Applied Mathematical Sciences, Vol. 2, 2008, no. 4, 149 - 159

Monte Carlo Matrix Computations based on Compression

Behrouz Fathi Vajargah

University of Guilan, Iran
fathi@guilan.ac.ir

Abstract

Here we construct an algorithm based on the minimisation of data by using a compressed matrix for parallel matrix computations. The idea is to minimise the communication between master and slave by eliminating the all zero elements of coefficient matrix.

1. Introduction

The non-singular matrix *B*, in the $Bx = f$ system is compressed to a vector format and used in the simulation of the matrix of the system to be solved by a MC using both max-norm and row-norm solutions.

We expect to observe the same behavior in computation in the solution of SLAE, when using the individual row-norm values and the maximum row-norm value to be same as the case of the non-compressed matrix. We do expect a difference in the speed of computation. In completed tests, a system containing a diagonally dominant sparse matrix was used. It is through the manipulation of the sparse matrix (i.e. the elimination of any dependence on the zero elements) that we seek to further reduce the computation time.

The use of Monte Carlo methods can give a good estimation of the solution of large sparse systems in particular diagonally dominant systems. The use of Monte Carlo methods for solving *SLAE* gives us an inherent parallelism. The solution can be realized in a distributed form and amalgamated at the end of the computation allowing a completely parallel method.

The *SLAE* is presented in the form $x = Ax + f$, where f is a given vector and we are looking for solution vector *x*. Note that *A=I-B, I* is an identity matrix of size *n, A* is an $n \times n$ coefficient matrix.

We seek to reduce the computational time by introducing a cut off point into the algorithm. When the solution element is within the required accuracy variable for two successive returned values then the solution of that element is deemed to be complete and a new component of the solution is computed.

2. The storage of requirements

One of the most important parameters in computational problems is the required memory for the storage of data. To reduce the cost of storage and complete the computation, we require two different steps to independently perform our computations. Since the zero elements of the coefficient matrix are not needed, then the elimination of zero elements of the matrix is considered. It means, we compress a given matrix where the number of elements in compressed matrix directly depends on density (or sparsity) of the matrix. Suppose *A* is $m \times n$ matrix and the density of *A* is *s%* then the number of elements in the compressed matrix (or the number of non-zero elements in matrix *A*) can be computed by:

$$\omega = s\% \times m \times n, \qquad (4)$$

and the number of zero elements of matrix *A* is determined by

$$\eta = (1 - s\%) \times m \times n, \qquad (5).$$

Equation (5) shows that how many numbers of elements of matrix *A* should be eliminated in the compressed matrix.

The compression of a given matrix is not a complex procedure. However there are a few methods for storing data, we select the simplest way for doing this task. We save the amount of the elements of the matrix in a format with its corresponding indices and the value of the corresponding elements. This constructs a matrix with 3 columns. The compressed matrix has ω rows which is obtained from (4). The number of rows in the compressed matrix is equal to *NNZ* (or the number of non-zero) elements of *A*. We can summaries these explanations by the following algorithm:

2. Compressing Algorithm:

$\text{int}\, k = 0;$

For i=0 to n-1

For j=0 to n-1

If (a[i,j]!=0.0) **then**

Stork[k]=a[i,j];

Istor [i]=i;

Jstor[j]=j;

k+=0;

End If

End of J Loop

End of I Loop

Print the stored vector ***Stor*** and the relevant indexes ***Istor*** and ***Jstor*** as a compressed matrix.

End of Algorithm

In the algorithm in this paper, we send our data from the master processor to the slave processors. We send only the realized compressed matrix, instead of sending the all elements of matrix *A*. We note that any computational results such as, the solution of linear systems, inversion matrices and etc have the same accuracy in comparison with computational results based on sending the all elements of matrix. Here, we intend to join the previous method for the reduction of Markov chains iteration in the solution of SLAE. By applying the row-norm value instead of the max-norm value, we expect that this method will effectively increase the speed of computation.

3. Sparse Matrix Compression

Consider a general $m \times n$ matrix A; the matrix will contain density (sparsity) factors. The non-zero elements of a matrix can be described by

$$Nonz(A) = \{a_{ij} \neq 0 \mid i, j \in [1,m] \times [1,n]\} \qquad (1)$$

If we define the number of non-zero elements in the matrix as λ, the density of the matrix is therefore

$$\kappa = \frac{\lambda}{m \times n}, \qquad (2)$$

the sparsity can now be described as 1-κ.

The diagonally dominant matrix is converted to a vector format as described above. As we are only dealing with the non-zero elements we have row-norm array in the form of vector.

We recall that the number of required Markov chains is given by $N \geq \frac{0.6745^2}{\varepsilon^2} \frac{\| f \|^2}{(1-\| A \|)^2}$, where ||*A*|| is the norm (max-norm) value for the matrix *A* [2].

The optimum norm for each row-norm is given by (2) and the maximum norm is given in (1).

We are using a Monte Carlo Markov Chain with a simple random walk. Corresponding to the estimator of SLAE i.e. $\Theta(g)$, we drop the chain when $|W_i f_i| < \delta$, where δ is a value of accuracy that indicates the termination of the Markov chain [1]. We recall that for a $t_0 \to t_1 \to ... \to t_i$, we have:

$$\Theta\ [g] = \frac{g_{t_0}}{p_{t_0}} \sum_{j=0}^{\infty} W_j f_{t_j} \text{ and } W_0 = 1, W_j = W_{j-1} \frac{a_{t_{j-1}t_j}}{p_{t_{j-1}t_j}}, \qquad (8)$$

where g_i is the i^{th} element of vector *g* defined in the dot product of <*g* ,*x*>, p_i is the i^{th} element of the initial probability of Markov chain and the value f_i is the i^{th} element of *f*, in *x*=*Ax*+*f* [4].

The value β reduces the number of chains in comparison with the previous summation and the present. If the previous and present result show no marked variation then it is reasonable to assume that the solution is complete or is close enough to the solution to stop the Markov chains, i.e. $|\Theta_i^{(k+1)} - \Theta_i^{(k)}| < \beta$ is valid we drop the random walks.

To calculate the solution vector x, in $x^{(k+1)} = Ax^{(k)} + f \quad k=0,1,\ldots,n$, if $\rho(A) \leq \| A \| = \max_i \sum_{j=1}^{n} | a_{ij} | < 1$, then the solution of $x^{(k)} \to x$ as $k \to \infty$, where x is the exact solution of given SLAE.

Then Monte Carlo estimation of the vector solution is given by:

$$x_i \approx \hat{\Theta}^* = \frac{1}{N} \sum_{s=1}^{N} \Theta_j [e(i)]_s \qquad (9)$$

where N is the number of Markov chains and $\Theta_j [e(i)]_s$ is the value of $\Theta_j [e(i)]$ in the s^{th} chain.

4. Parallel implementation

The solution was achieved using *PVM* (Parallel Virtual Machine) with a standard master / slave format, the master being responsible for all partitioning and the amalgamation of results. The methods were limited to the methods used in [4] i.e. each slave comprised only one machine and used only the memory allocation of that machine.

To achieve a reasonable balance, the matrix was partitioned using the number of Markov chains determined by the row-norm.

The very nature of the Monte Carlo method ensures that the data passing between master and slave is minimised. The whole $A = I-B$ matrix was sent to each slave processor with only the relevant segments of the solution vector returned.

It is data independence that also allows the scale of the *SLAE* to be increased.

Alexandrov et al [2], passed the full matrix to the slave to allow the independent processing of the solution. In a diagonally dominant sparse matrix this involved a considerable amount of redundancy in the form of the repeated zero elements. In this system only the non-zero elements were passed to reduce the primary storage.

5. Balancing

Balancing in this method distributes approximately the same information (data) to each slave for the relevant computations. After compressing the (sparse) coefficient matrix included 160 rows, we consider the number of slaves to be 2,4 or 8 and so on for our computations. If we use three slaves, this could cause an imbalance by 2 processors (slaves) with the same partition will take 50 rows and third one takes 60 rows. In this case, when two slaves have completed their computation and returned their results to the master, the master will still have to wait for third slave to complete its results.

Trivially, if the matrix and compressed matrix are big enough then the balance procedure can be ignored. For example if the compressed matrix has 100 rows and we use 3 slaves then the distributed data are such that, slaves 1,2 with 33 and slave 3 with 34 rows. Since this difference is not significant it will not affect the computation. Balancing here is used for those cases where the differences in amounts of data in individual partitions are high. In this case we have to use the number of processors in such a way that an efficient balance of data between them is available. Experimentally, it is better to consider balance with regards to the number of Markov chains for each row i.e. N_i also.

6. Experimental results

The following tables show the test results from the row-norm and max-norm cases for solving the *SLAE*.

The time taken is determined from the end of the spawn to the return of the completed solution vector.

The max-norm method which passed the full matrix to each machine may not be used successfully on our limited system for any matrix of appreciable size over 1000×1000. This meant that there was no table for a 10000×10000 matrix for the max-norm method.

Also, the correlation coefficient between the time required to complete the computation *T*, and the number of processors *P* used can be determined by (3), where p_i is the number of processors, t_i is the required time corresponding to p_i processors.

The results of *R* lie between 1 and –1. *A* negative result tends to -1 shows that there is a high negative dependency on the number of processors to the

reduction in time. A positive result tends to +1 shows there is a high positive dependency on the number of processors for an increase in time. In both cases the relationship between the required time T and the number of processors P can be shown as $T=a+bP$

$$a = \frac{\sum_{i=1}^{n} t_i - b\sum_{i=1}^{n} p_i}{n} \qquad (10)$$

where

$$b = \frac{n(\sum_{i=1}^{n} p_i t_i) - (\sum_{i=1}^{n} p_i)(\sum_{i=1}^{n} t_i)}{n(\sum_{i=1}^{n} p_i{}^2) - (\sum_{i=1}^{n} p_i)^2}$$

To interpret the graphs we exchange the symbols y and x to T and P respectively as in the equation $T=a+bP$

Table 1: Time in seconds to reach the solution vector using NMAX

Processors Matrix size	**1**	**2**	**4**	**6**	**8**	**10**	**R**
100	286.318	143.397	72.178	52.302	40.271	32.93	-0.83
500	1494.473	574.193	302.706	219.099	174.823	166.83	-0.79
1000	8856.155	4480.574	2487.332	1633.333	1470.917	1162.348	-0.83

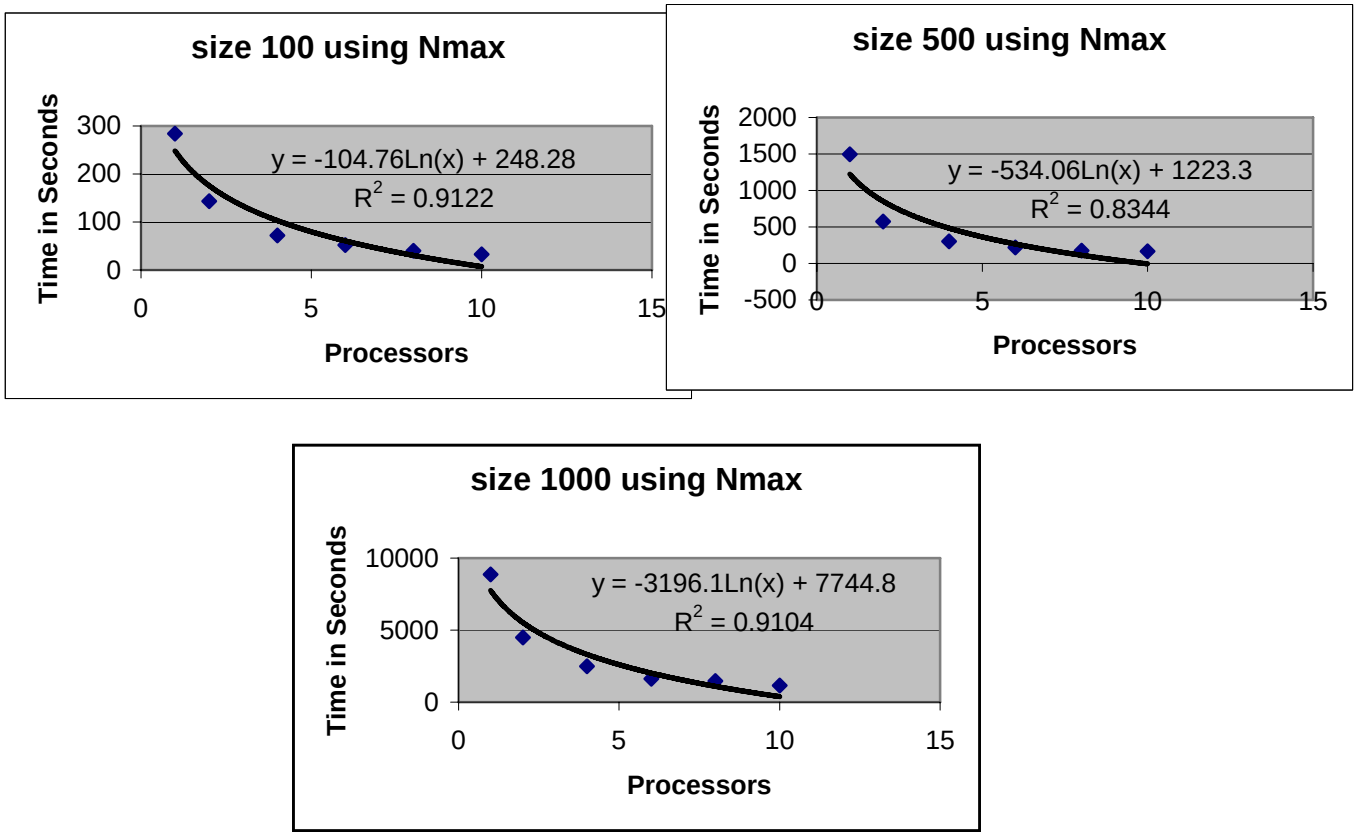

Figure 1: Regression model of computational times using max-norm

Table 2: Time in seconds to reach the solution vector x using row-norm compression

Processors Matrix size	1	2	4	6	8	10	R
100	117.426	63.544	34.729	28.826	19.414	16.940	-0.85
500	220.652	117.467	62.059	54.285	45.315	31.757	-0.84
1000	959.988	632.583	378.824	343.722	317.141	309.62	-0.83
10000	4213.898	2157.642	1194.339	748.385	587.491	449.717	-0.85

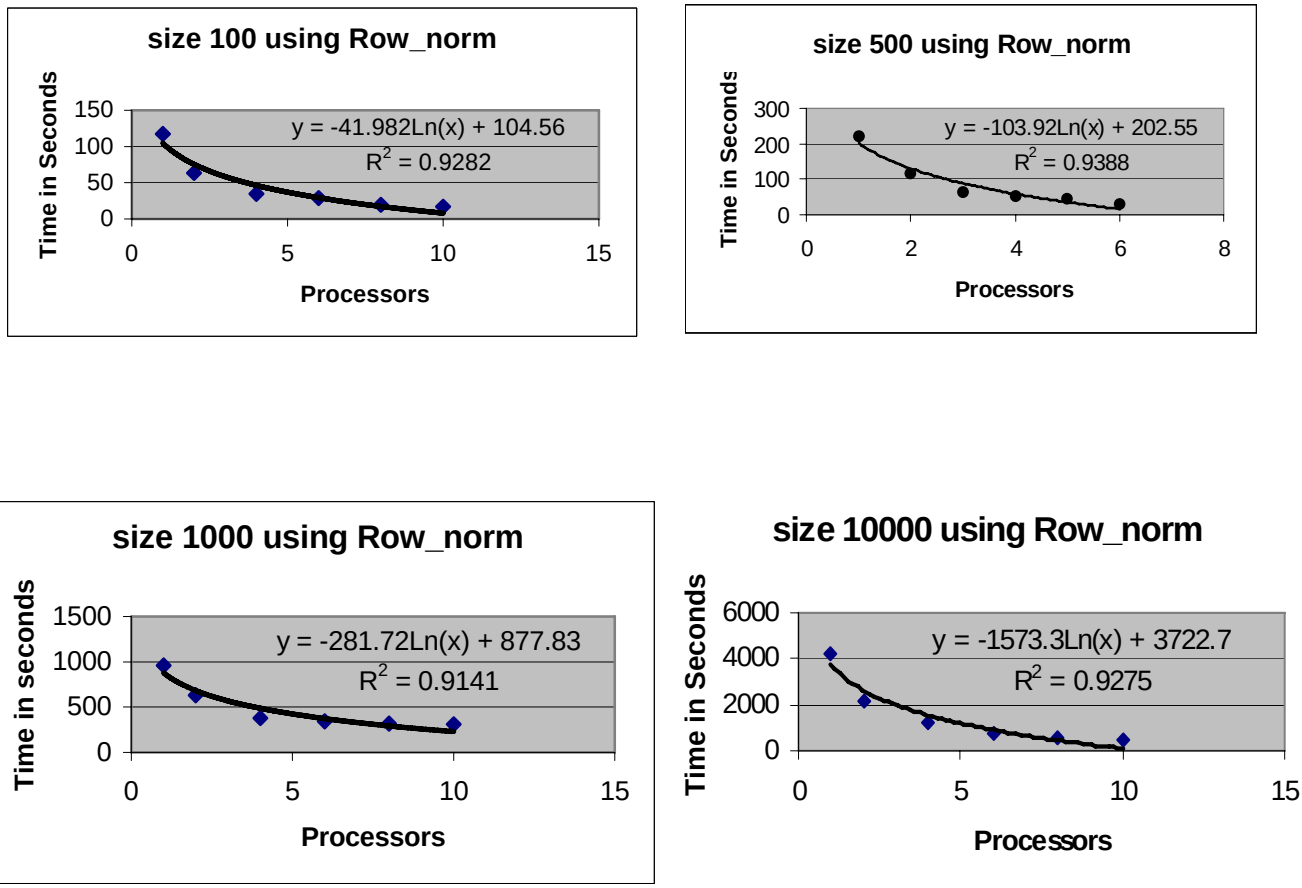

Figure 2: Regression model of computational times using row-norm

Table 3: Rate of required time

Matrix size	summation the time of Max-norm	Summation the time of Row_norm	T_max-norm/T_Row-norm
100	627.393	280.879	2.23
500	2932.124	531.535	5.52
1000	20090.659	2941.878	6.82

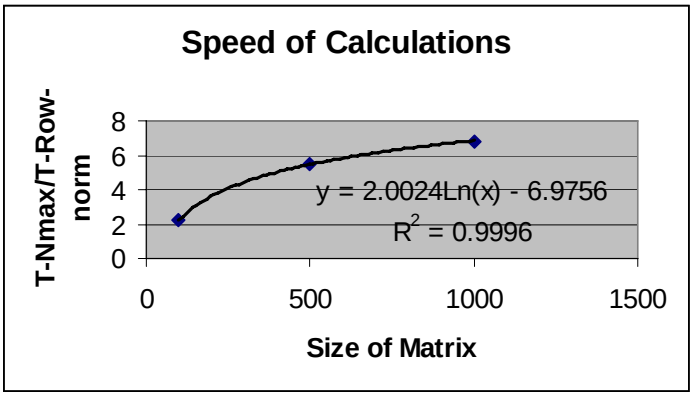

Figure 3: The progress of method on computational time

The model given for the speed of computation shows a significant reduction in computation time, we can use the realized model of the rate of computational time for prediction while computing larger matrices. It is well known that the regression model can be used for a prediction of future. For example, we can estimate the approximate rate of computational time based on our obtained model: *Y=2.0024Ln(X)-6.9756*, for any matrix size $n>1500$ so, where X is the size of matrix and Y is the rate of computational time. Then we can consider the following table:

Table 4: Rate of required time by estimated model

size	2000	5000	10000	20000	50000	100000	200000
T_1 / T_2	8.24	10.08	11.48	12.86	14.69	16.08	17.47

As the results are shown in the above table, we see the rate of improvement in terms of reduced computational time for dimension n=5000 is about 10 (i.e. method with compressed matrix n=5000 is 10 times faster) and for n=200000, the rate of improvement is expected to be 17.

7. Conclusion

We observed the following results:

For a rapid approximate calculation, the experimental results show that there is an improvement using the individual row-norm value as an indicator of the optimum number of Markov chains to be used and the reduction of the matrix to

only the non-zero elements. Using this method, we have achieved a reduction of 2 to 7 times of computational time in comparison with the more conventional method of full matrix with max-norm for the values tested.

The comparison of given tables shows that in the case of using the row-norm value with a matrix compression, not only we have an increased dependency between the time taken and the number of processors, but we also have a reduction in the computational time by at least two times.

We have noted that although there was a reduction in the number of Markov chains used, there were some problems which required 'at least' the estimated optimum number of Markov chains as determined by the row-norm value. The passing of a greatly reduced representation of a matrix in the form of a vector and eliminating the zero elements is extremely effective when processing sparse matrices. However with dense matrices obviously little or no reduction can be achieved.

Although only diagonally dominant sparse matrices were used in the initial trials the compression approach holds for all sparse matrices, as a reduction in both storage and processing can be made by elimination of unnecessary or duplicated elements.

References

[1] E. Cinlar, *Introduction to Stochastic Processes*, Prentice-hall, Englewood Cliffs, New Jersey, 1975.

[2] B. Fathi. Vajargah, PhD thesis, *Parallel Monte Carlo Methods for Matrix Computations*, UK, Reading University, 2003.

[3] R.Y. Rubinstein *Simulation and the Monte Carlo method*, John Wiley & Sons, New York , 1981.

[4] I.M. Sobol, *Monte Carlo numerical methods*, Moscow, Nauka, 1973 (in Russian).

Received: September 21, 2008