Solving Nonlinear Matrix Equations on a Hypercube

Dingju Chen*
Yizhong Wu*

TR 90-1161
October 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Department of Applied Mathematics, Cornell University.

Solving Nonlinear Matrix Equations On A Hypercube

Dingju Chen, Yizhong Wu
The Center for Applied Mathematics
Cornell University

July 1990

Abstract

Nonlinear matrix equations arise frequently in applied probability, especially in the numerical solution of many stochastic models in queueing , inventory, communications, and dam theories. Due to the huge amount of computations involved in these nolinear matrix equations, the existing algorithms for the solutions have not been satisfactory. With the advent of parallel computers, the door is open for efficient parallel algorithms to tackle the problem. This paper is an effort in this direction. A parallel algorithm on distributed computer systems is devised, and numerical experiment is done on the hypercube.

Key Words. nonlinear matrix equations, parallel algorithms, hypercube.

1 Introduction

The following problem is under our consideration.

PROBLEM. Given the sequence $\{A_i : i \geq 0\}$ of $m \times m$ nonnegative matrices such that $A = \sum_{i=0}^{\infty} A_i$ is stochastic, that is, $Ae = e$, where e is a vector of 1's, compute the matrix $R \geq 0$ that satisfies the nonlinear matrix equation

$$X = \sum_{i=0}^{\infty} X^i A_i \tag{1}$$

which in addition, is minimal in the sense that for any $X \geq 0$ satisfying (1), then $X \geq R$.

This equation is orginated from the analysis of queues with phase-type service times, as well as in queues that can be represented as quasi-birth-and-death processes. For more details on the background of this problem, refer to Neuts [1].

2 Solution Techniques

A straightforward iterative scheme is

$$R_0 = 0, \quad R_{k+1} = \sum_{i=0}^{\infty} R_k^i A_i, \quad k \geq 0 \tag{2}$$

It was shown (see [2]) to be such that $0 \leq R_k \uparrow R$ as $k \to \infty$ and the convergence is $r - linear$. Numerical experience shows that this scheme can be very slow and is not recommended as the solution method.

It is easy to see that the solution of (1) is equivalent to the solution of $F(X) = 0$, where $F(X) = X - \sum_{i=0}^{\infty} X^i A_i$. With each $m \times m$ matrix X we may uniquely associate the vector which is the m^2-vector formed by the successive columns of X. In this way, $F(X)$ can be regarded as a function from R^{m^2} to R^{m^2}. The application of standard functional analysis shows (see [3]) that $F(X)$ has for its Gateaux derivative $F'(X)$ the linear map

$$F'(X) : H \to H - \sum_{i=1}^{\infty} \sum_{j=0}^{i-1} X^j H X^{i-1-j} A_i \tag{3}$$

The Newton-Kantorovich scheme for (1) is then given by

$$R_{k+1} = R_k - (F'(R_k))^{-1}F(R_k) \tag{4}$$

Denote $S_k = -(F'(R_k))^{-1}F(R_k)$, or equivalently $F'(R_k)S_k = -F(R_k)$. If we substitute (3) for $F'(R_k)$ then S_k can be solved by

$$S_k - \sum_{i=1}^{\infty}\sum_{j=0}^{i-1} R_k^j S_k R_k^{i-1-j} A_i = \sum_{i=0}^{\infty} R_k^i A_i - R_k \tag{5}$$

It then follows that to solve for S_k, m linear systems must be solved at every iteration, which is unacceptable numerically. To circumvent this difficulty, several variations of Newton-Kantorovich scheme are proposed, based on truncating the second term on the left hand side of (5). If it is truncated at $i = 1$ and $j = 0$ respectively, the following schemes emerge

$$R_{k+1} = \sum_{i\geq 0, i\neq 1} R_k^i (I - A_1)^{-1} \tag{6}$$

and

$$R_{k+1} = A_0 (I - \sum_{i=1}^{\infty} R_k^{i-1} A_i)^{-1} \tag{7}$$

which also correspond respectively, to successive substitution schemes for the equations $X = \sum_{i\geq 0, i\neq 1} X^i (I - A_1)^{-1}$ and $X = A_0 (I - \sum_{i=1}^{\infty} X^{i-1} A_i)^{-1}$ which are equivalent to (1). Scheme (7) also suffers from doing a large number of matrix-matrix multiplications and solving systems of linear equations every iteration, hence it is slower even than the direct scheme (2). However, scheme (6) improves greatly over the direct scheme and has the advantage that the matrix $(I - A_1)^{-1}$ needs to be computed only once. Numerical experience also shows that scheme (6) has about 20 percent reduction of CPU time and number of iterations to those of the direct scheme.

However, a disadvantage of (6) is that it dose not incorperate any of the information contained in the current iterate into the construction of the appoximation of $F'(R_k)$. To tackle this problem, V. Ramaswami [3] proposed a new scheme by truncating the second term on the left hand side of (5) at $i = 2$, and obtained the following equation

$$S_k = \left[\sum_{i=0}^{\infty} R_k^i A_i - R_i\right] + S_k A_1 + (R_k S_k + S_k R_k) A_2 \tag{8}$$

Using this equation to solve for S_k still suffers from the need to solve a large linear system at every iteration. Therefore, instead of solving (8), Ramaswami estimated $Z_k = -F(R_k)(I - A_1)^{-1}$ of S_k on the right side of (8), thus

$$S_k = -F(R_k) + Z_k A_1 + (R_k Z_k + Z_k R_k) A_2 \tag{9}$$

which gives a new scheme as follows

Algorithm 1
$R_0 = 0$
$B_1 = (I - A_1)^{-1}$
Repeat
$Z_k = -F(R_k) B_1$
$S_k = -F(R_k) + Z_k A_1 + (R_k Z_k + Z_k R_k) A_2$
$R_{k+1} = R_k + S_k$
Untill

*Fig.*1

This algorithm is proved by Ramaswami to be such that it provides a monotonically increasing sequence of iterates converging to R. Numerical experience also shows that this algorithm results in a 50 to 70 percent of reduction in CPU time over the direct scheme. It is the recommended scheme for the solution of (1), also a basis upon which our parallel algorithm is to be developed.

3 Hypercube Distrbuted Memory Systems

Distributed memory systems are one type of parallel computer systems. They are so called because they have several identical *processors*, each of which having a significant *local memory*. The processors are connected with a *communication* network and they coordinate their computations by sending

and receiving *messages* through the network. A specific *topology* of the network corresponds to a specific kind of distributed memory parallel computers. Each of the processors supports two communication primitives *send* and *recv*. A processor executes a send to transfer information to other prosessors. A processor executes a recv to receive infomation sent by other processors. When a message is sent by a processor, the message remains in a buffer until the designated processor executes a recv. If a processor executes a recv and no message is available, it waits for one to come. Two processors want to communicate don't have to be directly connected since processors will automatically forward messages.

Hypercube is a popular type of topology for the communication network. An n-cube contains 2^n processors, where each processor is linked to n processors. A 0-cube consits of a single processor. A 1-cube is obtained by linking two 0-cubes together. In general, to construct a $n+1$-cube, take two n-cubes and find a one-to-one mapping between the two cubes. Then, connect each pair of corresponding processors by a communication link. Usually, an additional processor, called *host*, is attached to at least one processor in the cube. The host is uaually used by the user to send data to the processors of the cube and collect results from the cube.

In an n-cube, the *diameter* or the shortest path between any two processors is at most n, and for any processor *proc* a spanning tree rooted at proc with depth n can be constructed. Processor *proc* can use this spanning tree to broadcast a message to every other processors in time proportional to that required to sequentially send n messages between two adjacent processors. This is refered to as a *fan out* broadcast. Similarly, a *fan in* can be used to collect information from all other processors onto a single processor. For more topological properties of the hypercube, refer to Saad and Schultz [4].

4 Parallel Algorithms

As pointed out in section 2, our parallel algorithm will be based upon algorithm 1. In that algorithm, the most time consumming computation is the evaluation of $G(R_k) = \sum_{i=0}^{\infty} R_k^i A_i$. In the practical implementation, the sum is truncated at some finite number N, which is determined by some practical criteria. Thus, we need to devise an efficient algorithm for evaluating $G_N(R) = \sum_{i=0}^{N} R_k^i A_i$, which is just a matrix polynomial. A well known

sequential algorithm is Horner's scheme:

Algorithm 2 (HEVAL) Given matrices R and A_i, $i = 0, N$, the following algorithm computes $G_N(R)$

$F = RA_N + A_{N-1}$
for $k = N - 2 : -1 : 0$
    $F = RF + A_k$
end
$G_N(R) = F$

Fig.2 HEVAL

This algorithm requires N matrix multiplications and the same number of matrix additions. It is easy to see that Horner's scheme is optimal in the sense that it minimizes the number of matrix multiplies. However, it is not trivial to parallelize Horner's rule since the reccursion in it. It is because of this difficulty that makes us try to find some other approaches which have more parallelism.

To this end, let us first give an efficient procedure to compute the power of a matrix. Let's look at an example, where $A \epsilon R^{m \times m}$ and we want to compute A^5. The trivial way for this is to multiply A 5 times, hence 5 matrix-matrix multiplications are needed. On the other hand, since $A^5 = A^4 A$, if A^2 is computed, then $A^4 = (A^2)^2$, A^5 can be computed using only 3 matrix-matrix mutiplications. Formalizing this idea for any N we have

Algorithm 3 (BINEV) Given A and N, this algorithm computes A^N and stores it in P.

Let $N = \sum_{k=0}^{t} \beta_k 2^k$ be the binary expansion of N with $\beta_t \neq 0$.
$S = A$; $q = 0$
while $\beta_q = 0$
    $S = S^2$; $q = q + 1$
end
$P = S$
for $k = q + 1 : t$
    $S = S^2$
    if $\beta_k \neq 0$
        $P = PS$

```
    endif
end
```

Fig.3 BINEV

This algorithm can be found in Golub and Van Loan [7] (p. 553). This algorithm requires at most $2[log_2(N)]$ matrix multiplies, and if N is a power of 2, it requires only $log_2(N)$ matrix multiplies.

Next, the idea involved in $BINEV$ is used to devise a sequential algorithm for evaluating $G_N(R)$, which is just a combination of matrix powers. Let us assume that N is a power of 2 from here on. Note that R^{2^j}, $j = 0, \cdots, log_2(N)$ are sufficient to form all the powers R^i, $i = 1, \cdots, N$ if we utilize the binary representations of 1 to N. The following algorithm is designed so that it fully reuses the powers computed, hence reduces the number of matrix multiplies. In the algorithm, β_i is the array to store the binary representation of i, P_i is used to store $R^i A_i$, and $binary(i)$ is a function which returns the binary representation of i.

Algorithm 4 (FEVAL) Given R, A_i, $i = 0 : N$, this algorithm computes $G_N(R)$.

```
{ Initialization }
P = A_0
for i = 1 : N
    β_i ← binary(i)
    P_i ← A_i
end
{ ComputePowers }
for j = 1 : log_2(N)
    for i = 1 : N
        if β_i(j) ≠ 0
            P_i ← RP_i
        endif
    end
    R ← R^2
end
{ CollectPowers }
for i = 1 : N
```

$P \leftarrow P + P_i$

end

$G_N(R) = P$

Fig.4 FEVAL

Note that exactly half of the numbers $1 : N$ have 1's in the jth bit of their binary representations ($j = 1 : log_2(N)$), so it is easy to see that this algorithm needs $\frac{1}{2}Nlog_2(N) + log_2(N)$ matrix multiplications. Apparently, $FEVAL$ is inferior to $HEVAL$ on a sequential machine, but there are more parallelism in $FEVAL$ than in $HEVAL$.

To design an efficient parallel algorithm on a hypecube type architecture, several issues like data distrubution, node communication and load balancing have to be put into consideration.

Suppose we have a hypercube system with p processors, each of which has a unique name *myid*. Without loss of generality, we can assume that there are $2^l + 1$ terms in $G_N(R)$, i.e., $N = 2^l$. Let us also assume that $N \geq p$.

We would like to arrange the storage for all A_i's such that an efficient parallel algorithm for evaluating $G_N(R)$ can be obtained. It turns out that storing A_i's in wraped fashion meets our goal, that is

$$A_i \textit{ is assigned to node } j \textit{ , where } i = j \ mod(p), \quad j = 0 : p - 1. \tag{10}$$

It is also assumed that a copy of R is stored in node 0.

The following is an outline of our algorithm:

(1) node j, $j = 0 : p - 1$ is responsible for computing terms $S_j = \sum_{k \epsilon J} R^k A_k$, where J denotes the index set of A_i's stored in node j;

(2) these partial sums S_j are then routed to node 0 through a *fan-in* procedure and final result $G_N(R) = \sum_{i=0}^{N} S_i$ is thus obtained at node 0.

Part (1) can be further divided into four steps, except for step (a), all three other steps are done in parallel:

(a) node 0 broadcast R to all other nodes;

(b) node j, $j = 0 : p - 1$ computes R^j and R^p, using the idea of algorithm $BINEV$, only $log_2(p)$ number of matrix multiplications are required. Communications between nodes are needed in this step;

(c) each node j employes a Horner's type procedure and computes its own matrix polynomial $S_j/R^j = A_j + R^p A_{p+j} + \cdots + R^{p(N/p-1)} A_{p(N/p-1)+j}$ using

the matrix power R^p computed in step (b). The main computations of the algorithm are concentrated in this step and no commucations between nodes are needed;

(d) multiplying the resulting matrix S_j/R^j with the matrix power R^j that has been calculated in step (b) completes the calculation of S_j in node j, $j = 0 : p-1$.

Finally, the *fan-in* of S_j in part (2) is achieved through the spanning tree rooted at node 0.

The detailed algorithm is given on next page.

Algorithm 5 (PEVAL) If each node executes the following node program, then $G_N(R)$ is computed in node 0.

$h = N/p$ {number of sweeps}
{initialization}
$P = I$, $S = A_{(h-1)p+myid}$ ($S = A_{hp}$ for node 0)
node 0: broadcast(R)
{sweep 1, compute matrices R^{myid} and R^p}
for $j = 1 : log_2(p)$
  if $binary(myid)(j) = 0$ **then**
    $R \leftarrow R^2$
    $send(R) \rightarrow myid + 2^{j-1}$
  else
    $P \leftarrow RP$
    $recv(R)$
  endif
end
{other sweeps, compute S_{myid}/R^{myid}}
for $k = 2 : h$
  if $myid = 0$ **then**
    $S \leftarrow RS + A_{(h-k-1)p}$
  else
    $S \leftarrow RS + A_{(h-k)p+myid}$
  endif
end
{final sweep, compute S_{myid}}
if $myid = 0$ **then**
  $S \leftarrow RS + A_0$
else
  $S \leftarrow PS$
endif
{collect results to node 0}
$fan\text{-}in(S)$

Fig.5 PEVAL

It is easy to see that the time PEVAL spends in computation is

$$T_{comp} \approx [\frac{N}{p} + log_2(p)](m^3 + m^2)$$

and the time it spends in communication is

$$T_{comm} \approx 3log_2(p)(\alpha + \beta m^2)$$

the total time for PEVAL is then

$$T_{peval} \approx [\frac{N}{p} + log_2(p)]m^3 + [\frac{N}{p} + (3\beta + 1)log_2(p)]m^2$$

Hence the speedup of PEVAL over HEVAL is approximately

$$sp = \frac{T_{heval}}{T_{peval}} \approx p(\frac{N}{N + plog_2(p)})$$

From this formula we can see that the speedup is always greater than one, and increases with N as well as p. When N tends to infinite, the speedup tends to p, the best we could hope for. Furthermore, the efficiency of the algorithm is $N/[N + plog_2(p)]$, which increases with N and tends to one as N goes to infinite. But, the efficiency decreases with p, which is a general phenomenon in parallel computations.

Communications between nodes are essentially eliminated except in the first sweep and fan-in/fan-out parts of the algorithm. Again, this is *almost* the ideal situation in designing a parallel algorithm as far as communications are concerned.

Finally, load balancing among nodes is also achieved in our algorithm since essentially each node has the same amount of work load.

In conclusion, our parallel algorithm $PEVAL$ is very efficient and is optimal in the sense that it achieves a linear speedup.

The next expensive computation in Algorithm 1 is the computation of $(I - A_1)^{-1}$ at the start. It is quite straightforward to design a parallel algorithm for this purpose. To compute the inverse, the Gaussian elimination of $I - A_1 = LU$ is first computed in parallel by all the processors. There have been several efficient parallel algorithms on hypercubes for Gaussian elimination which can achieve near linear speedup, for example, see [5], [6]. Then, node

0 broadcast a copy of L,U to node j, $j = 0 : p-1$, which in turn computes columns i, $i = j \ (mod\ p)$of the inverse. And then all the node fan-in their results to node 0. The following is the detail of the algorithm.

Algorithm 6 (PINV) if all nodes execute the following node program, then $(I - A_1)^{-1}$ is computed at node 0 upon termination.

compute L U
if $myid = 0$ **then**
    $broadcast(L, U)$
else
    $recv(L, U) \rightarrow S$
endif
for $i = myid : p : m$
    solve $Ly = e_i$
    solve $Ux = y$
    $fan\text{-}in(x)$ to node 0
end

Fig.6 PINV

An easy computation gives the time for PINV is

$$T_{pinv} \approx \frac{8}{3p} m^3.$$

Since the time of an efficient sequential algorithm for computing $(I - A_1)^{-1}$ via Gaussian elimination is approximately $\frac{8}{3}m^3$, it is evident that the speedup of $PINV$ is almost linear.

Putting together all the ingradients, we have the following parallel version of Ramaswami's algorithm for nonlinear matrix equations.

Algorithm 7 Given A_i and N, if each node executes the following node program, the solution R to the matrix equation $X = \sum_{i=0}^{\infty} X^i A_i$ is computed at node 0.

{initialization}
$R = 0$
{compute $(I - A_1)^{-1}$ in parallel}
call PINV

Repeat
{evaluate $G_N(R)$ in parallel}
call PEVAL
{some lower order work}
node 0,1 and 2: compute step length matrix S
{node 0 updates next iterate R}
if $myid = 0$: $R \leftarrow R + S$
Until

Fig.7

As we have discussed before, the speedup of Algorithm 7 can be linear when N and m are large. Hence we can conclude that we have achieved the goal of designing an efficient parallel algorithm for the matrix equation (1).

5 Numerical Experiments

Since *PEVAL* is the dominant work in algorithm 7 and since *PINV* has been extensively analyzed and experimented by other researchers, we have only performed some numerical experiments with *PEVAL* on the 32-node Intel iPSC2 hypercube at Cornell Theory Center, running version 3.1 iPSC2 system software. The numerical results are listed in Tables 1-4.

In Table 1, we list the results of the speedup of algorithm *PEVAL* run on different cube sizes ranging from 2 nodes to the full cube (32 nodes). As expected, the ideal speedup are achieved (or almost) for runs on small cubes since the communication overheads are very small compared with the computation times. In order to achieve the same speedup on large cubes, we need to increase polynomial degree N so that the same low ratio of (communication overhead)/(computation time) observed on small cubes can be achieved. However, this is impractical for the hypecube machine we used since it would take too long for the sequential algorithm *HEVAL* to finish. Instead, we have ploted the projected speedup curves in *Fig.*8. Also, it is observed in Table 1 that as matrix size m incerases, the corresponding speedup also increases. This is what we have expected. Since as matrix size increases, the communication/computation time ratio drops (see Table 2,3), this results an increase in the speedup.

In Table 2 and Table 3, the computation and communication timing results and their ratio for runs on 2 nodes and 32 nodes are listed. As we mentioned before, while these ratios are extremely low, the ratio for 32 nodes for any combination of m, N is noticeably higher than that for 2 nodes. We also notice that this ratio drops as matrix size increases, although it becomes less and less noticeable.

Finally, in Table 4, the individual node timing is listed for run on 16 nodes with matrix size and polynomial degree equal 64 and 1024 respectively. For reference purposes, the individual node fan-in time is also included there. It is clear from this table that load balancing is achieved among all nodes of the cube.

In conclusion, these numerical results have comfirmed all of our therical analysis/results about algorithm $PEVAL$ of the last section, hence our goal of designing an efficient algorithm for the nonlinear matrix equation on hypecube is achieved.

6 Acknowledgement

We thank Cornell University's Theory Center and Advanced Computing Research Institute for the use of the Intel iPSC2 hypercube computer and in particular their staff members Roslyn Leibensperger and Douglas Elias for their help in the programming of this algorithm.

References

[1] M. F. Neuts, "Matrix-analytic methods in queueing theory," *European J. Oper. Res., 15 (1984), pp. 2-12.*

[2] M. F. Neuts, "Matrix-Geometric Solutions in Stochastic Models - An Algorithmic Approach," *The Johns Hopkins Univ. Press, Baltimore, London, 1981.*

[3] V. Ramaswami, "Nonlinear matrix equations in applied probability-solution techniques and open problems. " *SIAM Review, Vol. 30, Num. 2, June 1988.*

[4] Youcef Saad and Martin H. Schultz. "Topological Properties of Hypercubes," *Research Report YALEU/DCS/RR-389, June 1985.*

[5] E. Chu and A. George, "Gaussian elimination with partial piviting and load balancing on a multiprocessor", *Tech. Rept., Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1986.*

[6] G. A. Geist, "Efficient parallel LU factorization with piviting on a hypercube multiprocessor", *Tech. Rept. ORNL-6211, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1985.*

[7] G.H.Golub and C. Van Loan. *Matrix Computations, The Johns Hopkins University Press,Baltimore,MD.,1989.*

matrix size (m)	polynomial degree (N)	2 (nodes)	4 (nodes)	8 (nodes)	16 (nodes)	32 (nodes)
8	64	1.86	3.11	4.33	4.90	4.82
8	128	1.93	3.51	5.60	7.49	8.40
8	256	1.96	3.73	6.60	10.22	13.20
8	512	1.98	3.86	7.23	12.49	18.79
8	1024	1.99	3.93	7.59	14.03	23.65
16	64	1.92	3.40	5.24	6.65	7.09
16	128	1.96	3.68	6.33	9.38	11.60
16	256	1.98	3.83	7.07	11.85	17.02
16	512	1.99	3.92	7.51	13.61	22.74
16	1024	2.00	3.96	7.75	14.71	26.25
32	64	1.94	3.51	5.60	7.45	8.25
32	128	1.97	3.74	6.60	10.17	13.13
32	256	1.99	3.88	7.24	12.46	18.64
32	512	2.00	3.94	7.62	14.03	23.56
32	1024	2.00	3.98	7.82	14.98	27.20
64	64	1.94	3.53	5.70	7.69	8.65
64	128	1.97	3.75	6.66	10.39	13.62
64	256	1.98	3.87	7.27	12.60	19.11
64	512	1.99	3.94	7.62	14.10	23.94
64	1024	2.00	3.97	7.81	15.00	27.39

Table 1: SPEEDUP OF ALGORITHM PEVAL

matrix size (m)	polynomial degree (N)	total time(ms)	comp. time(ms)	comm. time(ms)	comm./comp. ratio
8	64	298	288	10	0.034722
8	128	575	566	9	0.015901
8	256	1129	1118	11	0.009839
8	512	2237	2227	10	0.004490
8	1024	4454	4444	10	0.002250
16	64	1945	1928	17	0.008817
16	128	3808	3790	18	0.004749
16	256	7534	7516	18	0.002395
16	512	14984	14966	18	0.001203
16	1024	29886	29870	16	0.000536
32	64	14111	14077	34	0.002415
32	128	27729	27695	34	0.001228
32	256	54965	54932	33	0.000601
32	512	109436	109402	34	0.000311
32	1024	218379	218349	30	0.000137
64	64	108153	108098	55	0.000509
64	128	212771	212715	56	0.000263
64	256	422007	421951	56	0.000133
64	512	840468	840412	56	0.000067
64	1024	1677413	1677356	57	0.000034

Table 2: TIME AND RATIO OF PEVAL RUN ON 2 NODES

matrix size (m)	polynomial degree (N)	total time(ms)	comp. time(ms)	comm. time(ms)	comm./comp. ratio
8	64	115	68	47	0.691136
8	128	132	83	49	0.590361
8	256	168	119	49	0.411765
8	512	236	189	47	0.248677
8	1024	375	326	49	0.150307
16	64	526	431	95	0.220418
16	128	643	545	98	0.179817
16	256	876	779	97	0.124519
16	512	1341	1244	97	0.077974
16	1024	2272	2178	94	0.043159
32	64	3314	3128	186	0.059463
32	128	4164	3977	187	0.047020
32	256	5867	5679	188	0.033104
32	512	9272	9085	187	0.020583
32	1024	16081	15893	188	0.011829
64	64	24205	23855	350	0.014672
64	128	30744	30390	354	0.011649
64	256	43821	43470	351	0.008075
64	512	69974	69619	355	0.005099
64	1024	122281	121929	352	0.002887

Table 3: TIME AND RATIO OF PEVAL RUN ON 32 NODES

node number	total time(ms)	comp. time(ms)	comm. time(ms)	fanin. time(ms)
0	222545	222273	272	212
1	222526	222210	316	211
2	222464	222182	282	150
3	222467	222181	286	149
4	222399	222138	261	92
5	222409	222163	246	84
6	222410	222160	250	86
7	222412	222161	251	85
8	222338	222123	215	25
9	222341	222117	224	24
10	222341	222114	227	24
11	222351	222128	223	24
12	222343	222130	213	24
13	222353	222128	225	24
14	2223543	222129	225	24
15	222358	222137	221	24

Table 4: INDIVIDUAL NODE TIME OF PEVAL RUN ON 16 NODES (matrix size = 64, polynomial degree = 1024)

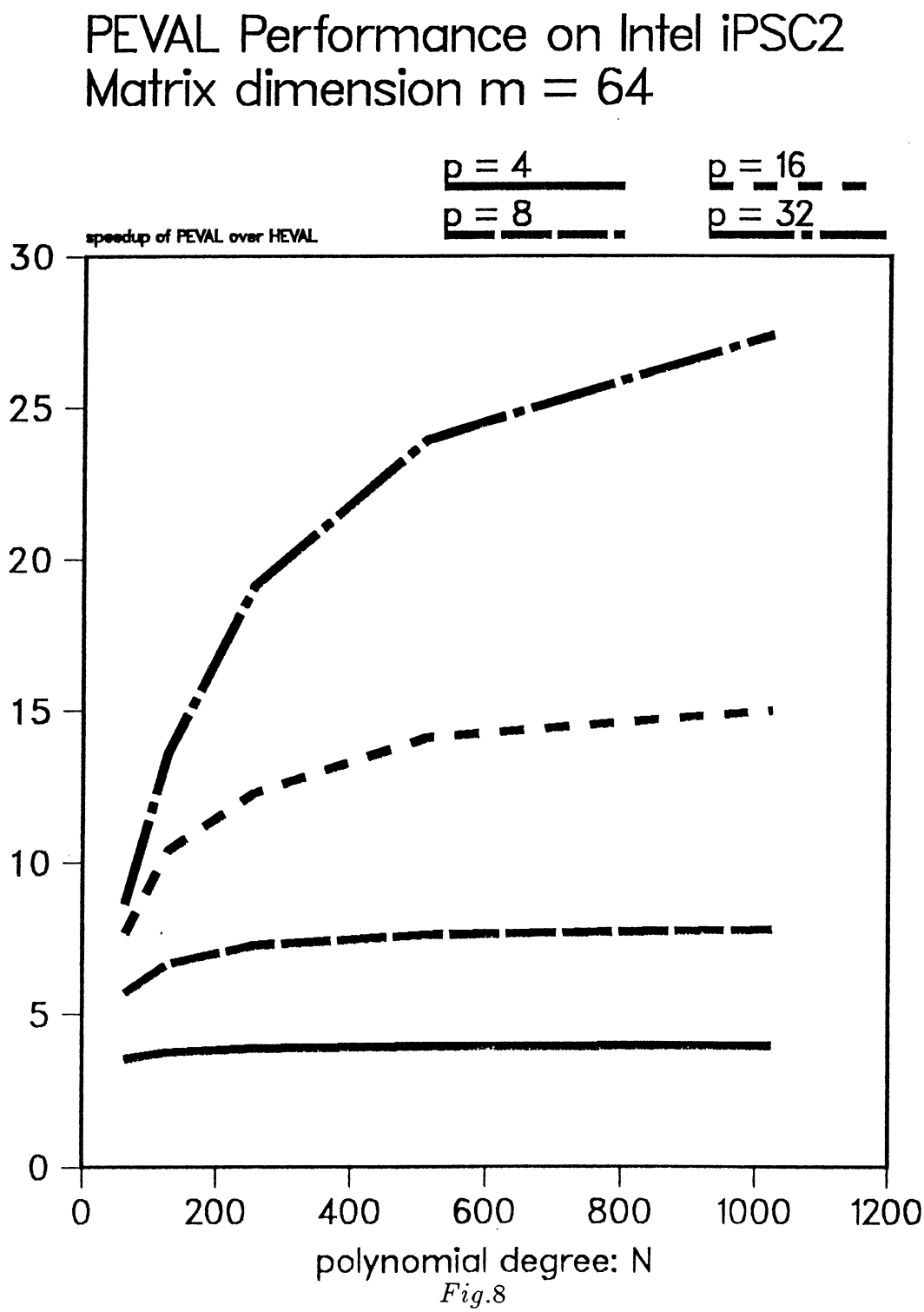
PEVAL Performance on Intel iPSC2
Matrix dimension m = 64
p = 4
p = 8
p = 16
p = 32
speedup of PEVAL over HEVAL
30
25
20
15
10
5
0
0
200
400
600
800
1000
1200
polynomial degree: N

Fig.8