to support the changes in the disposition of the nonzero elements: doubly linked lists, each one of which represents a column of the matrix. Each item of the list stores the row index, the matrix element and two pointers (one pointing to the previous item and another to the next). These lists are arranged in growing order of the row index. The parallel algorithm uses several routines for insertion and deletion in the lists in order to optimize their handling. In [7] we analyze another structure for the storage based on packed vectors [8, chap.2]. We don't need a two-dimensional doubly linked list as in a LU decomposition for sparse matrices because we only require efficient access by columns. This implies large memory savings and reduces the computation times.

With respect to data distribution, we have chosen a cyclic distribution (per rows and per columns). We are going to assume a general configuration of $m * n$ processing elements or cells. Each cell is going to be identified by coordinates *(idx,idy)*, with $0 \leq idx < n$ and $0 \leq idy < m$. So, element *(I,J)* of the matrix M is located in the cell *(idx,idy)=(J mod n,I mod m)*. After executing the algorithm, the original matrix M will be our matrix Q (in-place algorithm). We can get the local indices *(i,j)* in each cell from the global indices *(I,J)*: $i=\lfloor \frac{I}{m} \rfloor$, $j= \lfloor \frac{J}{n} \rfloor$ and the dimensions $a * b$ of the local matrices: $a = \lfloor \frac{A}{m} \rfloor + idy < A \; mod \; m$; $b= \lfloor \frac{B}{n} \rfloor + idx < B \; mod \; n$. If the logical expressions $idy<A \; mod \; m$, $idx<B \; mod \; n$

are fulfilled, they will return a value of one; otherwise, they will return a value of zero.

This cyclic distribution has been selected for two reasons: data and load balancing. The first reason arises from the nature of sparse matrices. If we work with matrices in which the nonzero elements are located randomly, cyclic distribution is most appropiate to provide good data balancing. The second reason is based on the characteristics of our problem, a problem of reducing the index space. The algorithm is made up of as many steps as columns in the matrix M (unless the rank is smaller). Let us assume that we are in step k. Then, as we will later see, elements (α,μ) of matrix Q and elements (k, μ) of matrix R, with $0 \leq \alpha < A$; $k \leq \mu < B$ will be updated. If we were to employ consecutive distribution of the data, as the algorithm was executed, a large number of the cells would be idle because the elements of the matrix that must be updated would be concentrated in only a few cells. This is prevented by using cyclic distribution without any need for increasing communications.

3 Implementation on the AP1000

The parallel algorithm has been programmed for Fujitsu's *AP1000* MIMD distributed-memory computer (2-D torus topology) and has been previously debugged using the software simulator CASIM. This algorithm has been generalized for any dimension of the torus and any dimension of matrix M, so that the execution for a single processor is equivalent to the sequential algorithm.

3.1 An overview of the program

We present the main procedure of the parallel algorithm for the cells:

cell_main()

```
Receiving_Data_from_Host();
rank=B;
Norms();
for (current_index=0;current_index<B;current_index++) {
  Pivot_Element();
  if (fabs(pivot)<EPSILON) {
    rank=current_index;
    break; }
  Column_Swap();
  Modified_Gram_Schmidt(); }
```

```
11    Back_Substitution();
12    Permutation();
13    Sending_Data_to_Host();
```

(1-2) The cells receive the submatrices corresponding to the matrix M according to a cyclic distribution; *rank* indicates the rank of this matrix. The maximum rank for M is B, as $A \geq B$.

(3) This procedure finds the square of the euclidean norms of the columns of matrix M and stores them in vector *norm*. Each cell obtains the local norms corresponding to the column segments (local lists) it contains. By means of reduction instruction *y_fsum* (sum by columns), the *norm* vector of a given cell will contain the norms of the global columns.

We go through the columns of the matrix and perform all the following actions:

(5-8) Obtain in variable *pivot* the pivot element, which is the maximum of the norms of the columns whose order is higher than or equal to *current_index* (iteration we are currently processing). The index of the global column containing the pivot element is called the *pivot_index*. Both values (*pivot* and *pivot_index*) will be contained in all the cells. If the pivot element is zero (*EPSILON* is the required precision), the rank of the matrix is given by the value *current_index* and the factorization ends. First, we obtain the local maximum of each cell (this value is the same for each cell column). The global maximum is obtained by means of the reduction instruction (*x_fmax*) which finds the maximum by cell rows.

(9) If the pivot is different from zero, we perform a swap of the column we are processing with the column of the pivot element in matrices Q,R and a swap of their corresponding norms.

(10) Once the pivoting has been carried out, we apply the Modified Gram-Schmidt algorithm updating the appropriate elements of matrices Q and R, as well as the corresponding norms.

(11-12) The upper triangular system $Rx = Q^T z$ is solved by means of a back-substitution. We also need to permute the elements of vector x to obtain the correct solution to the least squares problem.

In the following sections we describe the last four procedures in more detail.

(13) The local results of the cells are sent to the host in order to reconstruct the global results.

3.2 Pivoting

The *Column_Swap* procedure performs a column pivoting in order to consider those cases in which the rank of matrix M is not maximum and to provide numerical stability.

The cell column that contains the current column exchanges this column of matrices Q and R (as well as the corresponding norm) with the cell column that contains the pivot column. This can be observed in figure 1.

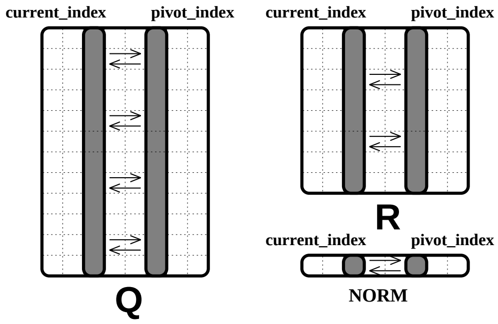


Figure 1: Column swap (pivoting)

We have to indicate the start memory address where this information is located and the size in order to send it to another cell. This implies that the information we send has to occupy consecutive memory positions, a situation that does not happen with a list. If we establish as many communications as components of the list for sending the data, this would imply a great temporal cost. In the *AP1000* computer, the communication time T can be modelled by an affine function so that for a message of N bytes, $T = t_0 + t_b * N\mu s$, where t_0 is a fixed overhead in establishing the communication and t_b is the transfer time per byte, $t_0 \gg t_b$. We find many experiments and measures of the communication times of the *AP1000* in [9]. Therefore, it would be interesting to send all the components of the list in a single message, thereby saving the fixed overhead in each one of the communications. In order to implement this solution, we use a special structure, a packed vector that acts as a buffer for sending the necessary information. As we show in figure 2, we now have our information in consecutive memory positions and we send the corresponding column of Q and R, as well as the square of the norm, in a single message.

3.3 Modified Gram-Schmidt algorithm

We obtain, in each iteration of this procedure, part of the row *current_index* of R (from the column *current_index* to the last one). All the elements of matrix Q from the column *current_index* are also updated, as well as the corresponding norms. This is shown in figure 3.

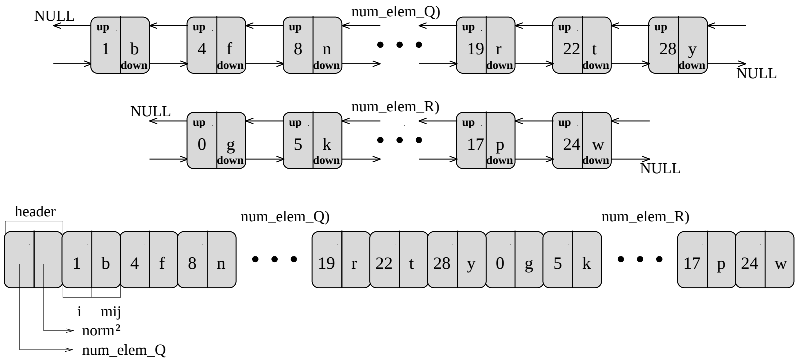


Figure 2: Buffer for sending local lists

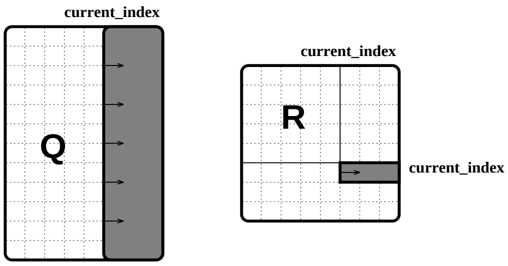


Figure 3: Process of updating matrices Q and R

First, we normalize the current column of matrix Q (which was the pivot column before the swapping) dividing it by its norm. Then, we store this normalized column in the packed vector *vcol*. An example of a $3*3$ mesh and a matrix of dimensions $9*9$ is shown in figure 4. The cells containing the normalized column broadcast it to all the cells on the X axis using the instruction *x_brd* (figure 4a).

For each column with a global index higher than *current_index*, we store in vector *vsum* (not packed) the local dot product of this column and the normalized current column (figure 4b). Finally, we obtain in *vsum* the global dot product for each column by means of the instruction *y_fsum* (figure 4c).

The elements of *vsum* will be the row *current_index* of matrix R, from the column *current_index* to the last one, as R is upper triangular. We don't need row access to matrix R in order to insert this new row because it is an insertion at the end of the corresponding lists of R, that is, column access.

Now, we update the elements q_{ij} of Q, $0 \leq i < A$ and $current\_index \leq j < B$. So, we

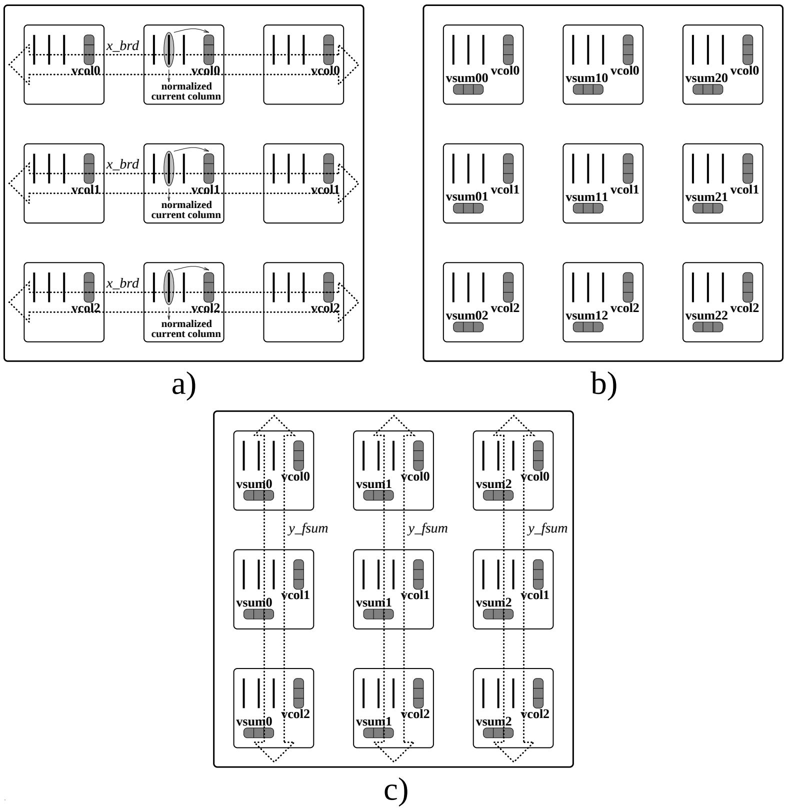


Figure 4: Using packed vectors in Modified Gram-Schmidt

overwrite q_{ij} with $q_{ij} - vcol_i * vsum_j$. When this operation is carried out, it may happen that $q_{ij} = 0$ and $vcol_i, vsum_j \neq 0$, so that an element of matrix Q which was initially null, now takes a nonzero value (fill-in); we therefore insert it in the corresponding local list. It may also happen that element q_{ij} takes a value of zero and consequently, we will have to eliminate it from the corresponding local list (the opposite phenomenon to the fill-in). This section of code is optimized because when a column of Q is updated, the corresponding local list suffers a single pass.

In this procedure, we will also obtain the product $Q^T z$, necessary to solve the least squares problem. Previously, vector z (we assume it is dense) has been cyclicly distributed in each cell column, so that the global component I of z will be replicated in the cell row with $idy{=}I \bmod m$. Remember that we obtain one column of matrix Q in each iteration of the algorithm. Therefore, we calculate in each iteration the local dot product of this column and vector z. Figure 5a shows this situation, where the current column of Q is the middle one. Then, we obtain the global dot product using *y_fsum* (figure 5b). Vector *qtz* stores the product $Q^T z$, so that one element of *qtz* is obtained for each iteration of the algorithm.

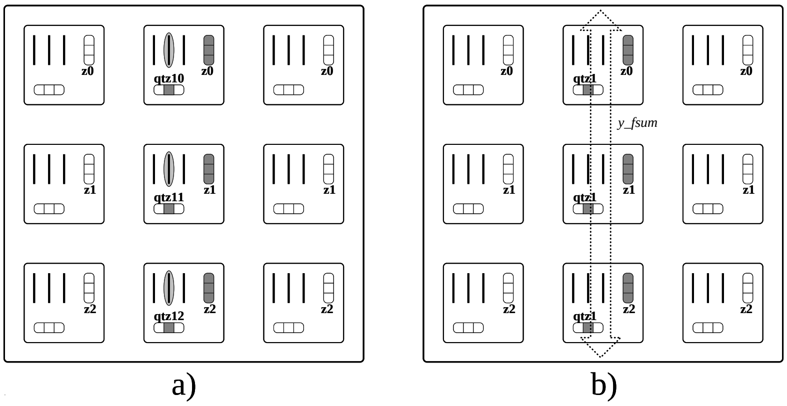


Figure 5: Obtaining $Q^T z$

3.4 Back-substitution and permutation

The back-substitution will solve the system $Rx = Q^T z$. The corresponding sequential algorithm is the following:

for (i=rank-1;i≥0;i- -)
$x_i = (qtz_i - \sum_{j=i+1}^{rank-1} r_{ij} x_j)/r_{ii}$;   being r_{ij} element (i,j) of matrix R

Here, we have a loop with data dependencies, and thus this loop must be maintained in the parallel code without any possibility of distributing it among the cells. In addition, it is necessary to access the elements of matrix R by rows (matrix R is stored by columns). This is solved by using an auxiliary pointer vector with as many components as columns in the matrix. In this way we can go through the linked lists corresponding to the columns of the matrix only once to gain access to matrix R (from bottom to top) by rows. Another option is to apply the column version of back-substitution [1, chap.3].

Once back-substitution is carried out we obtain, in each cell row, the solution vector x of global length B cyclicly distributed, so that the global component J of vector x will be replicated in the cell column with *idx=J mod n.*

Previously, we obtained a $M * \Pi = Q * R$ factorization, where Π is a permutation $B * B$, made up by the product of *rank* elementary permutations: $\Pi = \pi_0 * \pi_1 * ... * \pi_{rank-1}$, being

Matrix	*Origin*	*A∗B*	*Elem(M)*	*Elem(R)*
WELL1850	Least squares problems in surveying	1850∗712	8758	36913
SHERMAN1	Oil reservoir simulation	1000∗1000	3750	36711

Table 1: Harwell-Boeing sparse matrices for the evaluation

each π_j the identity matrix or a matrix resulting from swapping two of its columns. This was due to the pivoting we carried out in the *Column_Swap* procedure. In order to obtain the final solution to the least squares problem we will have to apply the Π permutation to the components of vector x obtained in the back-substitution, so that we overwrite x with vector Πx. All the cells will store a vector called *permut* of local length B. It is the only global vector whose components are not distributed among the cells, but which are completely stored within them. This vector will contain the index of the swapped column (pivot column) in each iteration and applying the swaps stored in *permut* starting from the end, we obtain the elements of vector x in the correct order.

4 Experimental results

With the aim of testing the performance of the parallel algorithm we have used the *Performance Analyzer* of the *AP1000*, which analyzes performance during execution and provides several temporal measures: times of the task, of inactivity of the cells, of interruptions, of calls to libraries, average, maximum and minimum values of these times for all the cells, information on various events, etc.

We have chosen two sparse real matrices with assembly format (sparse by column) from the Harwell-Boeing sparse matrix collection [10]. The selection of these matrices is due to the fact that they include vector z, necessary for solving the least squares problem and, in particular, the matrix *WELL1850* is specific for the solution of this problem. Table 1 presents a description of these matrices. $A{*}B$ indicates the dimensions of the matrix; *Elem(M)* the number of nonzero elements and *Elem(R)* the number of nonzero elements of matrix R, which shows the fill-in of the algorithm.

The factors that are going to influence the fill-in of a given sparse matrix are: the dimension and rank of the matrix, the degree of sparsity (number of null elements) and the pattern of the matrix, that is, the location of the nonzero elements, a factor of great importance but difficult to model. Thus, with the same dimension, rank and degree of sparsity for two matrices, the fill-in may vary significantly, depending on how the nonzero elements are placed. Consequently, there will also be large variations in the execution

Matrix	*1∗1*	*2∗2*	*4∗4*	*8∗8*
WELL1850	690.54	194.71	63.81	21.31
SHERMAN1	261.94	100.10	39.69	18.79

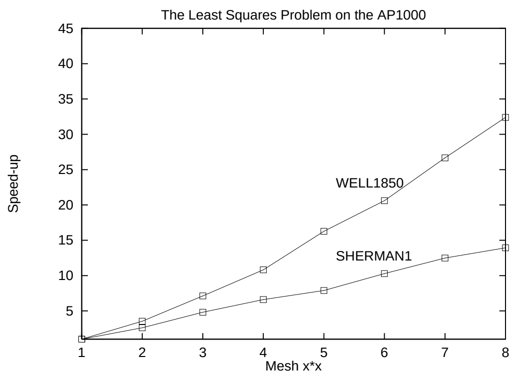
Table 2: Execution times (in seconds) for different sizes of the mesh

times and efficiencies. The reduction of the fill-in is the work we are currently carrying out by means of changing the pivoting strategy.

Table 2 shows the execution times (in seconds) for some of the configurations of the mesh. The times required for the distribution and recollection of the data are not included because we assume that the problem of solving least squares is a possible subproblem within a wider program.

Speed-ups and efficiencies obtained for these sparse matrices are shown in figure 6. As we see, the algorithm scales rather well. The results are better for larger sizes of the matrix because there are more calculations and, thus, the parallelism is more efficiently used. Small sizes of matrix M result in low efficiencies because the calculation time of the task itself is small with respect to the additional time required by communications (message passing) and other factors. Therefore, the matrices must have enough size to exhibit a complexity that makes them worth being implemented on a multiprocessor machine like the AP1000.

In addition, a larger number of nonzero elements (as is the case of matrix *WELL1850*) also provides longer effective calculation times and the parallelism applied gives rise to a longer calculation time of the task in each processor with respect to the times that are not related with the task itself.



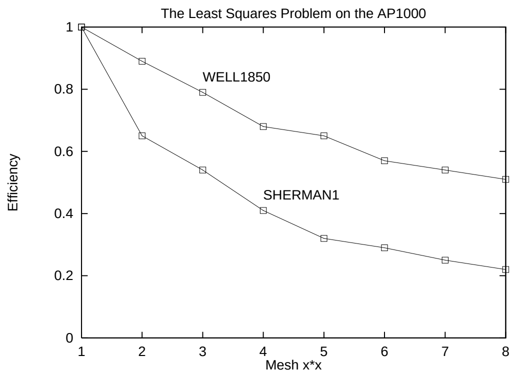


Figure 6: Speed-up and efficiency measures

Acknowledgments

We gratefully thank the Fujitsu Parallel Computing Research Centre (Japan) and the Galician Supercomputing Centre (Spain) for giving us access to the AP1000 machine.

References

[1] G.H.Golub and C.F.Van Loan. *Matrix Computations.* The Johns Hopkins University Press, second edition, 1989.

[2] A.Pothen and P.Raghavan. Distributed orthogonal factorization: Givens and householder algorithms. *SIAM J. Sci. Statist. Computing*, 10:1113–1134, 1989.

[3] E.L.Zapata, J.A.Lamas, F.F.Rivera, and O.G.Plata. Modified gram-schmidt qr factorization on hypercube simd computers. *Journal of Parallel and Distributed Computing*, 12:60–69, 1991.

[4] B.B.Zhou and R.P.Brent. Parallel implementation of qrd algorithms on the fujitsu ap1000. Technical Report TR-CS-93-12, Computer Sciences Laboratory. The Australian National University. Canberra, ACT 0200, Australia, November 1993.

[5] S.G.Kratzer. Sparse qr factorization on a massively parallel computer. *The Journal of Supercomputing*, 6:237–255, 1992.

[6] P.Matstoms. *Sparse QR Factorization with Applications to Linear Least Squares Problems.* PhD thesis, Department of Mathematics. Linköping University. Sweden, 1994.

[7] R.Doallo, B.B.Fraguela, J.Tourino, and E.L.Zapata. Sparse qr factorization: evaluation of two data storage methods (in spanish). In *Proceedings of VI Jornadas de Paralelismo, Barcelona*, pages 9–17, July 1995.

[8] I.S.Duff, A.M.Erisman, and J.K.Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, 1986.

[9] H.Ishihata, T.Horie, S.Inano, T.Shimizu, S.Kato, and M.Ikesaka. Third generation message passing computer ap1000. In *Proceedings of ISS'92*, pages 46–55, 6-9 November.

[10] I.S.Duff, R.G.Grimes, and J.G.Lewis. User's guide for the harwell-boeing sparse matrix collection. Technical Report TR-PA-92-96, CERFACS, October 1992.