PARALLEL IMPLEMENTATION OF LEMKE'S ALGORITHM ON THE HYPERCUBE

by

R. De Leone
and
T. H. Ow

Computer Sciences Technical Report #905

January 1990

ORSA Journal on Computing
Vol. xx, No. xx, xxx

0899-1499/90/xxxx-xxx $01.25

Parallel Implementation of Lemke's Algorithm on the Hypercube

RENATO DE LEONE *Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin Madison, 1210 West Dayton Street, Madison, WI 53706, ARPANET : deleone@cs.wisc.edu*

TERENCE THONG-HWEE OW *Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin Madison, 1210 West Dayton Street, Madison, WI 53706, ARPANET : ow@cs.wisc.edu*

(Received N/A; accepted N/A)

A simple but very effective method for parallelizing Lemke's algorithm for the solution of linear complementarity problems is presented. Implementation details on a 32-node Intel iPSC/2 hypercube for problems of dimension up to 1000 are discussed. A speedup efficiency as high as 76% is achieved with 32 processing nodes for a problem with 500 variables and 250,000 nonzero elements. By combining the effects of concurrency and vectorization the computing time on the Intel iPSC/2 in some cases is reduced by a factor of 100.

In this paper we study the effectiveness of parallel implementations of the Lemke's algorithm (also called the Complementary Pivot Algorithm) [5, 2] for the linear complementarity problem (LCP):

$$w = Mz + q \geq 0, \quad z \geq 0, \quad z^T w = 0 \tag{1}$$

where M is a real n-dimensional matrix and q is a given vector in $\mathbb{R}^n$. A parallel version of the algorithm was first proposed by Thompson [8] and near linear speedup was achieved by her implementation on the CRYSTAL multicomputer [3]. The intent of this paper is to report on new experience on the Intel iPSC/2 and on how the multi-processor architecture and vector capabilities of the machine were beneficial to the algorithm. Both the

Subject classification: **Programming:** Complementarity, Linear, Algorithms
Other key words: Linear complementarity, parallel algorithms, Lemke's algorithm, hypercube

CRYSTAL multiprocessor and the Intel iPSC/2 are distributed memory machines that utilize message passing for interprocessor communication. However, the topologies of the two networks are substantially different.

For the problem considered here, the number of variables ranges between 100 and 1,000. For this class of problems the efciency of our parallel implementation increases monotonically with problem size and near linear speedup is achieved for the larger problems. The use of vector routines also substantially decreases solution time and in many cases the time{reduction factor from vectorization is more than one order of magnitude. Combining these two effects (vectorization and concurrency) we are able to decrease the computing time, in some cases, by almost a factor of 100.

The paper is organized as follows. In Section 1. we will discuss briey Lemke's algorithm. In Section 2., the parallel algorithm is introduced. Computational results utilizing the scalar and vector processing capabilities are discussed in detail in Section 3.. Three different sets of randomly generated linear complementarity problems are solved and solution time and efciency are reported using the scalar and vector versions of the algorithm.

In our notation, a superscript T will denote the transpose and the inner product of two vectors z and w in $\mathbb{R}^n$ will be denoted by $z^T w$. For an mxn real matrix A, A_i will denote the i^{th} row of A, A_{ij} the element of A in row i and column j, and for any nonempty $I \subset \{1, \ldots, n\}$, $A_{.I}$ will denote the submatrix of A obtained by removing all columns i of A such that $i \notin I$. Finally, e is a vector of ones of arbitrary dimension.

1. Lemke's Algorithm for solving the LCP

The initial step of Lemke's Algorithm enlarges the space of the linear complementarity problem (1) by adding an articial variable z_0. An augmented linear complementarity problem(equivalent to the original one) is obtained:

$$w = Mz + ez_0 + q \geq 0, \quad w_0 = z_0, \quad (z, z_0) \geq 0, \quad (z, z_0)^T (w, w_0) = 0 \tag{2}$$

For this problem a basic feasible point, that is a basic vector such that $(w, w_0, z, z_0) \geq 0$, is immediately available. A solution $(z, z_0), (w, w_0)$ is said to be *almost complementary* if it is feasible and $z_i w_i = 0$ for all $i = 0, \ldots, n$ except for at most one i. The complementary pivot algorithm moves (at every nondegenerate step) from a basic vector for (2) to a new one maintaining the property of almost complementarity between (z, z_0) and (w, w_0) until either a complementary basic feasible solution for (2) is obtained or an unbounded ray is detected (see [6] for the denition of basic vectors and a more detailed explanation of the algorithm). At every step of the algorithm, a Gauss-Jordan elimination pivot operation is performed on a reduced tableau. The current basis is modied by bringing into the basis a new variable and dropping a variable from the current set. The new entering variable is uniquely determined by the complementary pivot rule (that is pick the variable which is complementary to the variable that just left the basis). Dropping a variable from the basis is determined by the minimum ratio test of the simplex algorithm.

We refer the reader to [6] for questions regarding the convergence of the algorithm. Sufce it here to say that the method terminates in a nite number of steps for important classes of matrices such as positive semidenite matrices and matrices with positive entries.

2. Implementation of Parallel Lemke's Algorithm

An examination of Lemke's algorithm indicates that it possesses many options for parallelism. In particular, updating of the columns of the matrix can be done concurrently, provided that the pivot column is available. Following [8] we decided to partition the matrix M among the nodes by columns, i.e. each node receives a subset of the columns of the matrix M. Thereafter, the node is responsible for updating this portion of the matrix. In order to minimize idle time for the processors, the matrix is partitioned in such a way that, given any two nodes, the number of columns residing in their respective memory differs by at most one. A copy of the right-hand-side vector q is also made available to all nodes and a list of current basic and nonbasic variables is maintained.

At the beginning of each iteration, the pivot column is determined by the complementary pivot rule and the node in charge of this column broadcasts it (before updating) to the remaining nodes. Each node independently computes the pivot row using the minimum ratio test and updates a portion of the matrix. This can be done in parallel, since each processing element has an updated copy of the right-hand-side vector q and the current pivot column.

We are now ready to present our parallel version of Lemke's algorithm.

Parallel Lemke's Algorithm

Let $I_1, I_2, \dots, I_r$ be a partition of $\{1, \dots, n+1\}$ where r is the number of processors used. Each node i performs the following steps:

```
Receive $M_{.I_i}$ and the right-hand-side $q$.
While (complementarity condition is not satised) do
    a) Determine if the pivot column index is in $I_i$.
    b) If the pivot column index is in $I_i$ then
           send pivot column
       else
           receive pivot column
       end if
    c) compute ratio test and determine pivot row.
    d) update $M_{.I_i}$ and the right-hand-side $q$.
end do
```

We observe that synchronization is needed between iterations and a vector of length n, must be broadcast at each iteration.

Vector processing capabilities are also exploited in this implementation on the Intel iPSC/2. During each iteration, the updated pivot column in the so-called \reduced tableau" is obtained by rescaling the current pivot column. This can be achieved by calling the vector routine x**SCAL**. For a nonpivot column, if the elements corresponding to the pivot row are not zero, the updated column can be obtained by subtracting from it a suitable multiple of the pivot column. This is achieved by calling the vector routine x**AXPY**.

Two special features of vector nodes are also used in order to obtain higher performance. Each vector board of the Intel iPSC/2 has, in addition to the 1M byte of dynamic memory, 16K bytes of static (fast) memory. Approximately 5K bytes of fast memory are not used by the operating system but available to the user. In our implementation the pivot column is copied into the static memory of the processor board. Since access time to static memory is less than half that of dynamic memory and the pivot column is repetitively used in updating nonpivot columns, we were able to substantially decrease the time needed for the updating portion of the algorithm. Note that, due to the 5K bytes limitation, when $n > 640$ the pivot column is not all contained in the static memory but spills over into dynamic memory. Consequently, the biggest improvement from using this feature was observed for problem of dimension 640 or smaller.

Next, in order to reduce the call overhead for vector routines, the asynchronous version of the *x***AXPY** routine is used. The asynchronous routines queue the vector operation on the vector board and return to the caller before the operation itself is complete. Hence, while the vector board is still updating a column of the matrix, the scalar processor proceeds to the next column that needs to be updated and queues also this new operation on the vector board. Therefore the vector processor is always kept busy during this part of the algorithm. In the updating phase, there is no data dependency between different columns. Thus, the vector board and the 386 microprocessor need to be synchronized only at the end of this updating phase.

3. Computational Experience

Computational testing was carried out on a Intel iPSC/2 [4] with 32 nodes. Each node has a 32-bit Intel microprocessor 80386 with a local memory of 8 MB. The nodes are connected together in a hypercube topology. Each 80386 processor is accompanied by an 80387 numeric coprocessor and 16 of the 32 nodes have also a VX vector board. Support for synchronous and asynchronous message-passing and broadcast routines is provided by the Intel iPSC/2.

Three sets of randomly generated test problems were solved, corresponding to symmetric positive semidenite linear complementarity problems, multi{commodity spatial equilibrium problems and linear complementarity problems with diagonally dominant matrix M.

The rst set of problems is generated as follows. First, an mxn matrix A is generated with elements from a uniform distribution in the interval [-10, 10]. In our test problems m is chosen to be $0.8 * n$. Then, the matrix $M = A^T * A$ is computed and a vector$z$ is generated with elements from a uniform distribution in the interval [0,1] with 20% of these entries equal to zero. Finally the vector q is created to ensure that$z$ solves the corresponding linear complementarity problem. Problems in this form are very common and arise in least square estimate of nonnegative parameters.

Table I shows CPU time versus number of nodes for the scalar version of the algorithm for different values of n ranging from 200 to 500. Five different test problems are generated for each value of n, with different seeds for the random number generator and the average running time for these 5 cases is presented. The efciency E_r, dened as

$$E_r := \frac{S_r}{r} \qquad \text{where} \qquad S_r(\text{the speedup using } r \text{ processors}) := \frac{T_1}{T_r}$$

where T_i is the CPU time with i nodes, is also reported in Table I. Table II shows CPU times for the vector version of the algorithm. Hardware limitations did not allow us to ll this table completely and we therefore show relative efciency instead of efciency. Relative efciency $RE_{rs}(r > s)$ is dened as

$$RE_{rs} := \frac{sT_s}{rT_r}$$

Relative efciency provides an indication of the improvement obtained by increasing the number of nodes from s to r. In Table II, relative efciency RE_{rs} is reported for $s = r/2$ and different values of r. The time shown for each value of n is the average running time for ve different cases, except for $n = 800$ where a single instance of the problem is solved.

We make the following observations. Parallelization of the scalar version of the algorithm is very effective and efciency as high as 76% was achieved with 32 nodes. The speedup monotonically increases with problem size (Figure 1). Almost linear speedup is achieved for n sufciently large. The combined effect of parallelization and vectorization substantially decreases the CPU time by a factor of almost 100 (Tables I and II). For example, on one scalar node it takes about 1524 seconds (average CPU time) to solve a problem of dimension 500; only 16 seconds are needed when 16 vector nodes are used. The efciency of the vector version of the algorithm is worse than that of the scalar version, although it still increases monotonically with problem size and a drastic reduction in solution time can be observed (Figure 2). This is to be expected since the communication time among nodes remains unchanged for both versions (the pivot column must be broadcast to all nodes) but the updating of the matrix is performed in less time when vector nodes are used.

The second set of test problems consists of randomly generated multi{commodity spatial equilibrium problems. It is well{known that, under assumption of linear supply and demand functions, spatial equilibrium point can be formulated as a linear complementarity problem. We generated our problems as suggested in [7] (where a hybrid method based on block successive overrelaxation method was proposed for the solution). In our examples, the number of commodities varies from 1 to 3 and the number of supply and demand locations ranges from 5 to 25.

Results are shown in Tables III to VI for the scalar and vector version of the algorithm. We note that for these examples also (Figures 3 and 4), the speedup increases with problem size, and a reduction factor of 100 can be achieved by combining concurrency and vectorization for sufciently large problems. However a reduction of efciency for this set of problems with respect to the previous set is observed. The poorer efciency is due to the large number of zero entries in the matrix M for the multi{commodity problems. The time needed for updating the matrix is proportional to the size of the problem n and the number of nonzero entries in the pivot row. The communication time, however, in our implementation depends only on the problem size (see [1] for more details on communication measurements for the Intel iPSC/2). This implies that (for xed value of n) the ratio between computation and communication time decreases if the matrix has many zeroes in the chosen pivot rows. This explains the lower efciency for the latter set of problems.

The vector version of the algorithm was also tested on some larger symmetric linear complementarity problems. This set of problems is generated as follows. First, the off-diagonal elements of the matrix M are generated with entries from a uniform distribution on the interval [-10, 10]. Motivated by the above discussion on speedup of the algorithm and sparsity, we decided to x 20% of the elements of M to 0. Then, for each $i = 1, \ldots, n$, we set $M_{ii} = \alpha \sum_{j \neq i} |M_{ij}|$ where α is chosen to be 1 or 1.25. The vector q is generated from a uniform distribution on [-10, 10] with 20% of the entries equal to 0. Table VII shows CPU time for this set of randomly generated problems with n ranging from 600 to $1,000$. Each entry represents the average CPU time for 5 different problems generated using different seeds. With 16 vector nodes, we are able to solve a problem of dimension $1,000$ in about 46 seconds.

4. Conclusions

Implementation of Lemke's algorithm on the Intel iPSC/2 was carried out with a very high degree of efciency (an efciency of 75% or more was reached for problems of size 500 and larger using 32 nodes). Speedup increased with problem size. Nearly linear speedup was achieved for sufciently large problems. Vectorization and concurrency were combined to reduce computing time by factors as high as 100. Linear complementarity problems of size 1,000x1,000 were solved in less than 1 minute using 16 vector processors.

REFERENCES

1. L. Bomans and D. Roose, 1989. Benchmarking the iPSC/2 hypercube multiprocessor, *Concurrency: practice*

and experience 1, 3{18.

2. R.W. Cottle and G. Dantzig, 1968. Complementary pivot theory of mathematical programming, *Linear Algebra and its Applications, 1,* 103{125.
3. D. De Witt, R. Finkel, and M. Solomon, 1987. The CRYSTAL multicomputer:design and implementation experience, *IEEE Transaction on Software Engineering SE-13,* 953{966.
4. Intel Corporation, 1988. *iPSC/2 User's Guide*, Beaverton, Oregon 97006. order number 311532-002.
5. C.E. Lemke, 1965. Bimatrix equilibrium points and mathematical programming, *Management Science 11,* 681{689.
6. K.G. Murty, 1988. *Linear Complementarity, Linear and Nonlinear Programming*, Heldermann Verlag, Berlin.
7. , 1981. A hybrid method for the solution of some multicommodity spatial equilibrium problems, *Management Science 27,* 1142{1157.
8. , 1987. A parallel pivotal algorithm for solving the linear complementarity problem, Tech. Rep. 707, University of Wisconsin, Computer Sciences Department.

Dimension	200		300		500	
No. of Proc.	Time Sec.	Eff. %	Time Sec.	Eff. %	Time Sec.	Eff. %
1	130.726	|	335.453	|	1524.476	|
2	67.036	97.50	170.641	98.29	770.816	98.89
4	35.155	92.96	88.165	95.12	393.002	96.98
8	19.675	83.05	46.275	90.62	203.471	93.65
16	11.458	71.31	25.766	81.37	109.653	86.89
32	6.825	59.86	16.557	63.32	62.550	76.16
Ave. # of iter.	195.8		282.4		465.6	

Table I. CPU time (in seconds) and efciency vs number of nodes for data set 1 (scalar version)

Dimension	300		500		600		800	
No. of Proc.	Time Sec.	Rel. Eff.%	Time Sec.	Rel. Eff.%	Time Sec.	Rel. Eff.%	Time Sec.	Rel. Eff.%
1	19.358	|	|	|	|	|	|	|
2	11.897	83.46	46.759	|	|	|	|	|
4	7.926	75.05	28.827	81.10	46.734	|	|	|
8	6.403	61.90	20.380	70.72	31.879	73.30	67.940	|
16	5.793	55.26	16.700	61.02	25.057	63.61	50.267	67.58
Ave. # of iter.	282.5		465.6		559.7		783	

Table II. CPU time (in seconds) and relative efciency vs number of nodes for data set 1 (vector version)

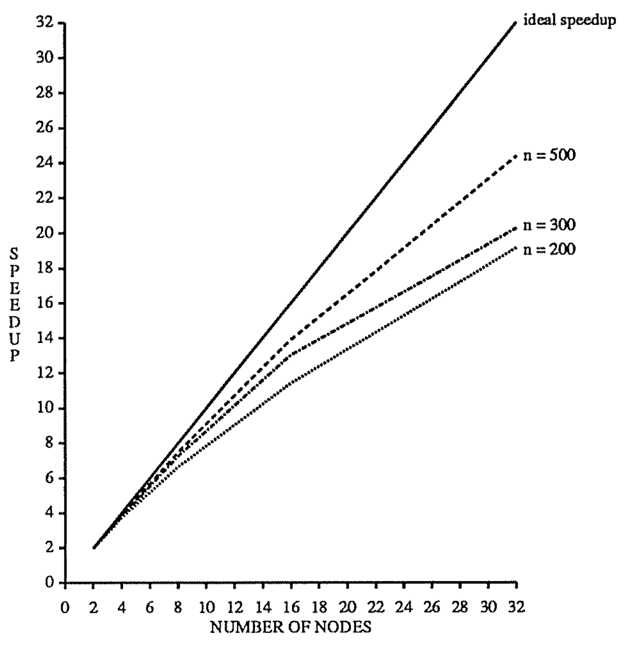

Figure 1. Speedup vs number of nodes for data set 1 (scalar version)

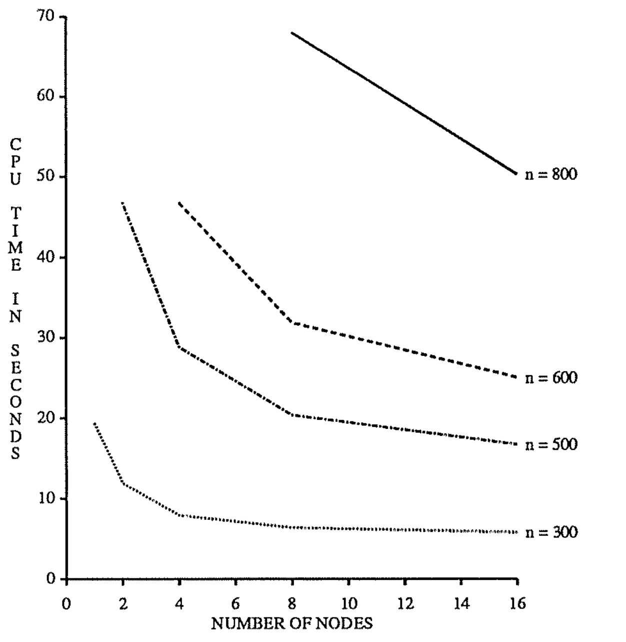

Figure 2. CPU time (in seconds) vs number of nodes for data set 1 (vector version)

Dimension	140		285		480		725	
# Supply	10		15		20		25	
# Demand	10		15		20		25	
No. of Proc.	Time Sec.	Eff. %	Time Sec.	Eff. %	Time Sec.	Eff. %	Time Sec.	Eff. %
1	15.713	|	79.351	|	409.510	|	1368.764	|
2	8.419	93.32	40.876	97.06	212.731	96.25	691.860	98.92
4	4.753	82.65	21.727	91.30	111.938	91.46	353.503	96.80
8	2.965	66.24	12.078	82.12	59.009	86.75	183.563	93.21
16	3.269	30.04	7.050	70.35	32.450	78.87	98.442	86.90
32	2.826	17.39	5.713	43.40	19.990	64.02	56.126	76.21
Ave. # of iter.	88		111		186		267	

Table III. CPU time and efciency vs number of nodes for data set 2, 1 commodity (scalar version)

Dimension	140		285		480		725	
# Supply	10		15		20		25	
# Demand	10		15		20		25	
No. of Proc.	Time Sec.	Rel. Eff.%	Time Sec.	Rel. Eff.%	Time Sec.	Rel. Eff.%	Time Sec.	Rel. Eff.%
1	1.458	|	5.659	|	|	|	|	|
2	1.057	68.97	3.601	78.58	|	|	|	|
4	0.910	58.08	2.649	67.97	9.350	|	|	|
8	0.821	55.42	2.289	57.86	7.079	66.04	17.634	|
16	1.240	33.10	2.321	49.31	6.184	57.24	13.988	63.63
Ave. # of iter.	88		111		186		267	

Table IV. CPU time and relative efciency vs number of nodes for data set 2, 1 commodity (vector version)

Dimension	135		420		855	
# Supply	5		10		15	
# Demand	5		10		15	
No. of Proc.	Time Sec.	Eff. %	Time Sec.	Eff. %	Time Sec.	Eff. %
1	13.655	|	315.920	|	2292.844	|
2	7.219	94.58	162.275	97.34	1162.387	98.63
4	4.080	83.67	84.574	93.39	593.511	96.58
8	2.625	65.02	45.478	86.83	308.126	93.02
16	2.701	31.60	25.561	77.25	164.919	86.89
32	2.366	18.04	16.780	58.83	94.861	75.53
Ave. # of iter.	87		204		337	

Table V. CPU time and efciency vs number of nodes for data set 2, 3 commodities (scalar version)

Dimension	135		420		855	
# Supply	5		10		15	
# Demand	5		10		15	
No. of Proc.	Time Sec.	Rel. Eff.%	Time Sec.	Rel. Eff.%	Time Sec.	Rel. Eff.%
1	1.367	|	|	|	|	|
2	1.016	67.27	12.099	|	|	|
4	0.890	57.08	8.212	73.67	|	|
8	0.808	55.07	6.409	64.07	27.906	|
16	1.173	34.44	5.677	56.45	21.723	64.23
Ave. # of iter.	87		204		337	

Table VI. CPU time and relative efciency vs number of nodes for data set 2, 3 commodities (vector version)

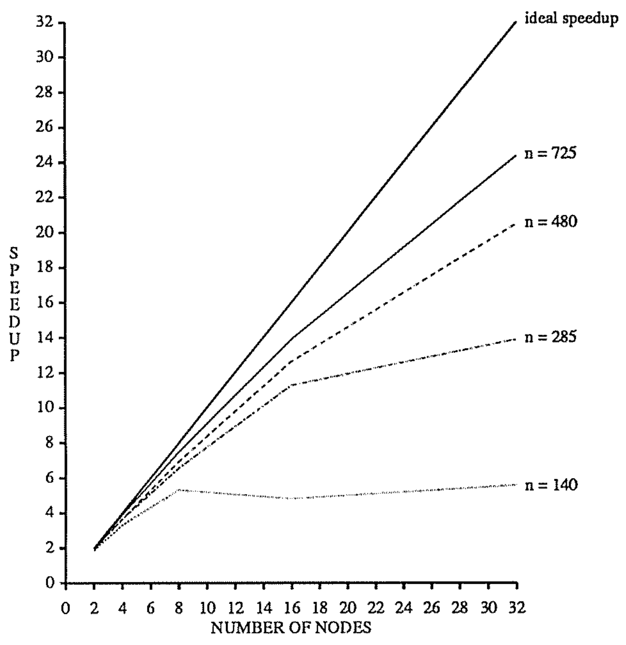

Figure 3. Speedup vs number of nodes, 1 commodity (scalar version)

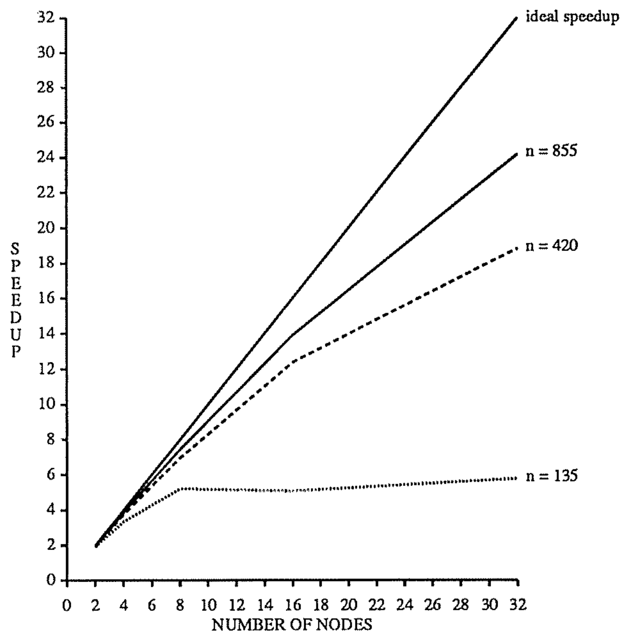

Figure 4. Speedup vs number of nodes, 3 commodities (scalar version)

Dimension	**600**		**800**		**1000**	
No. of Proc.	**Time Sec.**	**Rel. Eff.%**	**Time Sec.**	**Rel. Eff.%**	**Time Sec.**	**Rel. Eff.%**
4	26.752	|	|	|	|	|
8	18.579	71.99	37.316	|	|	|
16	14.844	62.58	28.408	63.32	46.659	|
Ave. # of iter.	**307.2**		**403.6**		**489.0**	

Table VII. CPU time (in seconds) and relative efciency vs number of nodes for data set 3 (vector version)