1986

ADI Methods on a Shared Memory Machine

John P. Bonomo

Paul E. Buis

Wayne R. Dyksen

Report Number:

86-622

TR- 622
wrd
sent
11/21/88

ADI Methods on a Shared Memory Machine

John P. Bonomo [1]

Paul E. Buis [2]

Wayne R. Dyksen [3]

Department of Computer Sciences

Purdue University

CAPO Report CER-88-622

[1]Work supported in part by the Air Force of Scientific Research under Grant 84-0385

[2]Work supported in part by the National Science Foundation under Grant CCR-8619817

[3]Work supported in part by the National Science Foundation under Grant ~~DCR-8602385~~ CCR-8612676

ADI Methods on a Shared Memory Machine

John P. Bonomo
Paul E. Buis
Wayne R. Dyksen

November 18, 1988

Abstract

This paper summarizes the results obtained using various sequential and parallel methods to solve partial differential equations on a shared main memory machine, the Sequent Symmetry. Four numerical methods are used and compared: 1) sequential band Gauss elimination, 2) parallel band Gauss elimination, 3) sequential Tensor Product Generalized ADI, and 4) parallel TPGADI. We discuss the various issues involved in the parallelization of a sequential algorithm to make best use of the Sequent Symmetry.

Keywords: elliptic partial differential equations, collocation, Alternating Direction Indirect method, tensor products, parallel computation, shared memory machines.

AMS(MOS) subject classifications: 15A06, 15A69, 35J15, 65F10, 65N35, 65W05, 68Q25.

1 Introduction

Parallel algorithms for numerically solving partial differential equations (PDEs) have been designed for many years, but with the increased availability of multi-processor machines the analysis of these algorithms has moved from the theoretical to the experimental arena. This paper summarizes the results obtained using various methods to solve PDEs on a shared main memory machine, the Sequent Symmetry. Four numerical methods are used and compared: 1) sequential band Gauss elimination, 2) parallel band Gauss elimination, 3) sequential Tensor Product Generalized ADI (TPGADI), and 4) parallel TPGADI. We begin in Section 2 with a brief description of the Sequent Symmetry. In Section 3 we discuss tensor products of matrices and analyze the time and space requirements of tensor product equations. We describe the TPGADI methods in Section 4 and a specific instance of them using Hermite bicubics in Section 5. In Section 6 we consider the issues involved in the parallelization of the Gauss Elimination and TPGADI algorithms. Section 7 contains the experimental results. Finally, in Section 8 we summarize these results.

2 Parallel Processing on the Sequent Symmetry

The Sequent Symmetry is a shared main memory multiprocessor machine built by Sequent Computer Systems, Inc. Each processor board contains two CPUs, each consisting of an Intel 80386 32-bit microprocessor with an 80387 floating point processor running at 16-MHz and 64 Kbytes of system managed local memory. A Weitek 1167 floating point accelerator is optional, but was not used in our experiment. Up to 15 of these processor boards may be installed, along with 40 megabyte memory expansion cards, a SCSI board with an Ethernet interface, and a Multibus card cage with 12 slots. All of these boards are connected to one 10 MHz system bus with a theoretical bandwidth of 80 Mb/sec. Special bus signals are used to enable the locking of any 4 bytes of memory for mutual exclusion purposes.

The Sequent Symmetry runs the DYNIX operating system, also developed by Sequent Computer Systems, Inc. DYNIX is a variant of UNIX BSD 4.2 with System V features supported via a mechanism where the user can choose which UNIX system to emulate. In addition to the usual system calls, DYNIX provides concurrent processing support in the form of a library of *microtasking* calls. This library contains a concurrent process creation routine, m_fork(), and a variety of concurrent process coordination routines based on the hardware locks. This library can be accessed directly from C and through intrinsic calls in other languages. We use the FORTRAN compiler by Silicon Valley Software which provides a preprocessor which transforms marked DO loops into an m_fork() call to a subroutine containing the body of the loop and all of the required process coordination code. For example, the following code for matrix addition

```
      subroutine matadd(a,b,c,nrowmx,nrows,ncols)
      real a(nrowmx,1), b(nrowmx,1), c(nrowmx,1)
c$doacross share(c, a, b, ncols, nrows), local(i)
      do 10 j = 1, ncols
         do 10 i = 1, nrows
            c(i,j) = a(i,j) + b(i,j)
   10 continue
      return
      end
```

is transformed into code which is essentially equivalent to

```
      subroutine matadd(a,b,c,nrowmx,nrows,ncols)
      external do10%matadd
      integer m_fork,nrowmx
      real a(nrowmx,1), b(nrowmx,1), c(nrowmx,1)
      if (m_fork(do10%matadd,ncols,nrowmx,nrows,b,a,c,j) .ne.0)
     *  stop 'do 10 matadd'
      return
      end
      subroutine do10%matadd(ncols,nrowmx,nrows,b,a,c,j%)
      integer m_get_myid,m_get_numprocs
      integer j,i,nrowmx,nrows
      real b(nrowmx,1),a(nrowmx,1),c(nrowmx,1)
      do 99997, j = j, ncols, m_get_numprocs()
         do 10 i = 1, nrows
            c(i,j) = a(i,j) + b(i,j)
   10 continue
99997 continue
      end
```

A call to m_fork() creates a number of processes which is at most one less than the number of processors. These processes are not tied to any particular processor, but like any other process are moved to the least busy processor whenever they are rescheduled by the operating system. Processes created with m_fork() share some of the memory associated with the parent process, unlike processes created with the traditional fork() system call which obtain copies of the parent's memory. After process creation, m_fork() starts the same subroutine with the same arguments running in each of these processes. Each of these subroutines is able to tell which process it is associated with by calling m_get_myid(). When the subroutine terminates, the process busy waits until it is needed again by the next m_fork() call. This reuse of processes by m_fork() reduces process creation overhead; however, if the processes are not needed again, it is wasteful of CPU cycles. These processes can be terminated by a call to m_kill() or suspended by a call to m_park_procs().

3 Tensor Products of Matrices

Let $A = \{a_{mn}\}$ and $B = \{b_{kl}\}$ be matrices of order $M \times N$ and $K \times L$, respectively. The *tensor product* of A and B, denoted by $A \otimes B$, is the matrix of order $MK \times NL$ given by

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1N}B \\ a_{21}B & a_{22}B & \dots & a_{2N}B \\ \vdots & \vdots & & \vdots \\ a_{M1}B & a_{M2}B & \dots & a_{MN}B \end{bmatrix}.$$

A detailed account of the properties of tensor products is given in [Hal58].

The main work done in both Gauss elimination and the TPGADI algorithms involves solving equations of the form

$$(A_1 \otimes B_2 + B_1 \otimes A_2)\mathbf{x} = \mathbf{b}.$$

In the next section we will see that matrices with this structure can be solved iteratively by repeatedly solving equations of the form $(A \otimes B)\mathbf{x} = D$ and performing multiplications $(A \otimes B)\mathbf{x}$. The TPGADI takes advantage of the particular matrix structure by using efficient algorithms for manipulation of tensor products.

To take advantage of these algorithms we need use only the factors A and B, and can avoid explicitly forming the tensor product $A \otimes B$. When working with $(A \otimes B)\mathbf{x}$, one can take advantage of the fact that $\Re^{NL}$ and $\Re^{N \times L}$ are isometric by representing the vector $\mathbf{x}$ by the matrix $X = \{x_{ln}\}$ defined by the isometry

$$x_{ln} = \mathbf{x}_{l+L(n-1)}.$$

The usefulness of this representation can be seen in the following simplifications of the more general results given in [dB79,PS73] which give efficient procedures for computing $(A \otimes B)\mathbf{x}$ and solving $(A_1 \otimes A_2)\mathbf{x} = \mathbf{b}$, respectively.

LEMMA. Let $A = \{a_{mn}\}$, $B = \{b_{kl}\}$ and $X = \{x_{ln}\}$ be matrices of order $M \times N$, $K \times L$ and $L \times N$, respectively. Then the $K \times M$ matrix $(A \otimes B)X$ is given by

$$(A \otimes B)X = (A(BX)^T)^T.$$

COROLLARY. Let A_k be matrices of order $N_k \times N_k$, let X and B be matrices of order $N_2 \times N_1$, and consider the linear system

$$(A_1 \otimes A_2)X = B.$$

If A_1^{-1} and A_2^{-1} exist, and if $A_2 Y = B$ and $A_1 Z = Y^T$, then $X = Z^T$.

4 The Tensor Product Generalized ADI Method

Let A_k and B_k be matrices of order $N_k \times N_k$, and consider the linear system

$$(A_1 \otimes B_2 + B_1 \otimes A_2)C = F. \tag{1}$$

While the tensor product $(A_1 \otimes B_2 + B_1 \otimes A_2)$ is an $N_1 N_2 \times N_1 N_2$ matrix, we wish to solve (1) by computing only with A_1, B_1 and A_2, B_2; that is, we wish to solve the two-directional problem (1) by using methods employed to solve the one-directional problems. We use the term *directional* rather than *dimensional* since one direction may encompass more than one dimension, as in the Method of Planes [Dyk86,Dyk88].

For a given set of positive *acceleration parameters* ρ_l, $l = 1, 2, \ldots$ the two-directional *Tensor Product Generalized Alternating Direction Implicit* (TPGADI) iteration method is defined by

$$\begin{aligned} C^{(0)} & \text{ given} \\ [(A_1 + \rho_{k+1} B_1) \otimes B_2] C^{(k+1/2)} &= F - [B_1 \otimes (A_2 - \rho_{k+1} B_2)] C^{(k)} \\ [B_1 \otimes (A_2 + \rho_{k+1} B_2)] C^{(k+1)} &= F - [(A_1 - \rho_{k+1} B_1) \otimes B_2] C^{(k+1/2)}. \end{aligned} \tag{2}$$

The TPGADI method converges to a unique solution C if the matrices $B_1^{-1}A_1$ and $B_2^{-1}A_2$ have a complete set of normalized eigenvectors with corresponding positive eigenvalues λ_i and μ_j, respectively [Dyk87]. Furthermore, if the acceleration parameters ρ_l are set equal to λ_l (μ_l), then the TPGADI method is an exact method (except for roundoff) in N_1 (N_2) iterations.

In order to compare the TPGADI method to other schemes, we summarize the computer time (via operation counts) and computer memory required to implement it. We assume that A_k and B_k are band matrices with bandwidth K_k and that all systems of linear equations are solved by band Gauss elimination with partial pivoting. Since the initial guess $C^{(0)}$ and the acceleration parameters ρ_l depend on the discretization method used, we assume here that they are given.

The work to compute the 1-direction sweep of the TPGADI Method (2) is summarized in Table 1. Thus, the total work to compute the 1-direction sweep is $O\left(N_1N_2(5(K_1+K_2)+1/2)\right)$. An analogous estimate shows that the work for the 2-direction sweep is the same. Hence, the total work per iteration is $O\left(N_1N_2(10(K_1+K_2)+1)\right)$ operations. If $N_1 = N_2 = N$ and $K_1 = K_2 = K$, then this work estimate simplifies to $O(20KN^2)$. The TPGADI iterative method can be a direct method in N iterations, requiring $O(20KN^3)$ operations. Note that the dominant work in the TPGADI method does *not* result from factoring W_1 or B_2. Instead, the dominant work involves computing the right side W and doing multiple back substitutions solving for $C^{(k+1/2)}$.

Consider the straight forward method of applying band Gauss elimination to the matrix $A = (A_1 \otimes B_2 + B_1 \otimes A_2)$. If $N_1 = N_2 = N$ and $K_1 = K_2 = K$, then the matrix A is of order $N^2 \times N^2$ with bandwidth $KN - N + K$. Band Gauss elimination with partial pivoting applied to it requires $O(2K^2N^4)$ operations to perform the LU factorization, and $O(3KN^3)$ operations to perform the forward and back substitutions.

Thus, even as a direct method the TPGADI method is asymptotically much faster than straight forward Gauss elimination as a direct method of solution. This conclusion warrants a few remarks. First, in order for the TPGADI method to be direct we must either know *a priori* the eigenvalues of $B_1^{-1}A_1$ or $B_2^{-1}A_2$ or we must compute them. In many applications arising from PDEs these eigenvalues are known explicitly. Secondly, one would almost never use the TPGADI method as a direct method in practice. Given the desired eigenvalues, we would use some subset of them to achieve moderate accuracy with many fewer than N iterations. In particular, since the low-frequency components of the error are associated with the smallest eigenvalues, using only a few in

Operation	Work
$W_2 = A_2 - \rho_{k+1} B_2$	$2K_2N_2$
$W = (B_1 \otimes W_2)C^{(k)}$	$2N_1N_2(K_1 + K_2)$
$W = F - W$	$1/2N_1N_2$
$W_1 = A_1 + \rho_{k+1} B_1$	$2K_1N_1$
Factor B_2	$2K_2^2N_2$
Solve $L_2U_2Y = W$	$3N_1K_2N_2$
Factor W_1	$2K_1^2N_1$
Solve $L_1U_1(C^{k+1/2})^T = Y^T$	$3K_1N_1N_2$

Table 1: Work to compute the 1-direction sweep of the TPGADI method

increasing order will often suffice [Lyn68].

A simple calculation shows that the amount of memory required to factor the matrix $(A_1 \otimes B_2 + B_1 \otimes A_2)$ by Gauss elimination with partial pivoting is $O(3KN^3 + 2N^2)$ words. The memory requirements for the TPGADI method are estimated as follows: A_1, B_1, A_2, B_2 each require $O(2KN)$ words; W_1 and W_2 also require $O(2KN)$ words; and W, F and C each require N^2 words. Thus, the total amount of computer memory required is $O(3N^2)$ words, which is nearly optimal since it is the same order of magnitude as the number, N^2, of unknowns.

5 TPGADI with Hermite Bicubics

We consider an elliptic problem of the form

$$\begin{aligned} L_xu + L_yu &= f \text{ in } \Omega = [0,1] \times [0,1] & (3) \\ u &= 0 \text{ on } \partial\Omega. & (4) \end{aligned}$$

where

$$\begin{aligned} L_xu &= -a_2(x)u_{xx} + a_1(x)u_x + a_0(x)u, \quad a_2 > 0, \\ L_yu &= -b_2(y)u_{yy} + b_1(y)u_y + b_0(y)u, \quad b_2 > 0. \end{aligned}$$

We assume for simplicity that we have homogeneous Dirichlet boundary conditions. The analysis is readily extended to problems with nonhomogeneous Dirichlet or Neumann boundary conditions [Dyk87,HMR85a,HMR85b].

The domain Ω is subdivided with a rectangular, tensor product grid with MN rectangles. We approximate $u(x,y)$ by

$$U(x,y) = \sum_{m=1}^{2M} \sum_{n=1}^{2N} c_{nm}\phi_m(x)\psi_n(y)$$

where ϕ_m and ψ_n are the standard one dimensional Hermite cubics with the grid lines as knots. The Hermite cubics which are zero on $\partial\Omega$ are discarded so that $U \equiv 0$ on $\partial\Omega$.

To determine the $4MN$ unknowns c_{nm}, we place in each subinterval (x_m, x_{m+1}) and (y_n, y_{n+1}), the two Gauss points $\tau_{2m+1} = \frac{1}{2}(x_m + x_{m+1}) - \frac{h_x}{2\sqrt{3}}$, $\tau_{2m+2} = \frac{1}{2}(x_m + x_{m+1}) + \frac{h_x}{2\sqrt{3}}$ and $v_{2n+1} = \frac{1}{2}(y_n + y_{n+1}) - \frac{h_y}{2\sqrt{3}}$, $v_{2n+2} = \frac{1}{2}(y_n + y_{n+1}) + \frac{h_y}{2\sqrt{3}}$. These collocation points give a fourth order discretization error for smooth problems [Hou78,PW80]. We then collocate the elliptic problem (4) at these $4MN$ points to obtain the *Hermite bicubic collocation equations*

$$L_x[U](\tau_i, v_j) + L_y[U](\tau_i, v_j) = f(\tau_i, v_j), \quad \begin{array}{l} i = 1, \ldots, 2M \\ j = 1, \ldots, 2N. \end{array} \tag{5}$$

The structure of the linear system in (5) depends on the ordering of the collocation points and the basis functions. If they are both ordered in a natural tensor product manner, then (5) may be written in tensor product form as

$$(A_x \otimes B_y + B_x \otimes A_y)C = F,$$

where

$$[A_x]_{im} = L_x\phi_m(\tau_i),\ [B_x]_{im} = \phi_m(\tau_i), \quad \begin{array}{l} i = 1, \ldots, 2M \\ m = 1, \ldots, 2M, \end{array}$$

$$[A_y]_{jn} = L_y\psi_n(v_j),\ [B_y]_{jn} = \psi_n(v_j), \quad \begin{array}{l} j = 1, \ldots, 2N \\ n = 1, \ldots, 2N, \end{array}$$

$$C_{nm} = c_{nm}, \quad \begin{array}{l} n = 1, \ldots, 2N \\ m = 1, \ldots, 2M, \end{array} \quad \text{and } F_{ji} = f(\tau_i, v_j), \quad \begin{array}{l} j = 1, \ldots, 2N \\ i = 1, \ldots, 2M. \end{array}$$

Since the support of each Hermite cubic ϕ_m and ψ_n spans at most two subintervals, it follows that A_x, B_x and A_y, B_y have bandwidth two, *regardless* of M or N.

Dyksen has shown that the TPGADI method (2) applied to these Hermite bicubic collocation equations converges [Dyk87]. In particular, he gives explicit formulas for the eigenvalues of $B_x^{-1}A_x$ and $B_y^{-1}A_y$, and shows that they are distinct, real and positive. Given these eigenvalues, the TPGADI method can be exact for this problem.

6 Parallelization of the Gauss Elimination and TPGADI Algorithms

For Gauss elimination, parallelization entails simply inserting a doacross directive in the code for the factorization step. The factorization step is the dominant time component for the sequential version of this algorithm. The forward and back substitution is not parallelized since it is possible only to parallelize a few instructions, and doing so is not worth the overhead expense.

For the TPGADI method, each of the steps in Table 1 except the factorization is parallelized. It suffices to use the doacross construct in Sequent FORTRAN. All of the parallelism comes from loops which have the property that each trip through the loop operates on different data. In an environment that did not contain the doacross construct, one could manually perform the same transformation that Silicon Valley Software's Sequent compiler uses as illustrated in Section 2. Also, if m_fork() did not reuse process(es) once they were created, equivalent behavior could have bee~~n~~n achieved by creating processes which each execute the main TPGADI loop and are coordinated by m_sync() calls. A version of our code using this technique has no significant time difference.

The Hermite ~~B~~icubic collocation is also parallelized, but the sequential version still takes so little time as to be insignificant compared to the solution phase of the parallel code. lc

7 Experimental Results

We carried out a series of timing experiments on a singly loaded Sequent Symmetry with 28 processors, and measured real time, not processor usage. Process creation time was counted as part of the solution phase. Also, we ran the program at maximum priority, disabled the automatic priority modification, and enlarged the maximum working set size parameter of the virtual memory system. These measures make it unlikely that a process will leave the processor it starts on or be interrupted by one of the operating system daemons.

For these experiments we used TPGADI with N iterations. This yields a direct method, but is somewhat naive since full accuracy may be achieved much sooner as described in Section 4.

Table 2 summarizes our results. It shows the speedups and efficiencies obtained versus number of processors and number of grid lines. The speedup is the time for the sequential version of the algorithms divided by the time for a parallel version of the algorithm. Efficiency is the speedup divided by the number of processors used. The values from Table 2 are displayed graphically in Figures 1 through 6. This shows that as N gets larger, the speedups obtained by the algorithms approach the optimal speedup. This is as expected since, as the size of the linear system to solve grows larger, the overhead of process creation time and the non-parallelizable but non-dominating parts of the algorithm become less significant, and vice versa.

Figures 1 and 2 show that the speedup obtained for a given N does not remain a constant factor of the optimal speedup; this results from the phenomenon that as more processors are used the process creation times becomes a larger percentage of the overall time. Also, since all processors access memory via a common bus a large number of active processors leads to bus congestion, reducing efficiency.

For the TPGADI solution phase, an equal distribution of the work among the processors (and hence increased efficiency) is attained if the number of processors evenly divides the number of linear equations. This is the reason there is a small upward bump in the efficiency curves near 16 processors in Figure 4. Also, for all but the smallest problems, all processors are used almost all the time. By contrast most steps of Gauss elimination leave several processors idle near the end of the submatrix update. Gauss elimination also requires more memory which increases process startup time since rather large page tables must be duplicated.

Since the TPGADI solution phase has one component that was not parallelized – the factorization of a matrix with half bandwidth 2 – the maximum obtainable efficiency can be calculated by estimating the work from Table 1 to be $(20.5N^2 + 16N)/(20.5N^2 + 8N + 8Np)$ where p is the number of processors and N is the number of unknowns. Thus for 65 grid lines and 27 processors the maximum attainable efficiency is 92 percent. We attain 73 percent, the difference being due to process creation overhead, bus bandwidth limitations, and synchronization overhead.

8 Conclusions

TPGADI outperforms Gauss elimination as expected by a factor of $O(N)$. Furthermore, it also uses a factor of $O(N)$ less memory and parallelizes more efficiently. TPGADI has proven to be effective in solving certain two and three dimensional partial differential equations [Dyk86,Dyk87,Dyk88]. We have shown that both theoretically and practically that it can be effectively parallelized. In the case of the Hermite Bicubic Collocation equations, it significantly outperforms band Gauss elimination lc
(the only alternative, since the equations have no special properties such as symmetry or self-adjointness). We conjecture that parallel TPGADI will be even more effective for three directional problems. It is a direct method in N iterations since each iteration of TPGADI eliminates N^2 components of the error. For three directional problems, these advantages will make computation of N eigenvalues worthwhile even when they are not known *a priori*. In the case of Hermite Bicubic lc
Collocation, a factor of $O(N^2)$ memory savings and a factor of $O(N^3)$ time savings will be achieved not storing and solving a N^3 by N^3 matrix with Gauss elimination

References

[dB79] C. de Boor. Efficient computer manipulation of tensor products. *ACM Trans. Math. Software*, 5(2):173–182, June 1979.

[Dyk86] W. R. Dyksen. A tensor product generalized ADI method for elliptic problems on cylindrical domains with holes. *J. Comp. Appl. Math.*, 16:43–58, 1986.

[Dyk87] W. R. Dyksen. Tensor product generalized ADI methods for elliptic problems. *SIAM J. Numer. Anal.*, 24(1):59–76, February 1987.

[Dyk88] W. R. Dyksen. A tensor product generalized ADI method for the method of planes. 1988. to appear, *Numerical Methods for Partial Differential Equations.*

[Hal58] P. R. Halmos. *Finite-Dimensional Vector Spaces.* D. Van Nostrand Company, Inc., Princeton, second edition, 1958.

[HMR85a] E. N. Houstis, W. R. Mitchell, and J. R. Rice. Algorithms INTCOL and HERMCOL: collocation on rectangular domains with bicubic hermite polynomials. *ACM Trans. Math. Software*, 11:416–418, 1985.

[HMR85b] E. N. Houstis, W. R. Mitchell, and J. R. Rice. Collocation software for second order elliptic partial differential equations. *ACM Trans. Math. Software*, 11:379–412, 1985.

[Hou78] E. N. Houstis. Collocation methods for linear elliptic problems. *BIT*, 18:301–310, 1978.

[Lyn68] R. E. Lynch and J. R. Rice. Convergence rates of ADI methods with smooth initial error. *Math. Comp.*, 22:311-355, 1968.

[PS73] V. Pereyra and G. Scherer. Efficient computer manipulation of tensor products with applications to multidimensional approximation. *Math. Comp.*, 27(123):595–605, July 1973.

[PW80] P. Percell and M. F. Weeler. A C^1 finite collocation method for elliptic equations. *SIAM J. Numer. Anal.*, 17:605–622, 1980.

	Gauss Elimination			TPGADI			
	Number of Grid Lines			Number of Grid Lines			
Processors	9	17	33	9	17	33	65
Sequential	3.02	40.4	591	1.55	12.2	98.5	798
1	3.06	40.8	593	1.58	12.5	99.6	803
	0.99	0.99	1.00	0.98	0.98	0.99	0.96
	98.6	98.9	100	98.0	98.0	99.0	99.3
2	1.93	23.4	319	0.89	6.58	51.3	407
	1.56	1.72	1.85	1.73	1.87	1.92	1.96
	78.2	86.1	92.6	86.7	93.3	96.1	98.0
4	1.34	14.0	177	0.53	3.48	26.3	206
	2.26	2.89	3.34	2.90	3.53	3.74	3.86
	56.4	72.3	83.5	72.6	88.2	93.6	96.5
8	1.19	9.55	107	0.39	1.98	13.8	105
	2.54	4.23	5.51	3.94	6.21	7.14	7.54
	31.7	64.8	68.9	49.3	77.7	89.3	94.3
12	1.17	7.71	80.1	0.45	1.65	10.7	74.9
	2.58	5.23	7.38	3.47	7.46	9.20	10.7
	21.5	43.6	61.5	28.9	62.2	76.6	88.8
16	1.33	7.58	72.6	0.42	1.33	7.77	57.1
	2.27	5.32	8.14	3.73	9.23	12.7	14.0
	14.2	33.3	50.9	23.3	57.7	79.3	87.4
20	1.38	6.60	60.4	0.48	1.40	7.86	51.7
	2.19	6.12	9.78	3.25	8.77	12.5	15.4
	10.9	30.6	48.9	16.3	43.8	62.7	77.2
24	1.52	6.65	57.3	0.54	1.47	6.44	45.7
	1.98	6.07	10.3	2.87	8.37	15.3	17.4
	8.27	25.3	43.0	12.0	34.9	63.7	72.8
27	1.62	6.47	54.1	0.58	1.52	6.55	40.2
	2.62	3.45	10.8	2.69	8.06	15.7	19.9
	6.91	23.1	40.1	9.95	29.9	55.7	73.6
VAX 8600				0.62	4.97	38.9	319

Table 2: Discretization and Solution Times (in seconds)/Speedup/Efficiencies for Parallel TPGADI

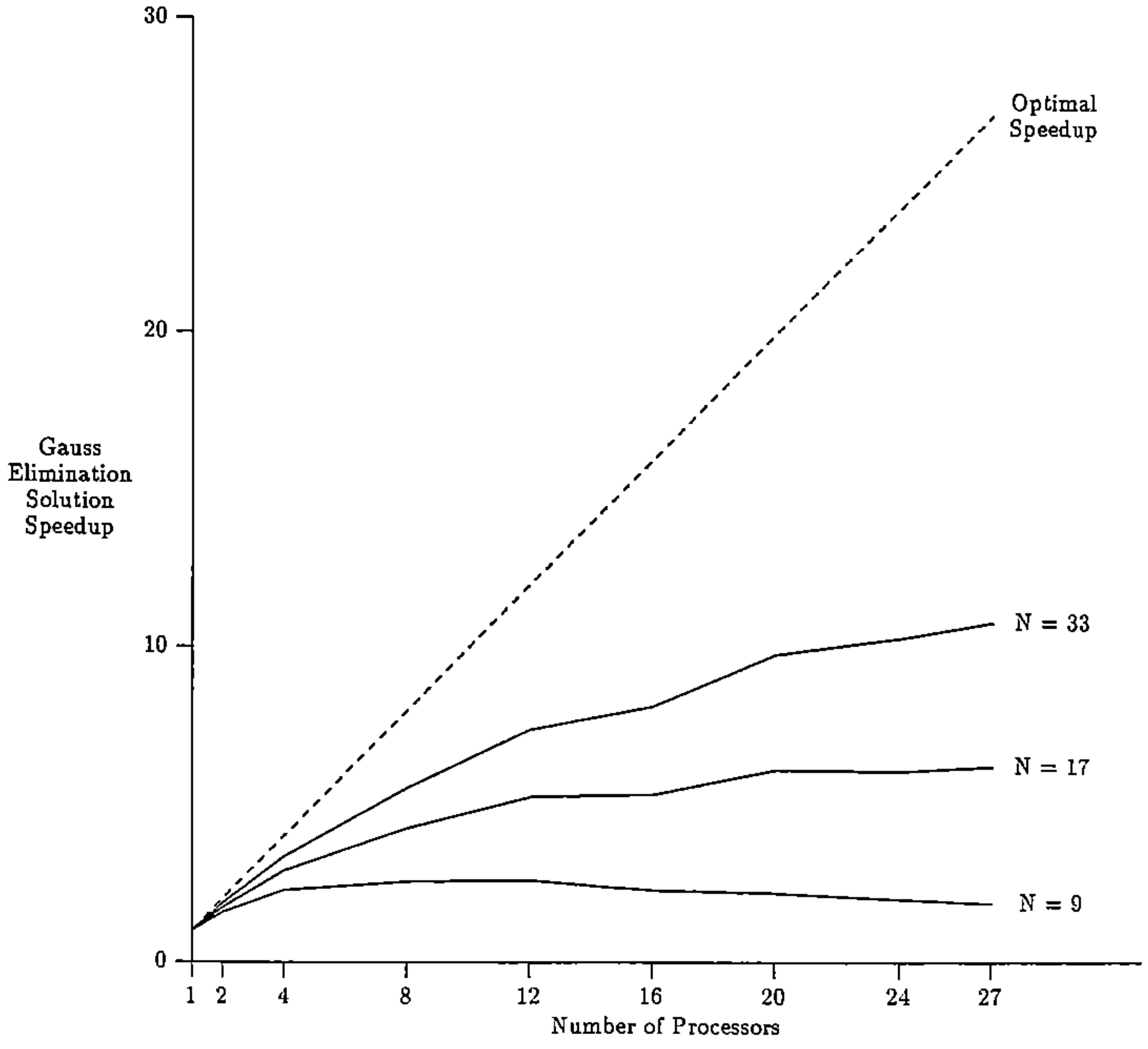

Figure 1: Speedups for Gauss Elimination. Note: N=65 case not tried because sequential version takes 2.5 hours

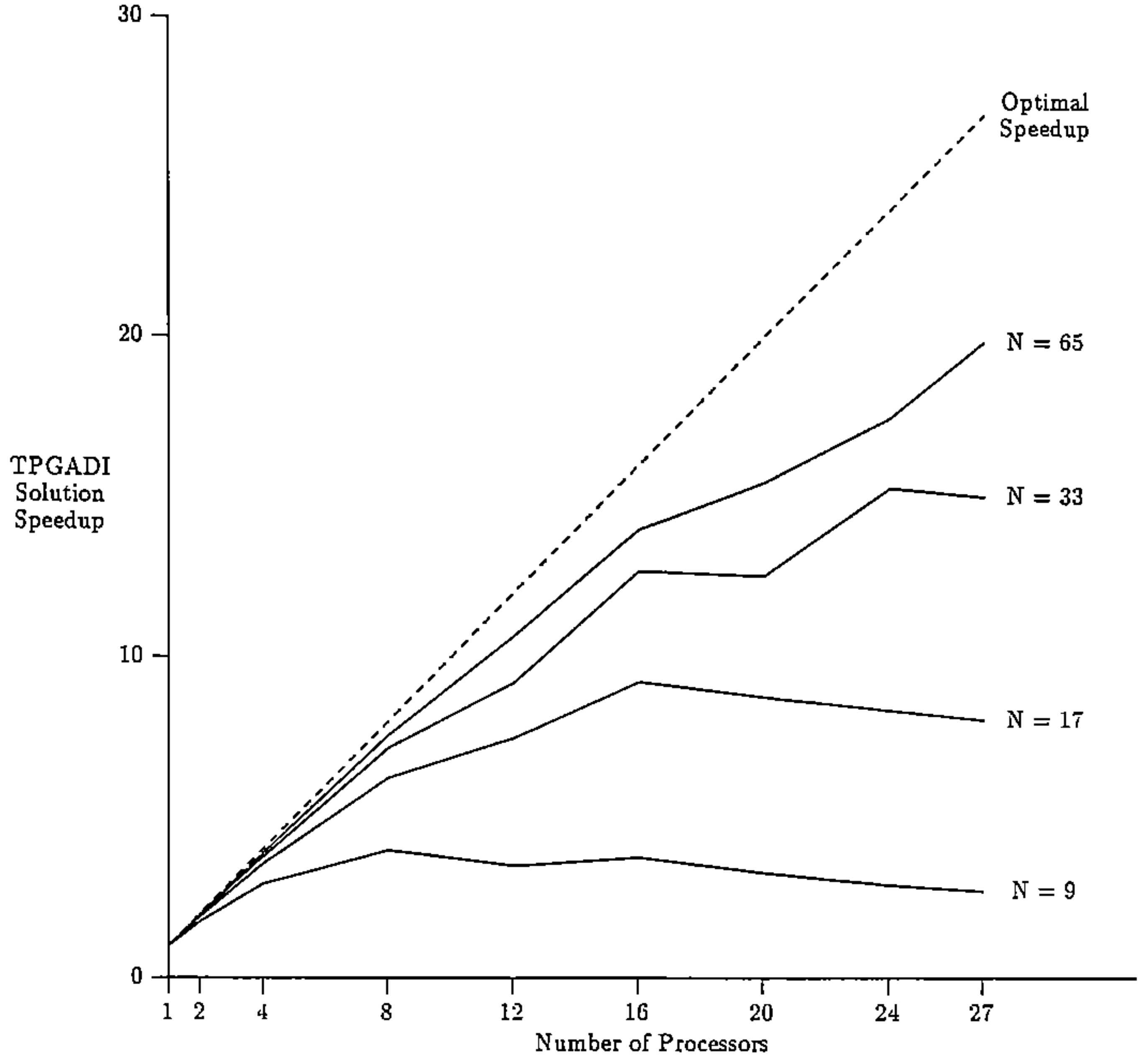

Figure 2: Speedups for TPGADI Solution

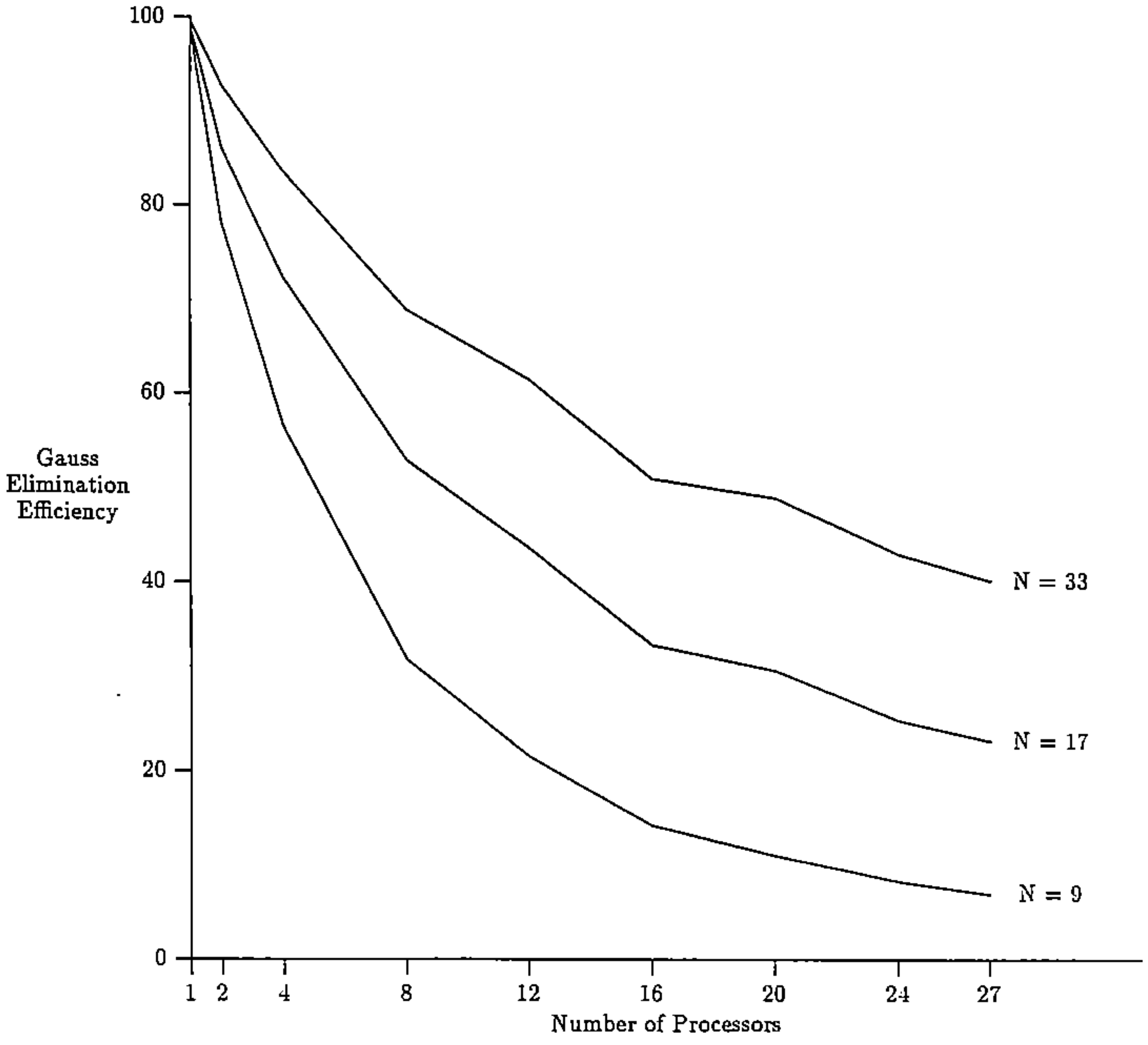

Figure 3: Efficiencies for Gauss Elimination

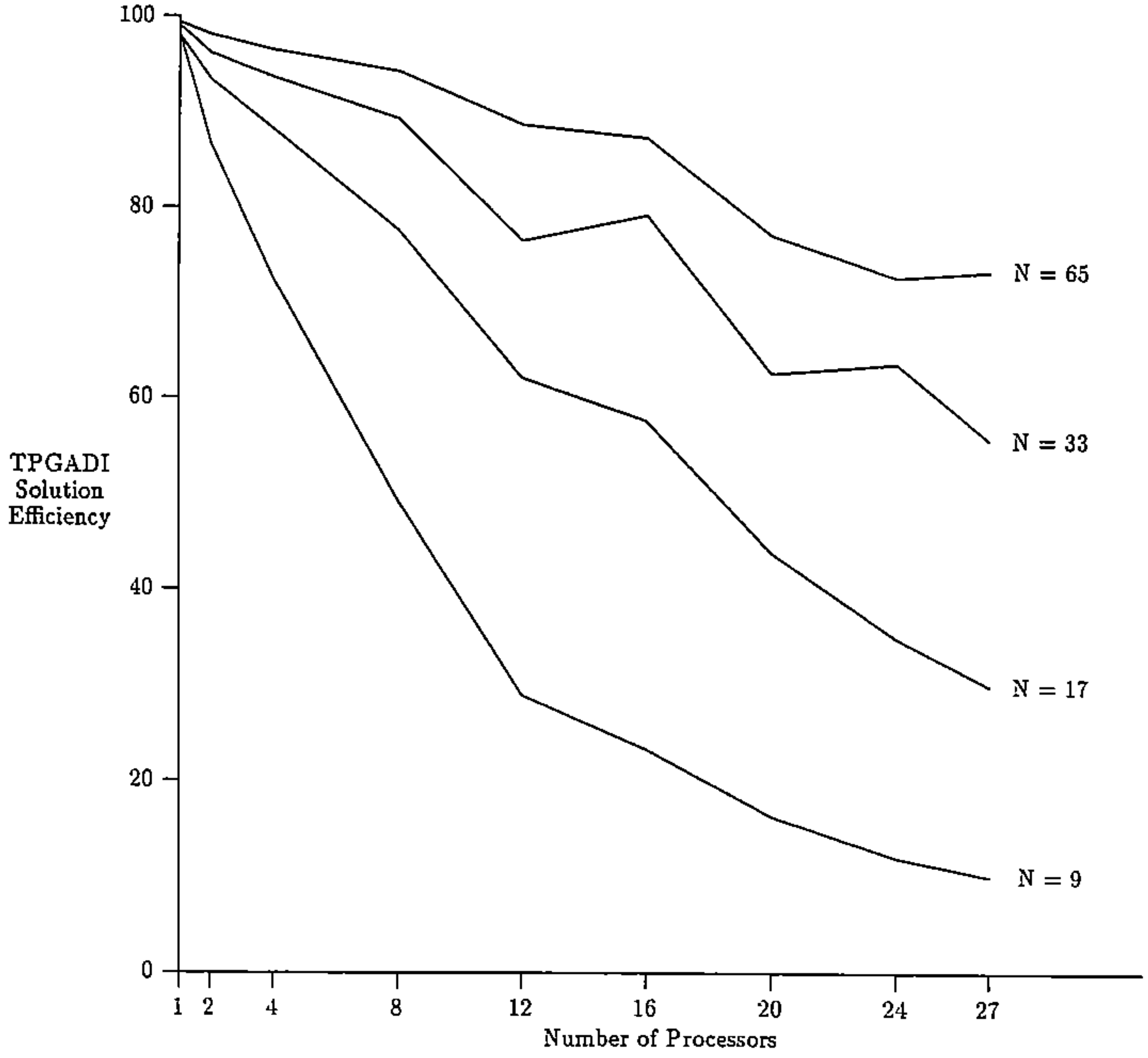

Figure 4: Efficiencies for TPGADI Solution. Note: Slight bump for 16 processors due to all processors being used equally since 16 evenly divides sizes of linear systems.

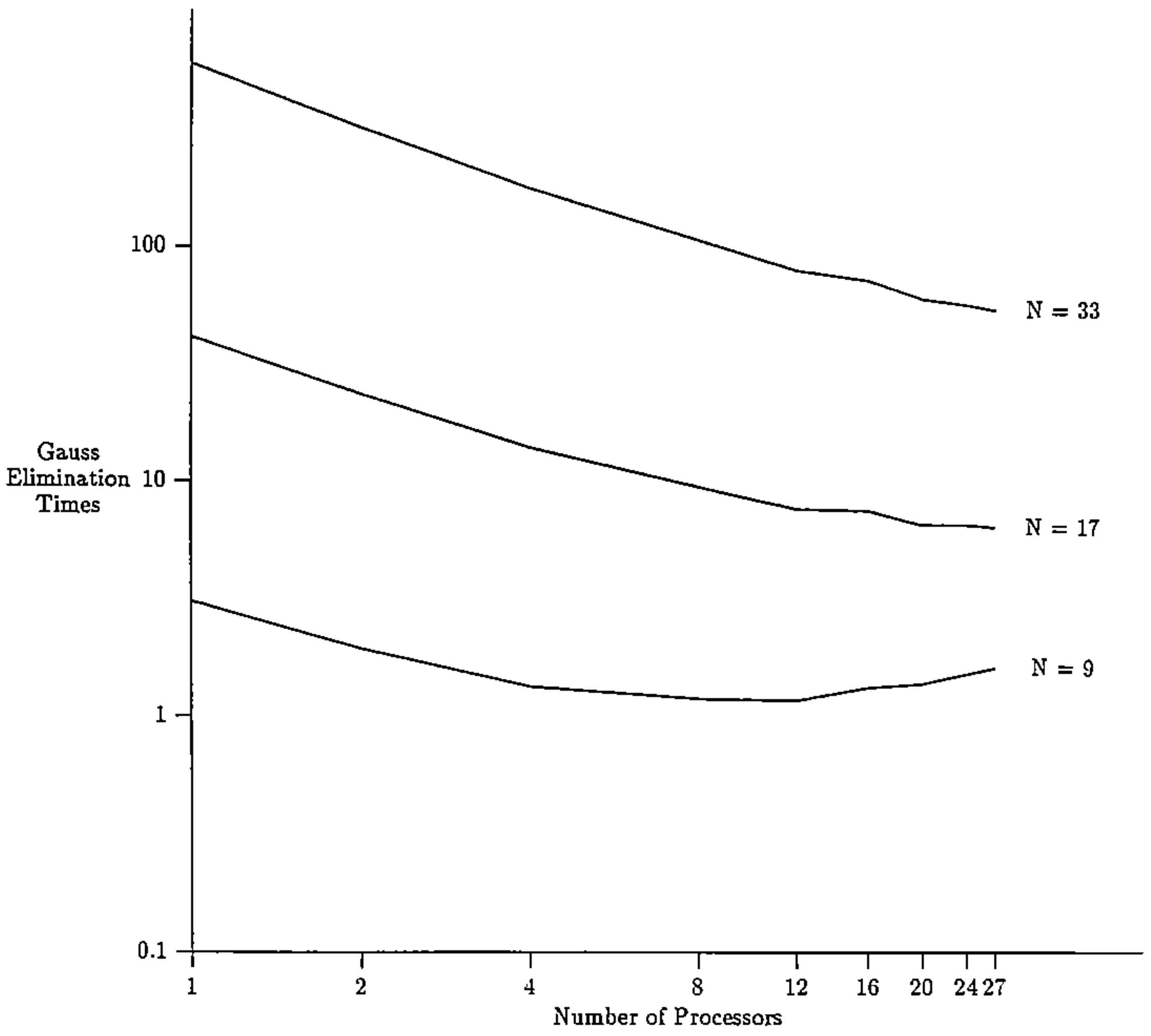

Figure 5: Times for Gauss Elimination (seconds)

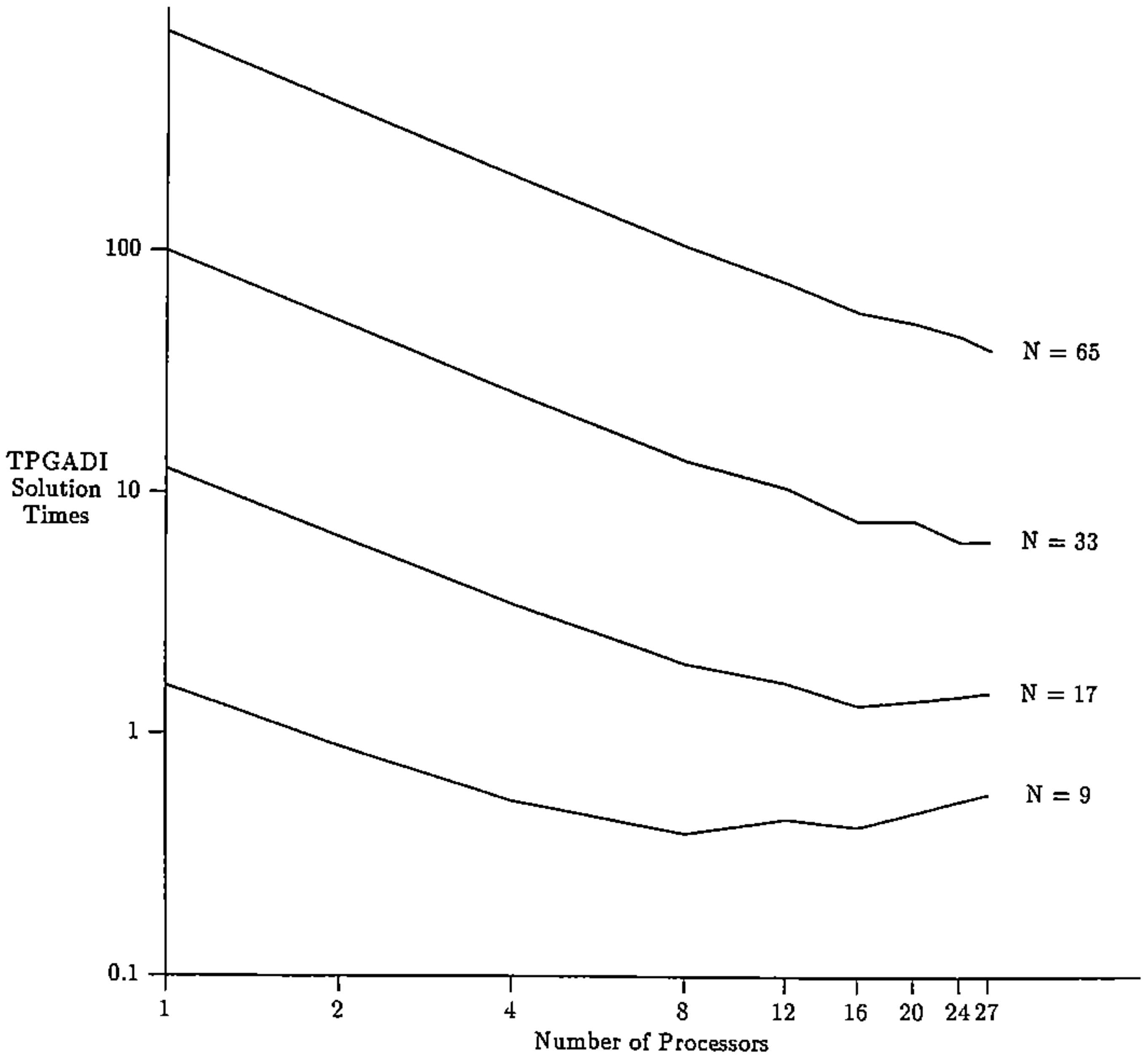

Figure 6: Times for TPGADI Solution