1987

Piplined Iterative Methods for Shared Memory Machines

John P. Bonomo

Wayne R. Syksen

Report Number:
87-688

Pipelined Iterative Methods for Shared Memory Machines

John P. Bonomo
Wayne R. Dyksen [†]

CSD-TR 688
August 24, 1987

Abstract

In this paper we describe a new parallel iterative technique to solve a set of linear equations. The technique can be applied to any serial iterative scheme and involves pipelining successive iterations. We give an example of this technique by modifying the classical successive over-relaxation method (SOR). The algorithm is implemented on a Sequent Balance 21000 and the experimental results are presented.

[†]Supported in part by National Science Foundation grant DCR-8602385

1 Introduction

Much research has been performed in the parallelization of sequential iterative methods for solving the sets of linear equations generated from the discretization of elliptic partial differential equations (PDEs). This research has been geared toward both vector and multiprocessor machines. Indeed, the Jacobi algorithm has been recognized as an ideal algorithm for parallelization, since the update of any matrix element in each pass can be done independently of all the others elements. However, parallel Jacobi schemes suffer from the same drawbacks as does serial Jacobi, namely very slow convergence rates.

Several different parallel modifications have been performed on the classical Gauss-Seidel and successive overrelaxation (SOR) methods. A general survey of these methods is given in Ortega and Voigt [5]. One set of methods involve the multicoloring of grid elements and the dovetail update of those elements of like color. The simplest of these methods is the Red-Black method, in which two colors are assigned to the grid elements in a checkerboard manner. Then, all grid elements of one color can be updated in a Jacobi-like sweep in odd numbered passes, while those of the second color are updated in even numbered passes. Work has been performed on this scheme by Ericksen [2] and Evans [3]. A second parallel variation is to assign each processor a set of grid elements to update, and then to allow all processors to run asynchronously (see [1] and [4]). Thus no attempt is made to synchronize each iterative sweep. This method avoids two problems which are inherent in any algorithm that attempts to synchronize sweeps: one is the extra computational work that must be performed by each processor at the end of a sweep to verify when it can start the next sweep, and the second is the time that a processor may waste while waiting for all other processers to finish a sweep. However, the asynchrony makes analysis of the algorithm and proofs of its convergence rate difficult. A third parallel variation has been developed by Patel and Jordan [6] where each processor is assigned the task of updating one row of grid points. Since SOR re-uses updated values as soon as they are available, each processor must wait for the previous processor's iterative updates before it can begin updating the elements in its row. Synchronization between the processors is controlled by full/empty flags assigned to each memory location. These flags are built into the Heterogeneous Element Processor (HEP) on which their algorithm was developed.

The techniques presented in this paper are closest in spirit to that of Patel and Jordan, but are more flexible and are not as machine dependent. In Section 2 we describe our general pipelined iterative schemes for multiprocessors. Section 3 gives pertinent details of the Balance 21000, a shared memory machine manufactured by Sequent Computer Systems, Inc. on which we have implemented several pipeline algorithms and in Section 4 we give details of one such scheme, the Pipelined SOR (PiSOR) algorithm along with the experimental results. We close with a discussion of the results in Section 5.

2 Pipelined Iterative Techniques

Consider the system of linear equations

$$A\mathbf{u} = \mathbf{f}, \tag{1}$$

where $\mathbf{u}$ is a vector of unknowns of length N^2 (we use N^2 in anticipation of the formulation which results when discretizing a PDE) and A is a square matrix with order N^2 and bandwidth K. Consider a so-called *regular splitting* of A given by

$$A = M_1 - M_2,$$

where M_1 and M_2 are also square matrices of order N^2 and M_1 is nonsingular. This leads to the general iterative technique

$$M_1\mathbf{u}^{(k+1)} = M_2\mathbf{u}^{(k)} + \mathbf{f}. \tag{2}$$

If A^{-1}, M_1^{-1} and M_2 are all positive, then the above iterative method is convergent for any initial $\mathbf{u}^{(0)}$ (Varga [7]). These conditions are satisfied by many of the discretization techniques used in solving elliptic PDEs. For our purposes, we will also assume that M_1 and M_2 are lower and upper triangular, respectively. Clearly, the bandwidths of both M_1 and M_2 are no more than K.

Let p be the number of processing elements (PEs) available, $PE_1, PE_2, \ldots, PE_p$. Each processor performs one iteration in its entirety, where each iteration proceeds by first updating vector element u_1, then element $u_2, u_3, \ldots, u_{N^2}$. PE_1 performs iterations $1, p+1, 2p+1, \ldots$, PE_2 performs iterations 2, $p+2, 2p+2, \ldots$ etc. Iterations follow one another in a pipelined fashion; i.e., after iteration 1 has proceeded for a certain amount of time, iteration 2 starts and both iterations update different vector elements simultaneously. After another period of time, iteration 3 begins and three updates proceed in parallel through the vector $\mathbf{u}$.

In order for traditional error analysis and convergence rate proofs for the given iterative technique to hold we must ensure that no iteration overlaps the one ahead of it. To accomplish this, we introduce a set of synchronization flags assigned to various vector elements. We define δ to be the *pipeline spacing* which specifies the spacing of the synchronization flags. Informally, the algorithm works as follows: Each processor updates δ vector elements at a time, starting with u_1. For the moment, assume $\delta = 1$. Before updating u_i, a processor must be sure that the previous iteration has updated all the vector elements that are used in the update, specifically u_{i-K} through u_{i+K}. In order to ensure this, the processor checks the synchronization flag associated with u_{i+K}. If it indicates that u_{i+K} has been updated by the previous iteration, then the processor can continue the current iteration by updating u_i (the fact that u_{i+K} has been updated implies that all vector elements u_{i-K} through u_{i+K-1} have also been updated by the previous iteration). Before proceeding to the next vector element, the processor updates the synchronization flag associated with u_i to signal the next iteration. Note that in the case $\delta = 1$ each iteration leads its successor by

$K+1$ vector elements. For general δ, each iteration will lead the next iteration by $(K+\delta)$ vector elements.

We now give a more formal specification of the algorithm. For a given value of δ, synchronization flags are associated with vector elements $u_{\delta+K}, u_{2\delta+K}, \ldots, u_{m\delta+K}$, where m is the lowest value such that $m\delta + K \geq N^2$, i.e.

$$m = \left\lceil \frac{N^2 - K}{\delta} \right\rceil .$$

Note that the last synchronization flag may be associated with a fictitious vector element. We label the synchronization flags $s_1, s_2, \ldots, s_m$. Each synchronization flag can have the values $1, 2, 3, \ldots$, and all are initialized to 1. The meaning of the value of each synchronization flag is the following:

> if synchronization flag s_i has value l, it implies that iterations $1, 2, \ldots, l-1$ have completed updating vector elements $u_1, \ldots, u_{i\delta+K}$, and that iteration l can proceed to update vector elements $u_1, \ldots, u_{i\delta}$; all higher iterations must wait.

Throughout an iteration, each processor performs two operations with each synchronization flag, check and update. A synchronization flag is checked prior to updating vector elements u_1, $u_{\delta+1}$, $u_{2\delta+1}, \ldots$, $u_{(m-1)\delta+1}$. Suppose that a processor is performing the k^{th} iteration and is currently at check point $u_{i\delta+1}$. In order to proceed to update the next δ elements, the processor must check synchronization flag s_{i+1}. If its value is equal to k, then the processor can proceed to update $u_{i\delta+1}$ through $u_{(i+1)\delta}$; otherwise, it must wait for the previous iteration to update s_{i+1}. A synchronization flag is updated by each processor after it updates the vector element associated with that synchronization flag. Thus, once the $(k-1)^{st}$ iteration has updated $u_{(i+1)\delta+K}$, it updates s_{i+1} by setting its value to k, which allows the k^{th} iteration to proceed. While it performs an iteration, each processor simultaneously performs stopping criteria calculations. As soon as one processor decides it can stop, it sets a global flag indicating that all processors should stop after completing their current iteration. This flag is checked by each processor after it updates u_{N^2} and s_m.

The application of pipeline techniques to a serial iterative method introduces two new features in the resulting parallel method which need to be considered when implementing the algorithm and interpreting the results. Since iterations are separated from each other by $(K+\delta)$ grid elements, at most $\lfloor N^2/(K+\delta) \rfloor$ iterations can be pipelined at once. If the number of processors $p = \lfloor N^2/(K+\delta) \rfloor \equiv p_{\max}$, then the use of additional processors is superfluous. In point of fact, once $p_{\max}$ processors are in use, additional processors can be expected to have a detrimental effect on the execution time since their presence results in added contention for shared memory. The second feature to be considered is the effect that pipelining has on the number of iterations needed to obtain a solution. Let ν represent the number of iterations that a serial iterative method needs to

reach a solution (this value is dependent on the particular elliptic PDE, the matrix splitting and $\mathbf{u}^{(0)}$). The total number of iterations needed for the pipeline version of the method is slightly larger than for the sequential method. This is due to the fact that once a processor determines that the stopping criteria have been met (which will occur at the end of iteration ν) it signals to the other $p-1$ processors and they in turn finish their iterations. Thus the total number of iterations is actually $\nu + p - 1$. For $p << N$ however, this will result in only a small relative increase in the total number of iterations.

The pipelined nature of the above process can be viewed in two different manners. The first way is to view the vector $\mathbf{u}$ as the "pipe" in which the processors are sent streaming through, with each of the vector elements u_i as stages of the pipe. Each processor is separated by the ones ahead and behind it by $(K + \delta)$ stages, and this separation is maintained by the synchronization flags. The second model (and perhaps the more orthodox of the two) is to view the data streaming through a pipe whose stages each represent one iteration. Each stage is performed by a processor, and each processor may perform many different stages. This differs from the conventional view of pipelines used in vector supercomputers is several ways. The conventional pipeline is used to perform a simple operation (e.g. the addition of two vectors), and does so by breaking down this operation into its basic assembly language instructions which then become the stages of the pipeline. In our discussion above, the operation which the pipeline performs is a much more complicated one, namely the solution of a set of linear equations; its stages are likewise more complicated, each performing one iteration of the vector. This difference is mainly one of scale. A second difference is that each stage of a conventional pipeline works on one item of data at a time, and then passes it on to the next stage. In our pipelined iterative methods, each stage works on a group of vector elements before "passing them on" to the next stage. This granularity of the data stream is determined by both δ and K. Specifically, each stage first works on $(K + \delta)$ vector elements, and thereafter on groups of δ elements. A final difference is that conventional pipelines have a known number of stages, whereas the number of stages in any pipelined iterative method is unknown.

3 The Balance 21000

The Balance 21000 is a shared memory multiprocessor machine manufactured by Sequent Computer Systems, Inc. The machine running at our installation is equipped with 12 processors and 16 Mbyte of memory. The Balance System Bus is a 64-bit bus used to connect the processors to the memory. Currently the processors use only 32 bits and can achieve a sustained data transfer rate of 26.7 Mbytes per second. The operating system developed by Sequent is called DYNIX and is completely compatible with both UNIX 4.2bsd and UNIX System V. In addition, DYNIX has certain enhancements which allow it to take advantage of the multiprocessor environment. For ex-

ample, the DYNIX kernel is shared over the entire system, and all operating system responsibilities (e.g. executing processes, handling interrupts, etc) are divided among all PEs. Because of this, it is advisable to use at most all but one of the PEs when running a parallel job, saving at least one to perform the necessary DYNIX functions.

The Balance system supports enhanced, parallel versions of several languages, among them FORTRAN 77, C and Pascal. Our software was written exclusively in FORTRAN 77. The Balance supplies two methods with which to include parallel features in this language. The first is the automatic generation of parallel code for FORTRAN DO loops. This method requires the user to classify all variables in the DO loop with regards to their read and write usage, and then to include various control statements which instruct the compiler to generate parallel code. The second method requires the user to include parallel routines explicitly in the code. This second method was used in our application of the pipeline iterative technique to the SOR method. The pertinent routines used include the following: m_set_procs — used to set the number of child processes; m_fork — executes a subprogram in parallel over p processors, where p is determined by the most recent call to m_set_procs; m_get_myid — allows each process to determine its unique identification number in the range 1 to p; m_get_numprocs — returns the number of current processes set by the most recent call to m_set_procs. The Balance system does not allow the user to specify which processor each process should run on. Instead, the user specifies only the number of processes needed, and the operating system automatically distributes them across the processors.

4 The PiSOR Algorithm and Experimental Results

We now give a specific example of our pipelined techniques, applying them to the serial SOR method. We start with a general self-adjoint elliptic PDE given by

$$\begin{aligned} -(pu_x)_x - (qu_y)_y + ru &= f(x,y) \qquad (x,y) \in \Omega, \\ u &= g(x,y) \qquad (x,y) \in \partial\Omega \end{aligned} \tag{3}$$

where p, q and r are all functions of x and y, and p and q are strictly positive. For simplicity, we assume Dirichlet boundary conditions. We discretize (3) by placing an $N+2$ by $N+2$ mesh over the domain and using symmetric finite difference approximations to obtain a set of linear equations (1). The u values associated with the N^2 interior grid points make up the vector u, and we will now refer to each vector element as a *grid element.* Since each grid element is related to its four nearest neighbors, matrix A has bandwidth $K = N$. The elements of u are numbered row-wise u_i, $i = 1, \ldots, N^2$, so that grid point (i,j) is associated with $u_{(j-1)N+i}$. Figure 1 shows an example of this numbering when $N = 6$. Figure 2 shows the interior grid points for this discretization and the placement of the synchronization flags for two cases: $\delta = 4$ and $\delta = 5$.

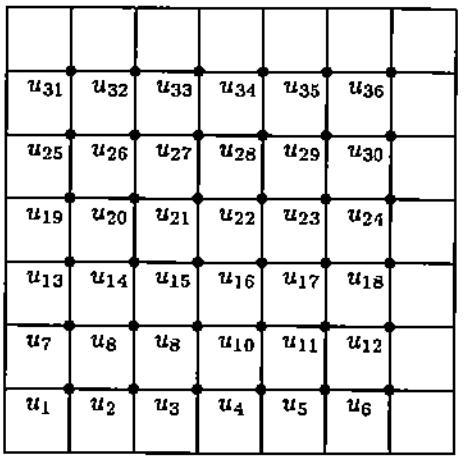


Figure 1: Grid element numbering for $N = 6$.

If we let $A = D - E - F$, where D is made up of the diagonal elements of A, and E and F are respectively strictly lower and upper triangular, then the SOR method is given by (2) with

$$M_1 = \frac{1}{\omega}(D - \omega E); \;\; M_2 = \frac{1}{\omega}(\omega F + (1-\omega)D), \tag{4}$$

where ω is the *relaxation factor* and $0 < \omega < 2$. The use of symmetric finite differences along with the restrictions on p and q insure that the matrices A, M and N satisfy the properties stated in Section 2. Given this formulation of the SOR method, we can now apply the pipeline techniques to obtain the PiSOR method. It is often the case that ω is not constant across iterations, but that is easily handled in the PiSOR algorithm, since each processor can calculate it's own value for ω and use it independently of the other processors.

An example execution of the PiSOR algorithm is shown in Figure 3 for the case when $N = 6, \delta = 12$. In Figure 3a, all processors but PE_1 are waiting on s_1, while PE_1 begins updating grid elements. Before it updates u_{13}, it checks s_2's value. Since that value is 1, PE_1 can proceed to update the next 12 ($= \delta$) grid elements. After it updates u_{18} it changes s_1's values to 2, and proceeds to update u_{19} (Figure 3b). Since $s_1 = 2$, PE_2 can start the second iteration. Figure 3c shows one more step in this process. After PE_1 finishes iteration 1 and updates s_3, it waits on synchronization flag s_1 until its value equals $p+1$, in which case is starts performing iteration $p+1$, and so on.

The PiSOR algorithm was applied for a variety of values of N, δ and p on the following elliptic PDE:

$$\begin{aligned} -u_{xx} - u_{yy} &= -4 && (x,y) \in (0,1) \times (0,1) \\ u &= x^2 + y^2 + y && x = 0,1, \;\; y = 0,1. \end{aligned} \tag{5}$$

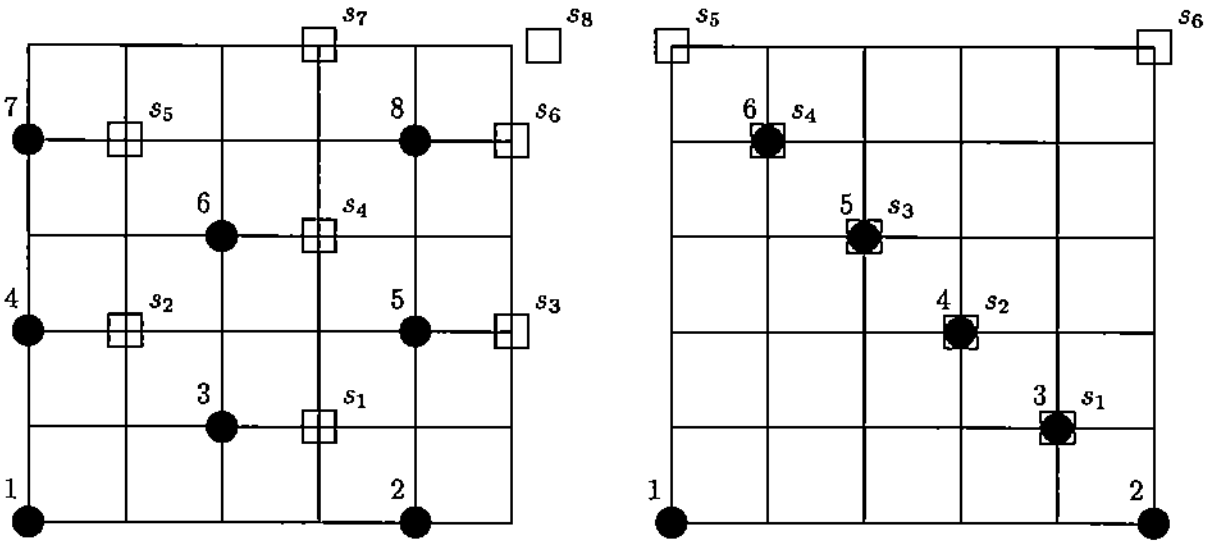


Figure 2: PiSOR synchronization flag placement with (a) $N = 6, \delta = 4$ and (b) $N = 6, \delta = 5$. Circles indicate flag check points and squares indicate flag locations and update points. Only interior gird points are shown.

Table 1 shows the experimental results for N^2 = 100, 400, 900, 1600, 2500 and 3600, δ = 1, 2, 5, 10, 20, 50 and 100, and p = 1, 2, 4, 8 and 11. Table 2 shows the execution times and iteration counts for sequential SOR. All times represent just the solution time; they do not include startup and discretization time.

The results for δ = 1, 10 and 100 are graphed in Figures 4, 5 and 6 respectively, where the lines show execution times for six values of N, and the circles under each line indicate the sequential running time. As would be expected, as δ grows the execution time of PiSOR with 1 PE approaches the sequential time. Note also that as the number of processors become greater than $p_{\max}$ (= $\lfloor N^2/(K+\delta) \rfloor$), the execution time increases, also as expected. This effect is most notable for δ = 100 (Figure 6). Table 3 lists the values of $p_{\max}$ for various values of N and δ.

The efficiency of the PiSOR algorithm is show in Figures 7 and 8 for N = 40 and 60 respectively. We use the standard definition of efficiency, $E(p)$, namely:

$$E(p) = \frac{\text{Sequential SOR time}}{p \times (\text{PiSOR time using } p \text{ processors})}$$

Typically for any given algorithm, as the number of processors increases, $E(p)$ decreases, and this is true for the PiSOR algorithm. However, for N = 60 and a suitable choice for δ, efficiency levels over 0.8 can be reached even when using 11 processors. When $p \leq 4$, efficiency levels over 0.9 are obtained over a wide range of δ values. Note once again that once $p_{\max}$ is reached, the efficiency of the algorithm diminishes rapidly.

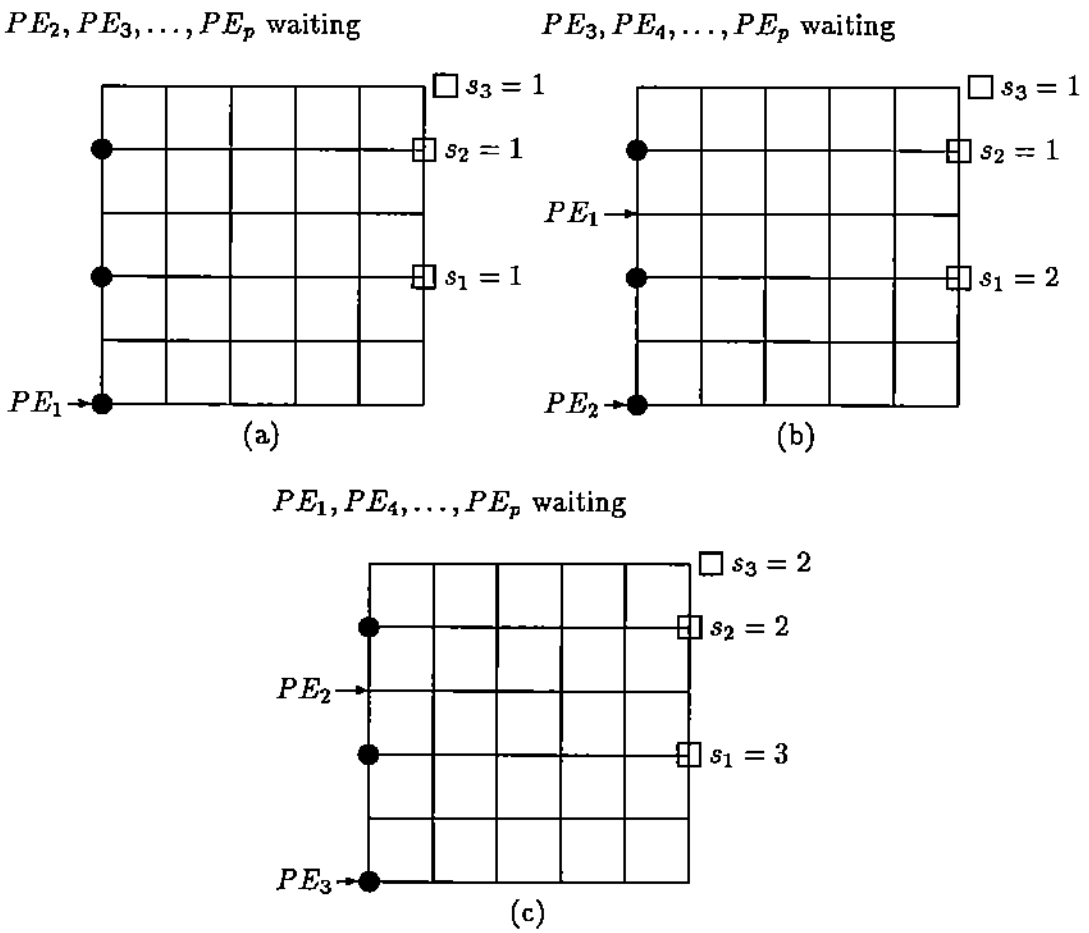


Figure 3: Sample iterations with $N = 6, \delta = 12$. Only interior grid points are shown.

N^2	δ	Number of Processors					N^2	δ	Number of Processors				
		1	2	4	8	11			1	2	4	8	11
100	1	.60	.32	.18	.11	.12	1600	1	34.65	17.60	9.01	4.81	3.67
	2	.57	.31	.17	.12	.12		2	32.93	16.74	8.56	4.57	3.51
	5	.55	.29	.17	.13	.14		5	31.89	16.22	8.29	4.43	3.38
	10	.55	.29	.17	.16	.17		10	31.52	16.02	8.20	4.38	3.35
	20	.54	.29	.20	.22	.24		20	31.34	15.93	8.15	4.35	3.34
	50	.54	.34	.37	.41	.44		50	31.24	15.87	8.11	4.33	3.33
	100	.54	.56	.59	.66	.71		100	31.19	15.85	8.14	4.36	3.34
400	1	4.76	2.46	1.29	.73	.58	2500	1	66.34	33.61	17.12	8.88	6.70
	2	4.53	2.34	1.23	.69	.55		2	63.02	31.92	16.26	8.44	6.38
	5	4.39	2.26	1.19	.67	.54		5	61.01	30.91	15.75	8.17	6.18
	10	4.34	2.25	1.17	.67	.53		10	60.32	30.56	15.58	8.09	6.11
	20	4.33	2.24	1.18	.67	.60		20	59.99	30.39	15.46	8.05	6.08
	50	4.31	2.23	1.18	.91	.95		50	59.80	30.28	15.39	8.01	6.08
	100	4.30	2.23	1.41	1.50	1.56		100	59.70	30.26	15.41	8.06	6.12
900	1	14.96	7.58	3.97	2.11	1.64	3600	1	109.98	55.50	28.32	14.63	10.92
	2	14.21	7.20	3.77	2.01	1.56		2	104.36	52.60	26.99	13.92	10.37
	5	13.77	6.97	3.65	1.95	1.52		5	101.05	50.87	26.04	13.47	10.03
	10	13.62	6.89	3.61	1.93	1.51		10	99.90	50.31	25.76	13.32	9.93
	20	13.53	6.87	3.59	1.93	1.51		20	99.38	50.05	25.60	13.22	9.88
	50	13.48	6.85	3.59	1.94	1.52		50	99.01	49.87	25.51	13.20	9.86
	100	13.46	6.84	3.58	2.26	2.33		100	98.90	49.78	25.49	13.19	9.90

Table 1: PiSOR Execution times (in secs).

N^2	time	iters
100	.51	34
400	4.02	68
900	12.60	95
1600	29.22	124
2500	55.92	152
3600	92.69	175

Table 2: Sequential SOR execution times (in secs) and number of iterations

	δ						
N	1	2	5	10	20	50	100
10	9	8	6	5	3	1	–
20	19	18	16	13	10	5	3
30	29	28	25	22	18	11	6
40	39	38	35	32	26	17	11
50	49	48	45	41	35	25	16
60	59	58	55	51	45	32	22

Table 3: $p_{\max}$ values

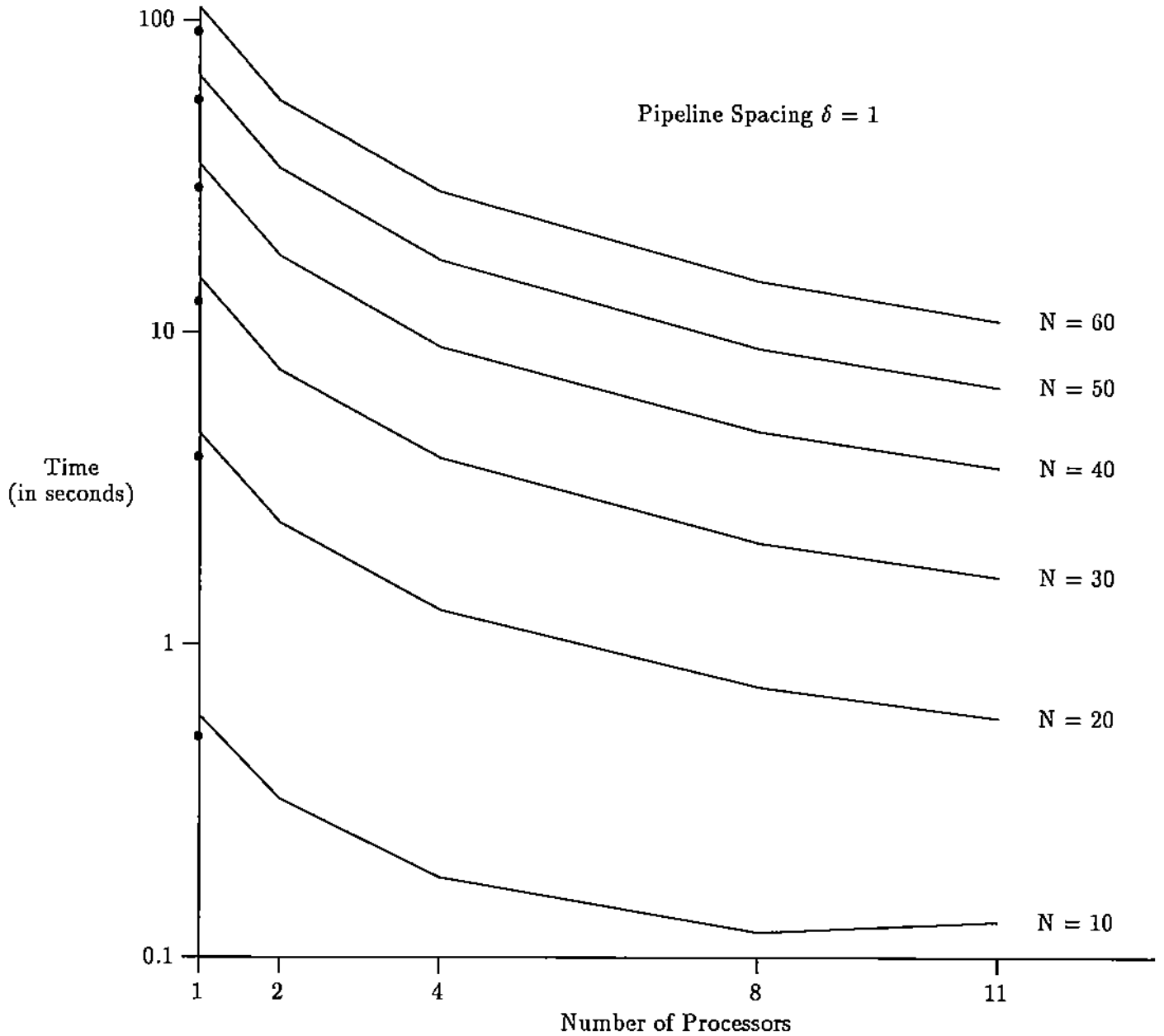


Figure 4: Execution times for $\delta = 1$. Black circles indicate sequential times.

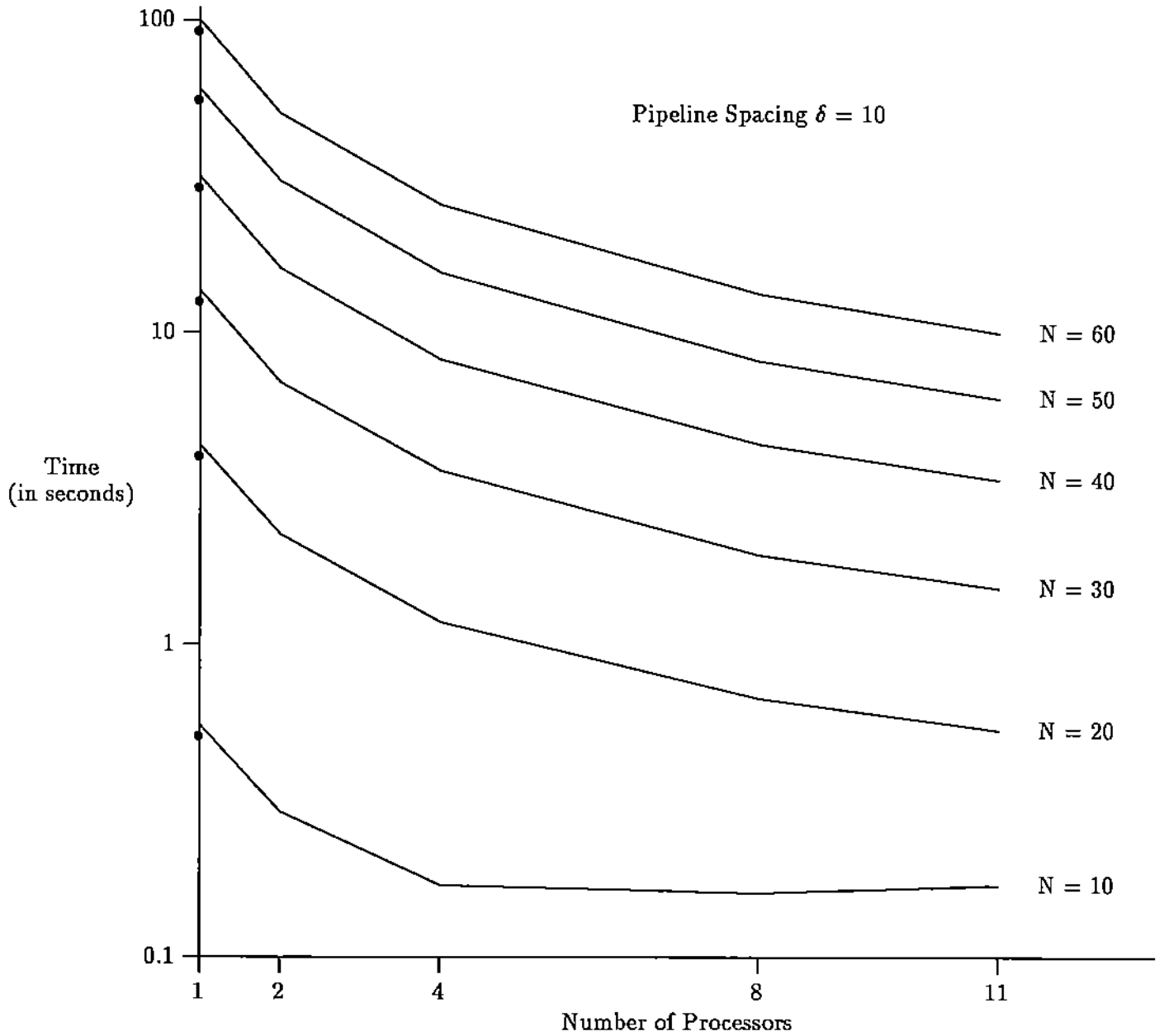


Figure 5: Execution times for $\delta = 10$. Black circles indicate sequential times.

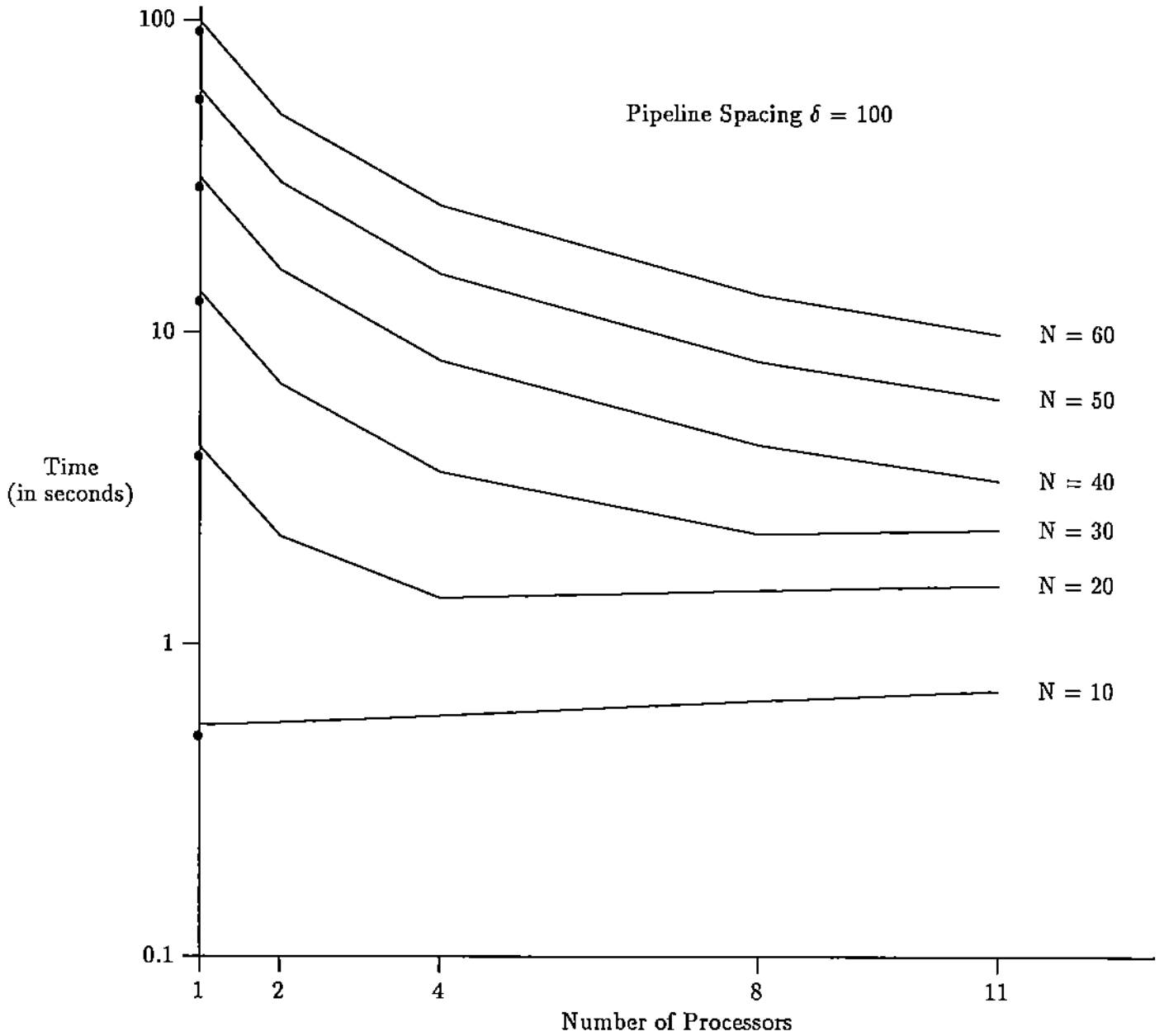
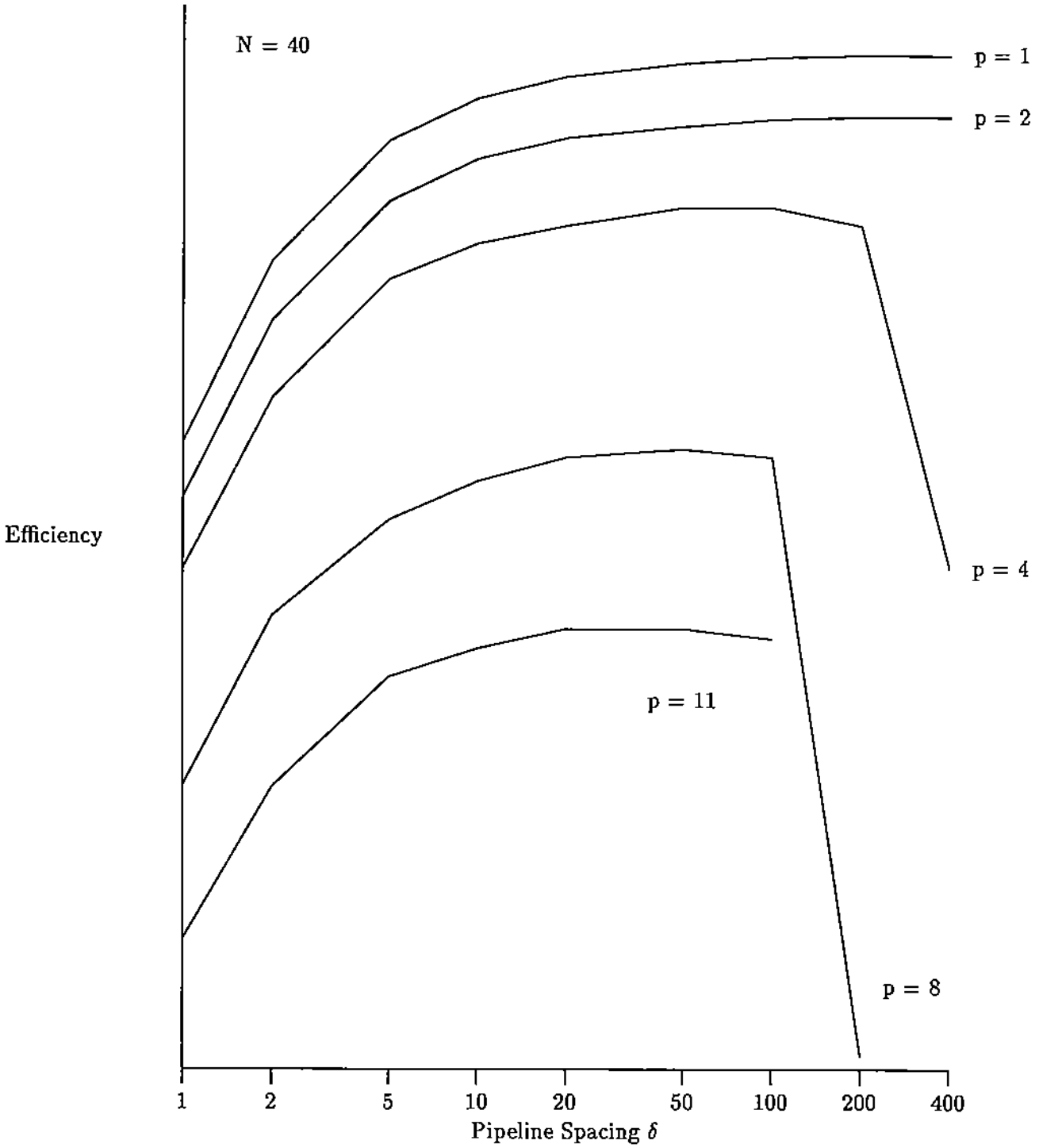
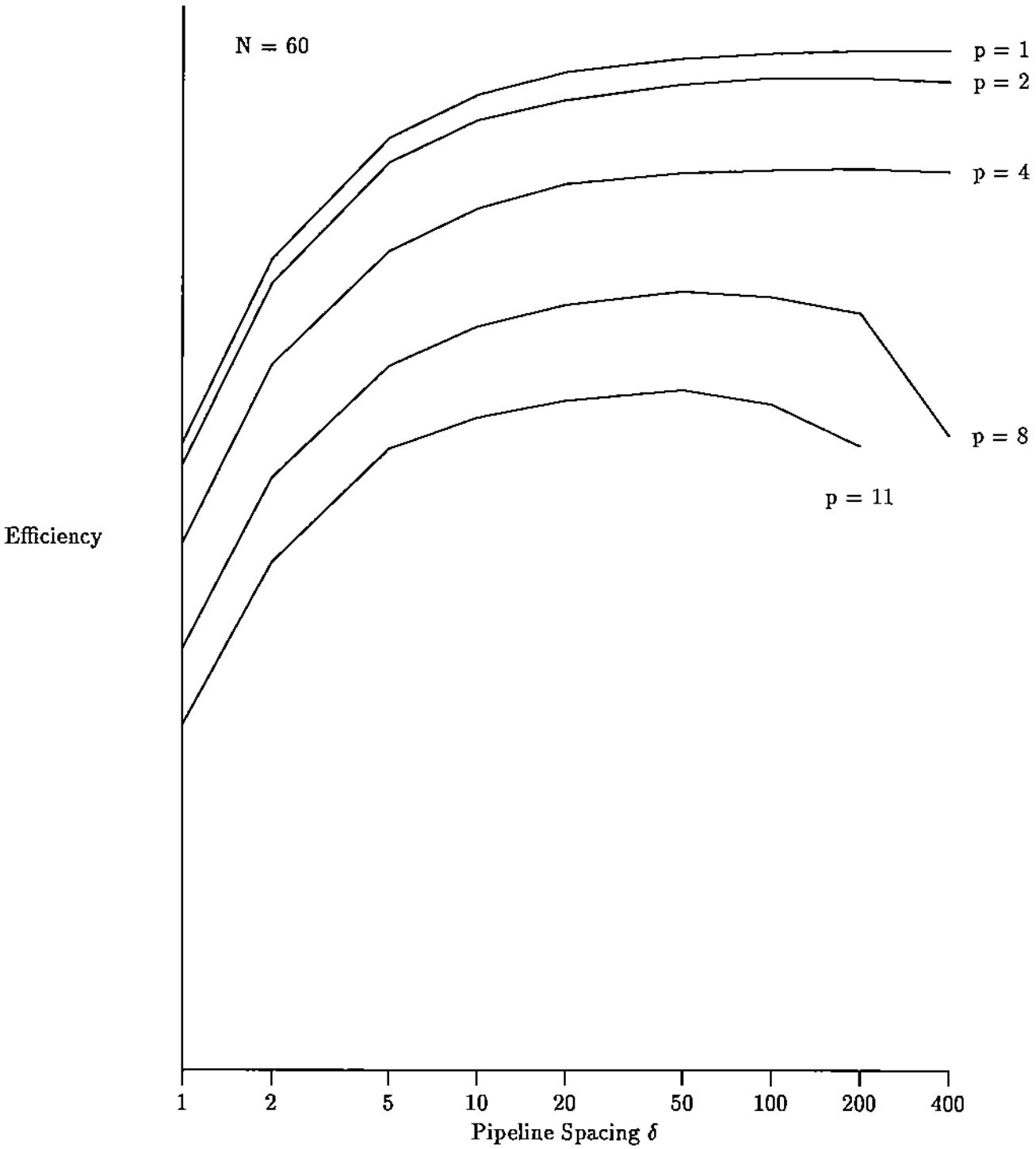


Figure 6: Execution times for $\delta = 100$. Black circles indicate sequential times.


N = 40
p = 1
p = 2
p = 4
p = 11
p = 8
Efficiency
1
2
5
10
20
50
100
200
400
Pipeline Spacing δ


N = 60
p = 1
p = 2
p = 4
p = 8
p = 11
Efficiency
1
2
5
10
20
50
100
200
400
Pipeline Spacing δ

5 Discussion

The pipeline iterative techniques presented in this paper present an effective means to parallelize basic serial iterative methods for the solution of linear systems of equations. As evidenced in the application of these techniques to the SOR method, the resulting algorithms sustain a high level of efficiency in processor usage while also maintaining the original serial algorithms convergence properties. The pipeline techniques are readily applicable to those methods which alternate iterative methods each sweep, e.g. SOR methods which vary the value of ω after each iteration.

Current work in progress includes the application of our pipeline techniques to Schwarz splitting, and a more generalized version of the method to cover semi-iterative techniques, where the k^{th} iterative sweep depends on two or more previous sweeps. Work is also being done on the determination of the optimal or near optimal pipeline spacing value. Note that for any given serial iterative method, the application of the pipeline techniques produces a family of methods, where each method differs from the others by its value of δ. The optimal value for δ is not intuitively apparent; low values of δ allow iterations to be pipelined very close to one another, but consequently force each iteration to perform a proportionally larger amount of work checking and updating the synchronization flags. Conversely, large values of δ allow each iteration to process a larger number of contiguous vector elements without having to potentially wait at a synchronization flag, but now the distance between iterations has increased. Preliminary results suggest that the efficiency of the algorithm varies slightly with δ so that only a rough estimate of the optimal pipeline spacing is needed.

References

[1] G. Baudet, "Asynchronous iterative methods for multiprocessors", J. ACM, 25 (1978), pp. 226–244.

[2] J. Ericksen, "Iterative and direct methods for solving Poisson's equation and their adaptability to ILLIAC IV", Center for Advances Computation Document 60 (1972), Univ. Illinois, Urbana–Champaign.

[3] D. J. Evans, "Parallel S.O.R. iterative methods", Parallel Computing 1 (1984), pp. 3–18.

[4] H. Kung, "Synchronized and asynchronous parallel algorithms for multi-processors", Algorithms and Complexity (1976), pp. 153–200.

[5] J.M. Ortega and R. G. Voigt, "Solution of Partial Differential Equations on Vector and Parallel Computers", SIAM Review, 27 (1985), pp. 149–240.

[6] N. R. Patel and H. F. Jordan, "A parallelized point rowwise successive over-relaxation method on a multiprocessor", Parallel Computing 1 (1984), pp. 207–222.

[7] R. S. Varga, *Matrix Iterative Analysis*, Prentice Hall, 1962.