Some Experiments with Reprogramming LINPACK Routines for Parallel Machines

L. D. Fosdick
C. J. C. Schauble
F. M. Dedolph
B. Schlaman

CU-CS-408-88 August, 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

We gratefully acknowledge the computing support of Encore Computer Corporation.

ABSTRACT

During the summer of 1986, experiments in parallel programming were made on the Encore Multimax for the purpose of gaining experience in and learning efficient methods for programming parallel machines. Four components of the Linpack package, the Gaussian elimination routine and its companion back-solution routine, DGEFA and DGESL, with the QR Decomposition routine and its back-solution routine, DQRDC and DQRSL, were rewritten to allow parallelism in this experiment. Timings are compared against the original sequential versions.

1. INTRODUCTION

The purpose of this report is twofold. First of all, it documents the results of experiments done in Summer 1986 in writing, running, and timing parallel versions of four Linpack routines. Secondly, it advocates the use of the Force macro extension of Fortran 77 as the language for writing parallel versions of existing Fortran code.

1.1. Statement of Project

The main goal of this project was to code and test optimally parallel programs on the Encore Multimax computer. We wanted to observe the improvements in execution times between sequential and parallel versions of a program. Fortran 77 was chosen as the language for the project because of the wide variety of commercially available numerical packages using Fortran.

The Linpack routines [DBM79], a collection of Fortran subroutines developed for use in solving linear algebraic equations and linear least squares problems, were chosen as a source of routines because they are in Fortran, they are well-documented, and they have been used previously for benchmarking parallel machines [Don81]. Rewriting well-tested sequential programs into parallel programs simplifies testing; the output for any input has to match for both versions. Compatibility with existing library routines makes the substitution of the new routine simply a matter of relinking existing programs. For this reason, a constraint of the project was to maintain complete compatibility with the Linpack routines.

A parallel Gaussian elimination routine from the Linpack package of Fortran subroutines rewritten in the Force had been used as an example in some papers describing the Force [Jor84]. To follow up this research, the Linpack Gaussian elimination routine, DGEFA, and its accompanying back-solution routine, DGESL, were chosen to be rewritten in parallel. It was also decided to include two similar routines for the QR Decomposition, entitled DQRDC and DQRSL.

1.2. Outline of this Report

The remainder of this paper is divided into three sections. Section 2 provides the details of the project: a description of the machine, the language, and the parallelized algorithms used for the experiment. Section 3 describes the experiments and the results for each of the routines. Section 4 summarizes the results, suggests what has been learned by the project, and considers possible future experiments.

2. EXECUTION OF PROJECT

2.1. Environment

2.1.1. Machine Used

The Encore Multimax is a shared-memory MIMD (Multi-Instruction, Multi-Data) machine. This experiment used a model with eight, tightly-coupled, 32-bit processors connected to the common memory and the I/O interfaces by a wide high speed bus (the "Nanobus"). The operating system, provided by Encore, is UMAX 4.2, which allows multi-threading.

The machine was on loan from the Encore Computer Corporation and open to use by the entire University. Experimentation on this machine was highly encouraged. For more information on this machine, see the Multimax Technical Summary [Enc87].

2.1.2. Language Used

Parallelism in the programs is implemented with the Force parallel programming constructs developed by Jordan [Jor87]. These constructs are built as an extension to Fortran 77 for shared-memory multiprocessors. A version of the Force macro-processor has been installed on the Encore.

The natural program state for a Force program is many parallel processes working together on one set of code. Sequential code to be executed by a single process can be coerced by the *barrier* construct, when needed. These processes may or may not have an actual processor associated with them. The number of processes is chosen at the beginning of execution, at which time the processes are automatically initiated. Eventually the processes terminate at a fixed point, usually at the end of the program. Thus, the programmer is spared the burden of spawning, destroying, and keeping track of a set of processes.

Original Force Code:

```
Critical Region  TT
  TT = TT + T
End critical
```

Preprocessed Code:

```
CALL spin_lock(LTT)
  TT = TT + T
CALL spin_unlock(LTT)
```

Figure 1: Example of a Critical Section

A Force program is converted into a Fortran 77 program with extensions for handling the parallel constructs. A macro preprocessor translates the parallel instructions into manufacturer-supplied system commands which support parallelism on the Multimax; the result is a Fortran program with the machine-specific parallel library calls. A description of the parallel constructs with their syntax and semantics is detailed in Jordan et al [JBA87]. Since the Linpack routines are in Fortran, using a parallel extension of Fortran for the reprogramming is a reasonable approach. Inserting the Force constructs to express the parallelism of the revised routines is easier than using the actual primitive parallel commands.

For example, the original Force code and the preprocessed code of a *Critical* section and a *Barrier* section are given in Figures 1 and 2. The *Critical* section acts like a mutually exclusive operating system critical section. The library functions, *spin_lock* and *spin_unlock*, are the system-supplied commands to lock and unlock shared memory locations. Spin-locks provide synchronization between the hardware and the processes. The *spin_lock* function does a busy wait until the location specified is free; then it locks the location and returns control to the calling procedure. Similarly, the *spin_unlock* frees the locked cell. Because of the busy waits, these commands are most efficiently used for

Original Force Code:

```
Barrier
  TT = - TT / X(J,J)
End Barrier
```

Preprocessed Code:

```
CALL spin_lock(BARLCK)
 IF (FFNBAR.LT.(NP - 1)) THEN
   FFNBAR = FFNBAR + 1
   CALL spin_unlock(BARLCK)
   CALL spin_lock(BARWIT)
 ENDIF
 IF (FFNBAR .EQ. (NP-1))  THEN
   TT = - TT / X(J,J)
 ENDIF
 IF(FFNBAR.EQ.0) THEN
   CALL spin_unlock(BARLCK)
  ELSE
   FFNBAR = FFNBAR - 1
   CALL spin_unlock(BARWIT)
 ENDIF
```

Figure 2: Example of a Barrier

short critical sections.

The *Barrier* forces all the processes that are executing to go into a busy wait for a spin-lock. Each time a process reaches the barrier, one is added to a count, originally set to zero. When the count is one less than the number of processes, the last process executes the sequential code within the barrier, resets the count, and frees the remaining processes.

Consider the Force code in Figure 3 which defines a *Presched DO*. This is a *DOALL* macro which assigns each iteration of the loop to a different process to be run concurrently. The preprocessed Fortran code using the system primitive commands is shown below it. Here, the variable NP contains the number of processors that have been allocated to the program, while ME is the identifier of the particular process itself and is in the range 1, ..., NP. Also the original Force macros, which are changed to comments by the preprocessor, have been included.

These examples are all taken from the parallel QR Back-substitution routine, PQRSL. They clearly show that the Force code is simple to understand as well as to program.

The Force is currently available on five shared-memory machines: the Encore Multimax, the Sequent Balance 8000, the Alliant FX, the Cray 2, and the Denelcor HEP. This allows portability of parallel programs and provides a common language in which to express a parallel algorithm. Even though the various manufacturers of parallel computers supply quite different primitive parallel commands for Fortran parallel programming, a program written in the Force can be run on more than one machine without any changes. This is another motivation for using the Force.

Original Force Code:

```
      Presched DO 73  I = 1, N-J+1
        II = I + J - 1
        T  = T + X(II,J)*QTY(II)
  73  End Presched DO
```

Preprocessed Code:

```
C     Presched DO 73  I = 1, N-J+1
      DO  73  I = 1 + ME - 1, N-J+1, NP
        II = I + J - 1
        T  = T + X(II,J)*QTY(II)
C  73 End Presched DO
73    CONTINUE
```

Figure 3: Example of a Presched DO

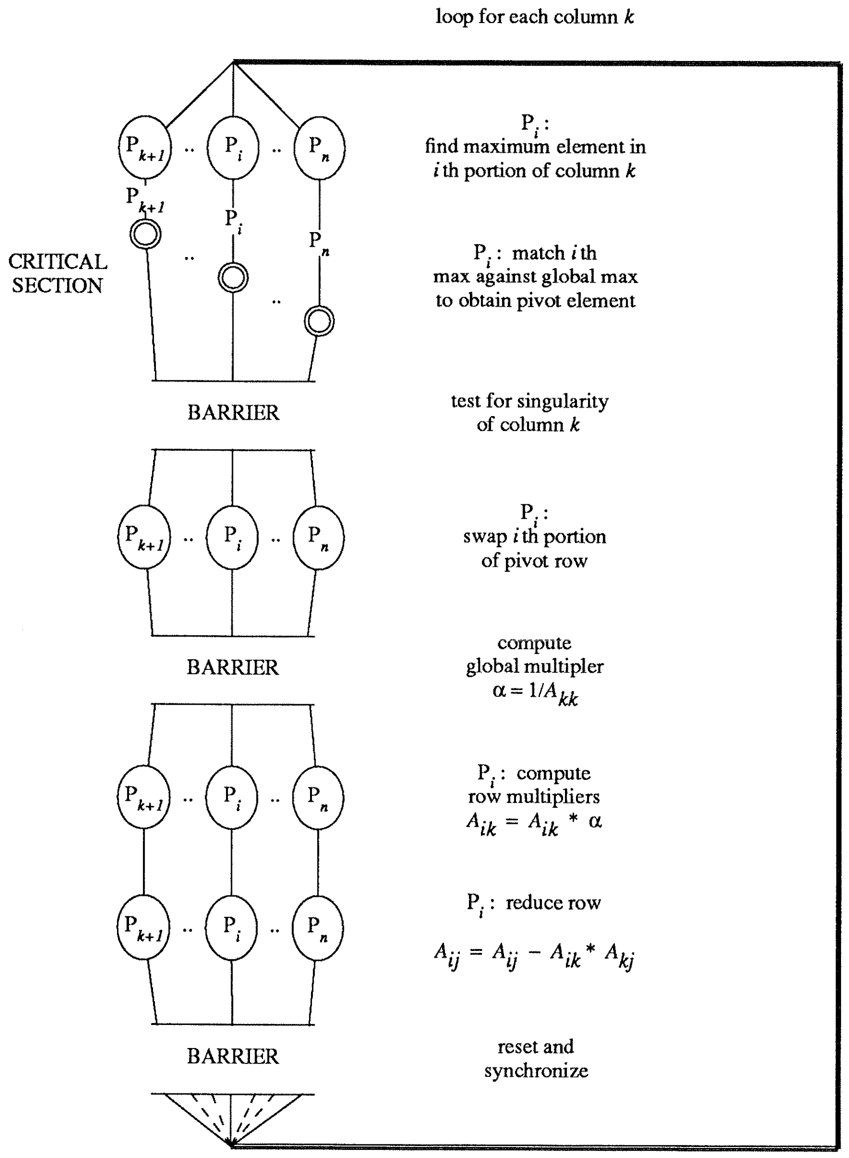

loop for each column k
P_{k+1} .. P_i .. P_n
P_i : find maximum element in ith portion of column k
P_{k+1}
P_i
P_n
CRITICAL SECTION
P_i : match ith max against global max to obtain pivot element
BARRIER
test for singularity of column k
P_i : swap ith portion of pivot row
BARRIER
compute global multipler $\alpha = 1/A_{kk}$
P_i : compute row multipliers $A_{ik} = A_{ik} * \alpha$
P_i : reduce row $A_{ij} = A_{ij} - A_{ik} * A_{kj}$
BARRIER
reset and synchronize

Figure 4: PGEFA Algorithm, Version 1 (Jordan)

For further information on the Force, see [JBA87] [Jor87].

2.2. PGEFA: A Parallel Version of DGEFA

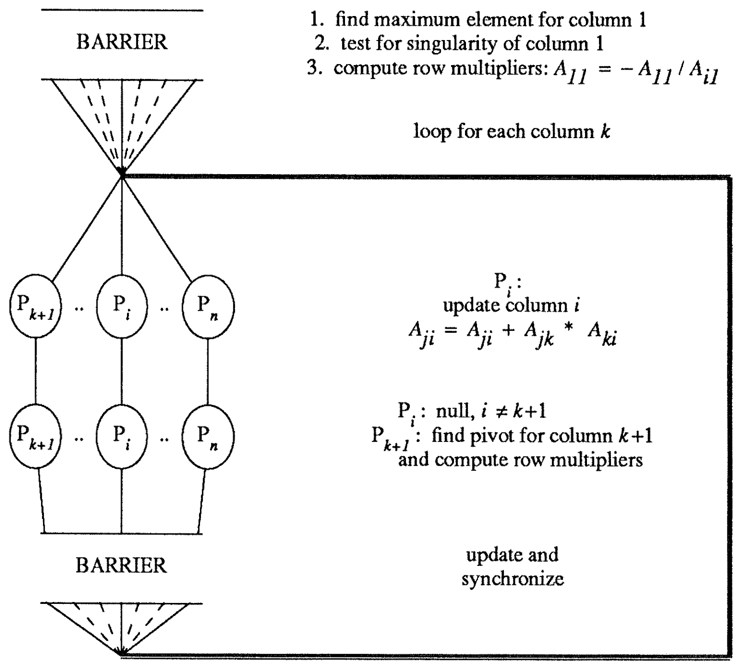
When the original Force macro preprocessor was designed, a parallel version of the Gaussian elimination routine, DGEFA, was written as an example of the implementation [Jor84]. Because of the number of barriers and synchronizations, this program design does not appear to be optimal; so one goal of our project was to write a new version designed to optimize performance. Both of these routines are equivalent to the Linpack DGEFA routine; they use the same input parameters and produce identical output.

Preliminary timing runs of both routines showed that the presumably sub-optimal original Force routine outperformed the new version by up to 40%. The poor performance of the "optimal" program led to a series of experiments designed to enhance the understanding of parallelism on the Multimax. The results of the experiments were used to further optimize the first algorithm, generating a third version of the routine.

2.2.1. Version 1

The design of the algorithm for the original parallel version of the Gaussian elimination routine is shown in Figure 4. The outer loop contains three explicit barriers with one critical section and two pre-scheduled DO loops. (A explanation of the symbols used in the figures which follow is given in Appendix A.)

The main work is done in an inner loop within a loop over all the columns of the input matrix. This inner loop reduces the rows of the matrix in parallel by the global multiplier. It performs $O(n^2)$ operations and accesses the elements in row order. The inner loop is done with a self-scheduled DO loop, which has an implicit barrier before the first execution of the DO loop body and a short critical section used to update the index variables after each iteration. Once execution of the self-scheduled DO loop body begins, execution proceeds asynchronously.



BARRIER
1. find maximum element for column 1
2. test for singularity of column 1
3. compute row multipliers: $A_{i1} = -A_{i1}/A_{11}$
loop for each column k
P_{k+1}
P_i
P_n
P_i:
update column i
$A_{ji} = A_{ji} + A_{jk} * A_{ki}$
P_i: null, $i \neq k+1$
P_{k+1}: find pivot for column $k+1$ and compute row multipliers
BARRIER
update and synchronize

Figure 5: PGEFA Algorithm, Version 2

The overall strategy of Jordan's algorithm is to reduce any operation on a column to a parallel operation, usually with a pre-scheduled DO loop. If this is to be efficient, the cost of making the loop parallel must be minimal. For example, even the search for the maximum (pivot) element in a column is divided between processes. Such a fine-grained task may cost more in synchronization overhead than in execution. Certainly, it will be inefficient for small vectors.

2.2.2. Version 2

Parallel processing experience shows that program constructs which cause processes to wait are major bottlenecks to efficient program performance. For this reason, reducing the number of barriers was a primary focus in efforts to optimize the Version 2 algorithm shown in Figure 5. Theoretically, this provides an optimal algorithm if the amount of computation done is the same as the algorithm without barriers.

The key point of the algorithm is that both the search for the pivot elements and the calculation of the multipliers are done during the parallel reduction of the submatrix. This sets up the next submatrix for reduction, so that as soon as the current submatrix reduction is complete, work can begin on the next. Conceptually, the larger the amount of work done in parallel, the better.

As an example, assume that the machine being used has three processors and the program is working on a large square matrix. Assume also that the current execution point is where the first row is the pivot row, $k = 1$ and that process P_2 is assigned the job of reducing the second column. After the reduction of the second column is complete, process P_2 will search for the pivot element and calculate the multipliers to be used in the next step of the reduction. While process P_2 is doing this, processes P_3 and P_1, are busy reducing columns 3 and 4, respectively. Since P_1 and P_3 have less work than the other processor, they should finish their tasks before P_2 and will be assigned next the tasks of reducing columns 5 and 6, within the construct of a self-scheduled DO loop. This approach maximizes the amount of work done in parallel because calculation of the multipliers is done in parallel with the reduction of the previous submatrix.

2.2.3. Version 3

The third version, given in Figure 6, was designed after experimentation with Versions 1 and 2, as an improvement of the original Force algorithm (Version 1). The improvement is to change access in the main inner work loop from row-ordered access to column-ordered access. The multipliers are precalculated outside the inner main loop to facilitate this.

Otherwise, the strategy of the algorithm is identical to the earlier version. At each step, the work to be done on a column is divided between the processes. The difference between the Version 1 and Version 3 algorithms is in the matrix element access patterns used to reduce the submatrix inside the main inner loop. Calculation of the multipliers adds yet another barrier to this algorithm, for a total of four explicit barriers inside the large outer loop. [Ded86]

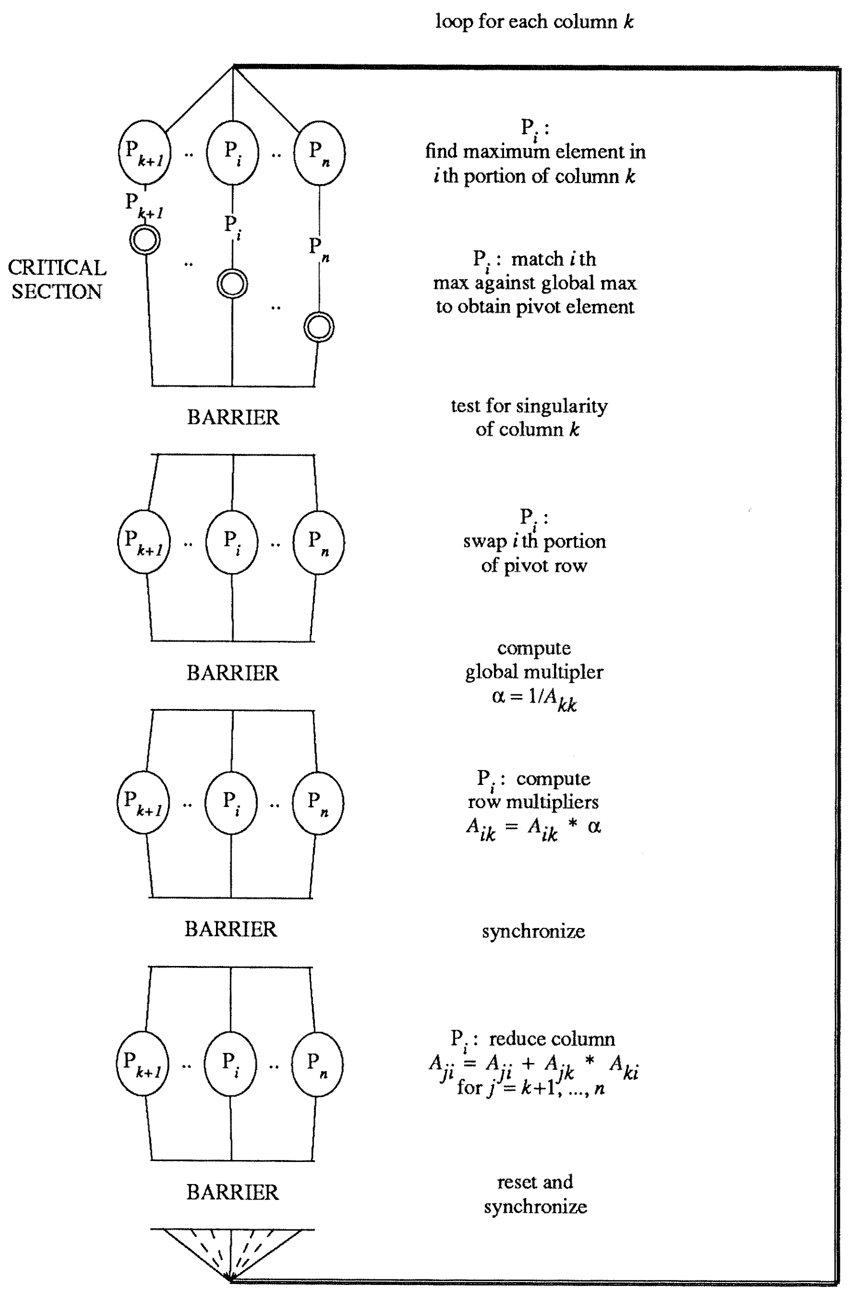


Figure 6: PGEFA Algorithm, Version 3

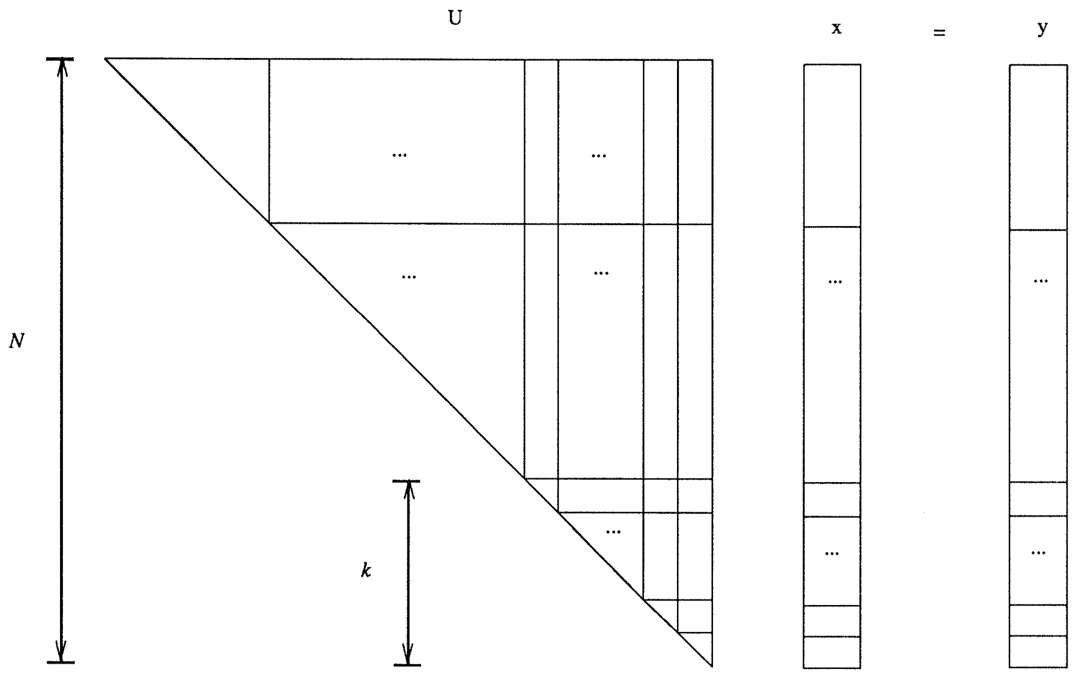


Figure 7: Basic Concept of PGESL Algorithm

2.3. PGESL: A Parallel Version of DGESL

PGESL is a parallel version of the Linpack subroutine DGESL. PGESL solves the real system $Ax = b$ or $A^T x = b$ using a pivot vector and the matrix produced by DGECO, DGEFA, or PGEFA. This parallel general solution is written for the Encore Multimax computer using Fortran 77 and Force macros. The basic strategy for this routine is given in Figures 7, 8, and 9, and is discussed below. Two versions of PGESL are described later. Both versions are designed to be used as a substitute for DGESL and require the same input parameters as DGESL. The only noticeable difference should be in the speed of the two subroutines.

Since the input matrix is in the form of an LU decomposition (a lower and upper triangular matrix superimposed), the problem $Ax = b$ is solved in two parts. First, the elements of the vector y for the problem $Ly = b$ are found. Second, the elements of x are determined for the problem $Ux = y$. Likewise, $A^T x = b$ is determined by solving $U^T y = b$ and $L^T x = y$. The parallel algorithm PGESL takes advantage of the fact that when solving each of the subproblems, several of the matrix operations are independent from one another and can be done in parallel.

The basic algorithm used in PGESL consists of solving a small triangular portion of an upper/lower triangular system sequentially and then updating the remaining rows of the b or y vector in parallel. This, in turn, yields a smaller triangular matrix to solve.

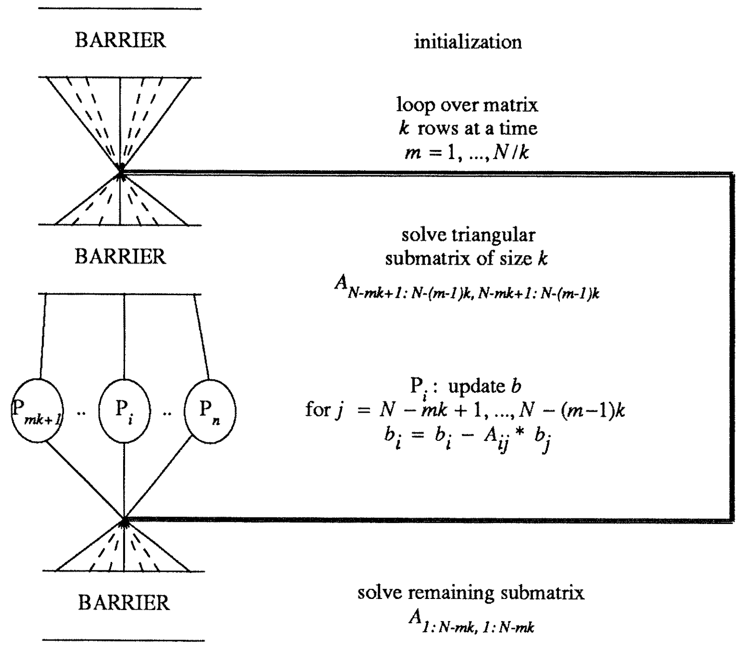


Figure 8: PGESL Algorithm for $Ux = y$

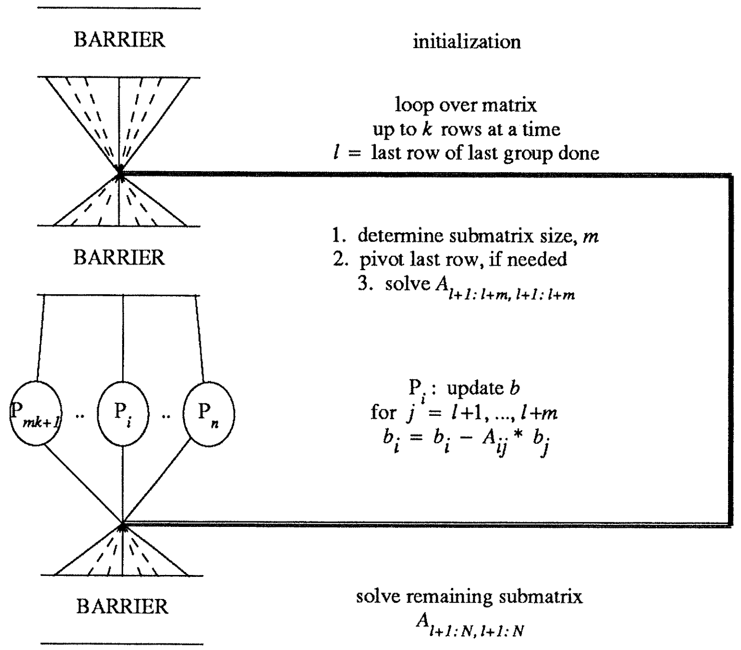

BARRIER
initialization
loop over matrix
up to k rows at a time
l = last row of last group done
BARRIER
1. determine submatrix size, m
2. pivot last row, if needed
3. solve $A_{l+1:l+m,\ l+1:l+m}$
P_{mk+1} .. P_i .. P_n
P_i : update b
for $j = l+1, ..., l+m$
$b_i = b_i - A_{ij} * b_j$
BARRIER
solve remaining submatrix
$A_{l+1:N,\ l+1:N}$

Figure 9: PGESL Algorithm for $Ly = b$, Version 2

For example, suppose the problem to be solved is $Ux = y$ where U is an $N \ x \ N$ matrix. The algorithm finds k elements of x sequentially, say $x(N-k+1) \cdots x(N)$. Then the $y(1) \cdots y(N-k)$ elements are updated in parallel by multiplying the $x(N-k+1) \cdots x(N)$ elements by a portion of the rows of the U matrix. Now the problem becomes $U'x' = y'$, where U' is an $(N-k) x \ (N-k)$ matrix. The process of solving for k elements sequentially and updating the element of y in parallel is repeated until all the elements of x have been computed.

A similar method is used to solve $U^T y = b$. Due to the fact that elements of y may be pivoted, the above algorithm must be modified to compute $Ly = b$.

Version 1 and Version 2 differ only in their treatment of the lower triangular matrix when solving $Ax = b$. They both use the algorithm discussed above for determining $Ux = y$ or $U^T y = b$. When computing $Ly=b$, however, Version 1 solves only one element of y at a time. The processes then update the appropriate elements of the b vector in parallel.

Version 2 uses the modified algorithm for $Ly = b$. It checks the pivot vector within a loop until it finds an element which is to be pivoted or until it has checked k successive elements without encountering any element to be pivoted. At this point, the number of rows the triangle is to contain is determined. Depending upon the values of the pivot vector, 1 to k consecutive elements of y are computed sequentially and then the elements of the b vector are updated in parallel.

The parallel code for $Ux = y$ is the same for both versions and is different from the original Linpack routine. The DDOT function call in the Linpack routine is replaced by in-line code which computes the dot product. [ScS86]

2.4. PQRDC: A Parallel Version of DQRDC

PQRDC is a parallel version of the Linpack subroutine DQRDC. The algorithm is shown in Figure 10. The purpose of PQRDC is to compute the QR factorization of the matrix X. PQRDC can be substituted for the DQRDC routine; the parameters for both are identical. The parallel QR decomposition subroutine is written for the Encore Multimax computer using Fortran 77 and Force macros.

PQRDC and DQRDC use Householder transformations to calculate the QR decomposition. The actual Q matrix is not computed. Since Q is a product of Householder matrices, the vector used to compute each Householder matrix is stored in the original matrix. Because the elements of these vectors and the R matrix overlap on the diagonal, the elements associated with the Householder vectors are stored in an auxiliary vector.

When applying the Householder transformation to the columns of the X matrix, each iteration is independent of the other iterations. Therefore, this portion of the algorithm can be performed in parallel. This is the modification used for the PQRDC subroutine. [Sch86b]

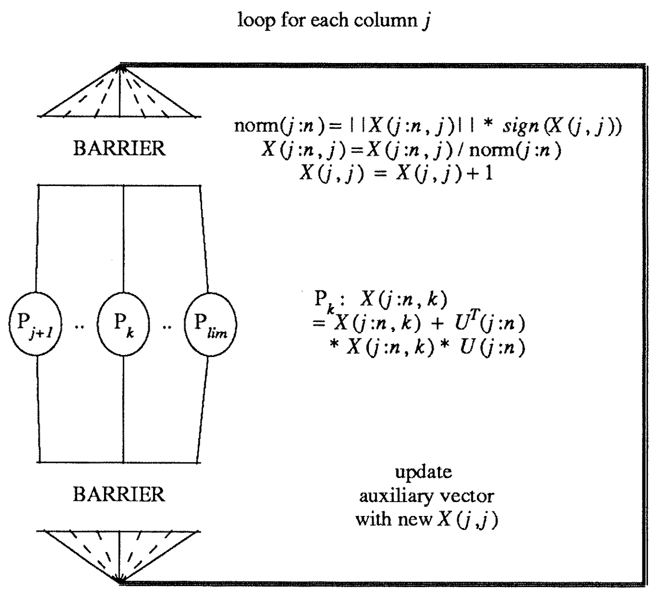

loop for each column j
BARRIER
$\text{norm}(j:n) = ||X(j:n,j)|| * sign(X(j,j))$
$X(j:n,j) = X(j:n,j) / \text{norm}(j:n)$
$X(j,j) = X(j,j)+1$
P_{j+1} .. P_k .. P_{lim}
$P_k: X(j:n,k)$
$= X(j:n,k) + U^T(j:n) * X(j:n,k) * U(j:n)$
BARRIER
update auxiliary vector with new $X(j,j)$

Figure 10: PQRDC Algorithm

2.5. PQRSL: A Parallel Version of DQRSL

PQRSL is a parallel version of the Linpack subroutine DQRSL. This routine is written for the Encore Multimax computer using Fortran 77 and Force macros at the University of Colorado at Boulder. PQRSL is designed to be used as a substitute for DQRSL; the only noticeable difference should be in the speed of the two subroutines. It takes the output of PQRDC (or DQRDC) and computes the coordinate transformations, projections, and/or the least squares solutions, as requested. It requires the same input parameters as DQRSL and produces the same results.

The Linpack DQRSL code is the basis for the PQRSL routine. In that code, three areas are candidates for possible improvement by a parallel algorithm. The first is in the back substitution computation of the vector b which solves the least squares problem of minimizing $||y - X_k b||$, where $X = QR$ and y is some input vector. This is replaced by a wavefront algorithm, as illustrated in Figures 11 and 12, since each value in the result vector depends only on the values in the matrix in the subtriangle below it. An additional global array of multivalued semaphores keeps track of the rows and columns of the matrix already visited.

Other results from DQRSL include Qy, $Q^T y$, Xb, and the residual $r = y - Xb$. These vectors are computed as follows: each element of the vector is determined using the result of the dot product of the vector with a column of the Q matrix. DQRSL performs this computation in a loop over the columns, using the BLAS subroutines, DDOT and DAXPY. The parallel version again loops over the columns, obtaining the dot product and doing the updating, but does both of these computations utilizing pre-scheduled DO loops. This algorithm is shown in Figure 13.

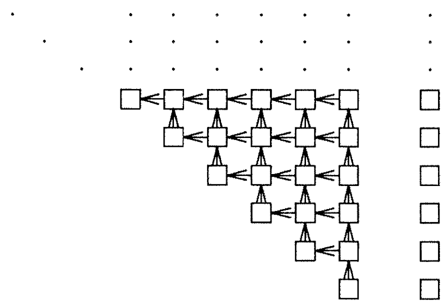


Figure 11: Basic Concept of the PQRSL Wavefront Algorithm

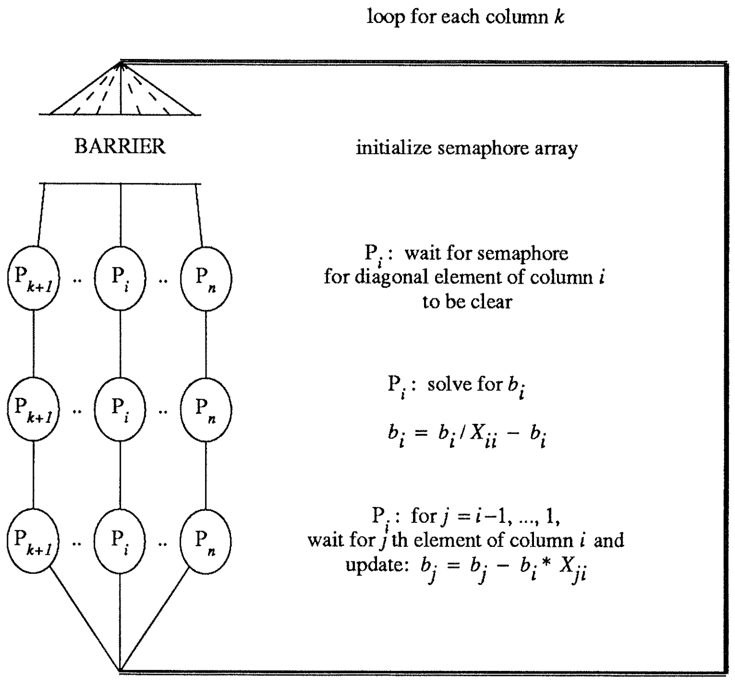

loop for each column k
BARRIER
initialize semaphore array
P_{k+1} .. P_i .. P_n
P_i : wait for semaphore for diagonal element of column i to be clear
P_i : solve for b_i
$b_i = b_i / X_{ii} - b_i$
P_i : for $j = i-1, ..., 1$, wait for j th element of column i and update: $b_j = b_j - b_i * X_{ji}$

Figure 12: PQRSL Algorithm, Computation of b

The output vectors are initialized by copying the input vector y or the computed vector $Q^T y$. The DQRSL routine does this with calls to the BLAS routine, DCOPY; the parallel version uses pre-scheduled DO loops. As a result of these changes to DQRSL, the parallel version, PQRSL, does not require the Linpack BLAS subroutines, DCOPY, DDOT, and DAXPY. [Sch86a]

3. RESULTS

Graphical representations of the results obtained by both the original Linpack and the revised versions executed using different numbers of processes are shown in Appendix B. Notice that the speedups are less than linear.

The runs using seven, eight or more processes from the eight available processors show degradation in performance. This is mainly due to swapping. The Encore is a time-shared machine; the operating system must periodically access at least one processor to keep the system running. The Force allows any number of processes to be requested (even above the actual number of processors available); however, if more processes are activated than the number of physical processors, the overhead of context switching between processes becomes quite high. Also, the Force barriers and critical sections are implemented in a sequential manner by spin-locks, which should only be used for short waits; the greater the number of processes in use, the more serious a bottleneck they become [BeJ88].

Most of the runs were made when the load on the machine was low. However, they were not necessarily done in a single-user environment. This also relates to the poor performance for a higher number of processes.

3.1. PGEFA vs. DGEFA

Four aspects of timing the three algorithms were considered: timing variations, performance predictability, comparisons with Linpack, and speedup. Most of the timing was done when the number of users on the system was low (less than four). Graphical results can be found in Appendix B.7.1.1 and B.7.1.2.

During the experiments, variations of over 7% were observed on identical runs conducted back to back with no other users on the system. There seem to be four factors that affect run time variation:

(1) the number of users: the fewer the better.

(2) the number of processes used: it is best not to exceed the actual number of physical processors.

(3) the run time: programs with run times longer than ten seconds show less variation.

(4) chance.

Most of the results generated with other users on the system match closely with results generated when no other users were on the system.

For the experiments comparing the standard Linpack DGEFA routine against the parallel versions, repeated runs were made with no one else on the system and the system load factor was recorded after each run. Fifty Gaussian elimination runs were conducted. These were done in groups of five, with each set representing five runs with the same number of processes and a 100x100 matrix as input. Because each set was small, the

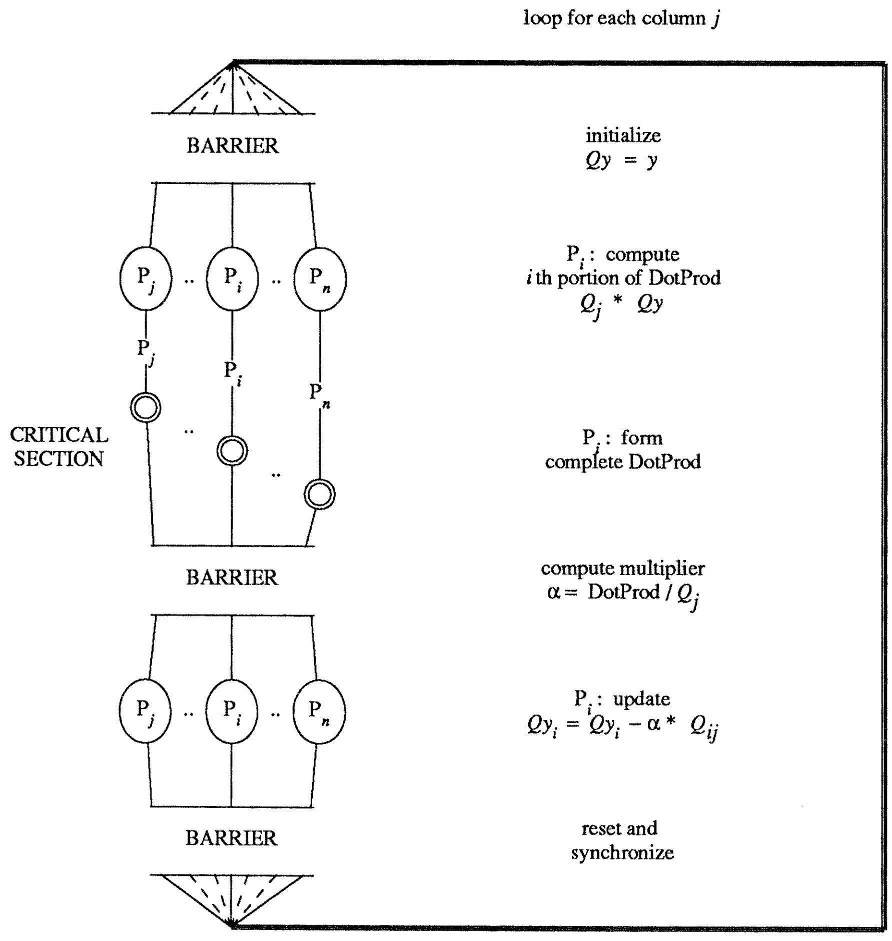


Figure 13: PQRSL Algorithm, Computation of Qy Vector (Similar to computation of $Q^T y, Xb$, and the Residual of $y - X_k b$

standard statistical measure of variance is not used to measure the timing variations. Instead, the percentage difference between the fastest and slowest execution times is used as a measure of run time variation.

Using this measure, the following are typical variations with no other users on the system. Six of ten groups have variations less than 0.6%. Three of the ten groups show variations between 1.2% and 1.7%. The tenth group has a variation of 3.5%; no reason for the difference could be found.

For comparison, several single-process Linpack runs were made with no other users on the system. The average time of five runs for matrices of dimension 20x20, 50x50, 100x100, and 150x150 was compared to the averages for the other algorithms. Appendix B.7.1.1 shows bar graphs of the results when all the runs were pivoted; Appendix B.7.1.2 shows the results when each run had only a single pivot. Also some multiprocessor run results for 200x200 matrices are shown. For matrices of dimension 100 or more, the Linpack version runs in 50% of the time of the Version 1 algorithm, 33% of the Version 2 algorithm, and 52% of the Version 3 algorithm, all executed as a single process. As expected, parallelism has a cost.

The best parallel algorithm requires two processes to match the single process Linpack performance, but all the algorithms using three processes are faster than the Linpack performance. More processes substantially improve on the run times, as long as the processors are physically available. For example, the Version 1 algorithm will solve a 100x100 matrix in 4.9 seconds with seven processes, compared to 17.2 seconds for the Linpack sequential version, a speedup factor of 3.5.

The time complexity of the Linpack DGEFA routine is of order $O(n^3)$ for an $n \ x \ n$ matrix. As can be seen by Figure 4, Version 1 of PGEFA spreads the work over p processes, reducing the time complexity by $1/p$. This is apparent in the bar graphs given in Appendix B, although the speedup is less than perfect because of the synchronization and required sequential code in the routine. Version 3 is basically the same algorithm as Version 1, but should show a little more speedup because of the contiguous access of the column elements which require less paging than the row-ordered access of Version 1. The results in Appendix B agree.

While Version 2 also spreads the work of updating the matrix elements over the p processes, it also introduces additional code within the loop to avoid extra barriers. This essentially sequential control flow code is duplicated by all the processors. Thus, the speedup is less than that of Version 1 and 3.

3.2. PGESL vs. DGESL

Four experiments were conducted using various versions of the PGESL routine. The DGESL and PGESL routines are capable of running two different "jobs". The first job is to solve the real system $Ax = b$; the second, to solve $A^T x = b$. The user can specify which job is to be done by setting an input parameter. The solve routines also perform column pivoting based upon the values contained in a pivot vector.

The purpose of the first experiment was to find an optimal value, called SIZE, for the number of elements which should be solved sequentially. The experiment used 1, 2, 4, 6, and 8 processes while varying the SIZE from 1 to 15. The tests were run three times on two different matrices of size 150 x 150 and 100 x 100. The results, in Appendix B.7.2.1, which are the mean times for the three runs, show the optimal value for SIZE to be around six. The version of PGESL used for the first experiment is only capable of solving $Ax = b$; it does not perform pivoting.

The remaining experiments, each consisting of five runs on three different matrices, were conducted for the Linpack DGESL routine and the two PGESL routines. The sizes of the input matrices were 50 x 50, 100 x 100, and 150 x 150. The experiments consisted of running both jobs on the three matrices for pivoting every column, pivoting no columns, and pivoting every fourth column. The experiments were timed for 1, 2, 4, 6, and 8 processes. The results, for runs with no pivoting and with full pivoting, are the mean times of the three middle timing values found and are shown in Appendix B.7.2.2 and B.7.2.3.

The results show that the sequential Linpack version runs appreciably faster than the two parallel versions using one process. It is not until the parallel versions use four to six processes that they begin to be faster than the Linpack version. The speedup obtained by the parallel versions over the Linpack version is only a little over one. Part of the explanation is memory contention in the way that the parallel versions access the shared array elements. The Linpack code accesses the elements of the input matrix in a column-wise fashion; the parallel versions access the elements by rows.

For an $n \times n$ matrix, the Linpack routine, DGESL, has a multiplication count of $(n^2+n)/2$. As can be seen by the algorithms shown in Figures 8 and 9, the only parallelization done for PGESL was in the updating of the matrix as each value was determined. This reduces the number of operations for that part of the algorithm by p, the number of processors, making it $O(n^2/p)$. However, because the code which computes the solution for each submatrix is sequential and requires synchronization, the speedups shown in the bar graphs in Appendix B are less than p.

3.3. PQRDC vs. DQRDC

Three different experiments were conducted involving sequential and parallel versions of the QR decomposition. Seven matrices were used in each experiment, varying from 50x50 to 200x100. Five runs were made of each experiment; the results given in Appendix B.7.3.1 through B.7.3.3 are the means of the three middle timing values obtained for each experiment.

The first experiment used two different parallel versions of the QR decomposition along with the Linpack routine. The difference between the two versions is that the first contained one Force *barrier* while the other contained two *barrier*s. (See Figures 10 and 14.) The parallel versions used for this test did not contain any of the pivoting code necessary to be compatible with the Linpack routine. The experiment was basically intended to see if there was a noticeable difference between using one barrier or two barriers. The results, shown in Appendix B.7.3.1, indicate that there is virtually no

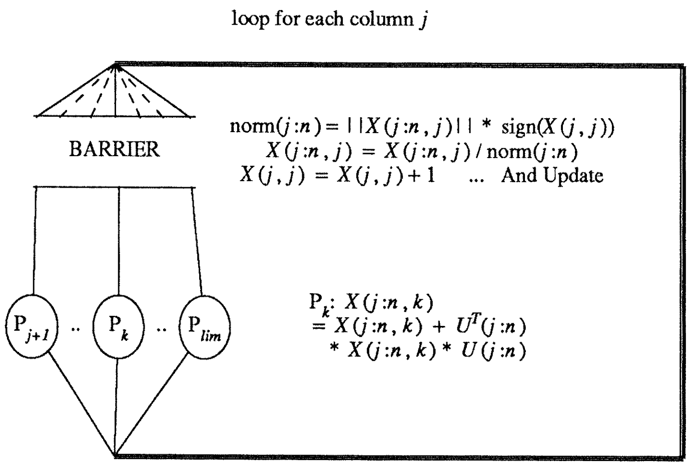

loop for each column j
BARRIER
$\text{norm}(j:n) = ||X(j:n,j)|| * \text{sign}(X(j,j))$
$X(j:n,j) = X(j:n,j)/\text{norm}(j:n)$
$X(j,j) = X(j,j)+1$ ... And Update
P_{j+1} .. P_k .. P_{lim}
P_k: $X(j:n,k) = X(j:n,k) + U^T(j:n) * X(j:n,k) * U(j:n)$

Figure 14: PQRDC Algorithm, Version 2

difference and that the barriers are fairly cheap to use. This was consistent with the results of the PGEFA experiments.

The second experiment tested a fully Linpack compatible parallel version of the QR decomposition when pivoting is not requested. The last experiment tested the parallel version when pivoting is requested. (When pivoting is requested, more steps are done sequentially than when pivoting is not requested.) It is expected that the job with no pivoting should require less time than the job with pivoting. The results given in Appendix B.7.3.2 and B.7.3.3 confirm this.

For an n x m matrix, the sequential version of the routine, DQRDC, requires about

$$nm^2 - m^3/3$$

multiplications as well as additions. For square matrices, $n = m$, this becomes

$$2m^3/3.$$

The portion which was rewritten into a parallel algorithm applied the Householder transformations and computed norms for each column. These are both operations of $O(nm)$ and are done m times; so reducing the executing time for these portions of the routine by a factor of p, the number of processes, should have a great effect. When running with six processes, the speedup shown by the graphs in Appendix B for PQRDC is four or better for large matrices, as expected.

All the test cases used had $n \geq m$, and for a given n, the effect of these changes should be more apparent on the square matrices, where $n = m$. This is indeed true, as the speedup approaches five for the largest square matrix used, 150x150.

The results for the parallel versions running as a single process are slightly slower than the results for the Linpack DQRDC version. This, with the less than perfect speedup, reflects the overhead cost of parallelism.

3.4. PQRSL vs. DQRSL

Four experiments were conducted using various versions of the PQRSL routine. The results of these experiments are in Appendix B.7.4.1 and B.7.4.2. Seven matrices were used in each experiment, sized from 50x50 to 200x100. Each run was repeated five times; the average of these times with the best and worst runs removed is used in the graphical results. In all cases, the subroutine performed all the possible computations.

The purpose of the first experiment was to find the improvement in running time (if any) achieved by the parallel routine compared to the original Linpack routine for the seven different matrices and 1, 2, 4, 6, 8, and 10 processes. Experiments 2 through 4 were done to see how each of the three different portions of PQRSL that were parallelized affected the overall performance. (See Figures 12 and 13.) Experiment 2 ran a version of PQRSL without the wavefront algorithm; instead, the original Linpack DQRSL code computed the result vector. Experiment 3 used the DQRSL nonparallel computations of the output vectors with calls to DDOT and DAXPY instead of the parallel code. Finally, Experiment 4 substituted the pre-scheduled DO loops for initialization of the vectors by the original DQRSL calls to DCOPY. These three experiments were run on the same seven matrices as Experiment 1 for 1, 2, 4, 6, and 8 processes.

Appendix B.7.4.1 compares the average running time for each matrix size using differing numbers of processes, under the four experiments, against the original Linpack version, DQRSL. The final graph, Appendix B.7.4.2, shows the effect of matrix size on the different experiments using six processes.

With only one process, the sequential Linpack version, DQRSL, runs about twice as fast as the parallel version, PQRSL, with DQRSL doing better as the matrices increase in size. Of course, there is some overhead needed for the Force routines and some cumbersome code required by the parallel algorithms used in PQRSL. For instance, the wavefront algorithm necessitates the use of an array of semaphores; the array initialization and testing is expensive when only one process is available. As the number of processes is increased, the efficiency of PQRSL improves; given four or more processes, it takes about only half the time of DQRSL.

The results of Experiments 2 through 4 show that the parallel computations of the output vectors have the most effect on speeding up the execution time. In fact, without that parallel part of PQRSL, the routine takes *longer* than DQRSL on matrices smaller than 150x100, on as many as six to eight processes.

However, the other two parallelizations provide some improvement in running time, as the complete PQRSL routine performs better than any version omitting one of the three parallelizations. While the wavefront algorithm for back substitution improves the efficiency of the routine somewhat, it does not seem as effective as expected. However, only one of the five output vectors is computed using this method. Thus, it should be expected that the results show a one to four difference in the speedup caused by each experiment. Finally, the pre-scheduled DO initializations which replaced the DCOPY calls to the BLAS routines seem to have the least effect on the relative speedup.

The following provides a more detailed analysis. Consider an $n \; x \; m$ matrix $X = QR$; let $k = \min\{n, m\}$, and let $Q = (Q_1, Q_2)$, where Q_1 is of length k. The five possible output vectors are Qy, $Q^T y = \boxed{\begin{matrix} Q_1^T y \\ Q_2^T y \end{matrix}}$, the least squares approximation of $Xb = Q_1 Q_1^T y$, the residual $r = Q_2 Q_2^T y$, and the solution $b = R^{-1} Q_1^T y$. The Linpack report [DBM79] gives the multiplication count for these vectors in DQRSL to be as follows:

Qy	$(2n-k)k$
$Q^T y$	$(2n-k)k$
Xb	$2(2n-k)k$
r	$2(2n-k)k$
b	$(2n-k)k + k^2/2$

Because all the examples used had $n \geq m$, m can replace k in all the formulas above. In the case where $n = m$ (X is a square matrix), the formulas become simpler. This provides the multiplication counts in the following table.

	Rectangular X	Square X
Qy	$(2n-m)m$	n^2
$Q^T y$	$(2n-m)m$	n^2
Xb	$2(2n-m)m$	$2n^2$
r	$2(2n-m)m$	$2n^2$
b	$(2n-m)m + m^2/2$	$3n^2/2$

The addition counts are similar.

Since the test data produced all the possible output vectors, it would seem that the total number of multiplications would be the sum of these formulas:

$$7(2n-m)m + m^2/2.$$

However, the vector, $Q^T y$, is used in the computations of the vectors, b, Xb, and the residual. This reduces the count to

$$4(2n-m)m + m^2/2.$$

By spreading out the work of the parallel matrix-vector computation over p processes, it would be hoped to divide the first term by p. However this revised algorithm requires a critical section to put together the dot product, and the actual factor is a little less than p. Since this term has a large factor, namely 4, the parallel matrix-vector computation has more effect than the other parallelizations incorporated by PQRSL.

It would also be hoped that the wavefront algorithm would be reduced by a factor of p; however, with start-up time, necessary synchronization, and semaphore overhead (m initial assignments plus $m^2/2$ comparisons and decrements), it is less efficient. Also, it has less effect on the overall speedup, as it has a smaller term in the operation count of the algorithm.

In the case where $m < n$, the first term will be larger, as $2n - m > n$. Hence, improvement of this term will appear to have more effect. This can be seen by comparing the bar graphs in Appendix B of the square matrices against those of the rectangular matrices. Similarly, improvement of the second term, that is, inclusion of the wavefront algorithm, will have more effect on the square matrices.

The parallel initialization has the smallest effect. This modification merely spreads the work of copying the k elements of a vector over p processes, while the original version called the subroutine, DCOPY, to do the same thing. Since the largest number of processes used that proved consistent in these tests was six, we would expect the time for this copying to be $k/6$, especially as a subroutine call has been eliminated. However, the parallel version was written to copy only one element per loop iteration, while DCOPY is carefully designed to perform seven copying operations per iteration, giving it a smaller percentage of loop overhead. A better test would have been to have used the DCOPY loop body in the parallel version. Nevertheless, only five vectors are copied in this way: 4 of length n and 1 of length m. So the original operation count for this part of the code was $4n + 1m$ and has been reduced to $(4n + m)/p$ or $(4n + m)/6$. Since $4n + m \leq 5n \ll n^2$, we should not expect to see much effect in comparison to the other work being performed by this routine.

4. SUMMARY

This project served as a good introduction to parallel programming. It provided experience in the possible speedups, problems, and limitations of parallel computing. Further, it gave the Force group working under Prof. Jordan useful feedback concerning the operation of the Force on the Encore.

4.1. Guidelines

Some guidelines for parallelizing Fortran programs using the Force on shared-memory non-vector machines with a small number of processors emerge from this project. These programming methodologies follow:

(1) Column ordered access is faster than row ordered access. This is standard for any Fortran implementation.

(2) A self-scheduled DO loop usually works best to divide the work between processes. Simple vector operations (on non-vector machines) are performed better by pre-scheduled DO loops.

(3) Any operation performed even a few times on a vector is worth doing in parallel, provided the vector is long enough and the application computer does not have special vector hardware.

(4) Any complicated task should be reformulated in at least two different parallel algorithms and tested before the final algorithm choice is made.

(5) Speedup seems greatest when all the processes are performing essentially identical work, with few synchronizations.

(6) To debug a parallel program, begin by running it as a single process, i.e., in a sequential manner.

It should also be noted that the time to learn the Force constructs and to embed them into the existing Fortran code was only a small part of the total effort of the project. This suggests the ease and naturalness of using the Force to describe Fortran parallel algorithms.

Debugging Force programs, as in debugging any parallel program, proved difficult. The preprocessor added additional lines of code to the program, so Fortran compiler errors referred to incorrect line numbers. [A new Force compiler now being written should correct this problem.] Usually, any new code was run as a single process to weed out sequential code errors before doing parallel testing. Most parallel problems were found to be the incorrect use of shared or critical variables.

4.2. Further Work

Since these routines are in the Force, it would be quite easy to port the programs to other parallel computers. In this way, the speedup of the different computers, relative to the efficiency of their Fortran compiler and primitive parallel commands, could be compared [Pon88].

In hindsight, it is easy to see places where the parallel code could have been made more efficient. For instance in PQRSL, the code which replaces the DAXPY calls in DQRSL only computes one vector element at a time. The DAXPY routine does four elements within a loop iteration. By unrolling the parallel loop body to allow four or so computations instead of one, slightly more speedup should be obtained. In general, many of the loops which were parallel over all the processes had small bodies and probably would have benefited from some unrolling.

Parallel versions of the BLAS routines have been developed for many parallel machines [Har87]. These are the routines which do most of the work of the Linpack routines, like SAXPY ($ax + y$) or SDOT ($x \cdot y$), for vectors x and y. Hence, creating a parallel version of the Linpack routines is not as important as it might have been.

However, except for the PQRSL and parts of the PGESL routines, most of the work done for this project was on the structure or algorithm design of the main subroutines, almost ignoring the BLAS routines. It would be interesting to observe the improvements possible by running these routines with the parallel set of BLAS routines. Furthermore, the comparison of the PGESL and PQRSL routines with and without the parallel BLAS routines might be worth some experimentation.

4.3. Acknowledgements

We would like to thank Muhammed Benten for patiently explaining the Force constructs and the operation of the Encore Multimax and for keeping us well-informed as the Force macro-processor was updated.

5. BIBLIOGRAPHY

[ArJ87] N. S. Arenstorf and H. F. Jordan, ''Comparing Barrier Algorithms'', CSDG 87-3, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, June 1987.

[BeJ88] M. S. Benten and H. F. Jordan, ''Multiprogramming and the Performance of Parallel Programs'', CSDG 88-2, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, Jan. 1988.

[Ded86] F. M. Dedolph, ''Experiments in Parallel Programming on the Multimax'', for L. D. Fosdick, University of Colorado, Boulder, CO, Oct. 1986.

[DBM79] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia , 1979.

[Don81] J. J. Dongarra, ''Some Linpack Timings on the CRAY-1'', *Tutorial on Parallel Processing*, 1981, 363-380.

[Enc87] Encore Computer Corporation, *Multimax Technical Summary*, Encore Computer Corporation, Marlboro, MA, 1987.

[Har87] W. J. Harrod, ''Parallel Programming with the BLAS'', in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon and R. J. Douglass (editor), The MIT Press , 1987, 253-276.

[Jor84] H. F. Jordan, ''Structuring Parallel Algorithms in an MIMD, Shared Memory Environment'', CSDG 84-2, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, Sep. 1984.

[JBA87] H. F. Jordan, M. S. Benten, N. S. Arenstorf and A. V. Ramanan, *Force User's Manual, Revised edition*, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO , June 1987.

[Jor87] H. Jordan, ''The Force'', in *The Characteristics of Parallel Algorithms* , L. H. Jamieson, D. B. Gannon and R. J. Douglass (editor), MIT Press, Cambridge, MA, 1987, 395-436.

[Pon88] C. G. Ponder, ''Benchmark Semantics'', *SIGPLAN Notices 23*, 2 (Feb. 1988), 44-48.

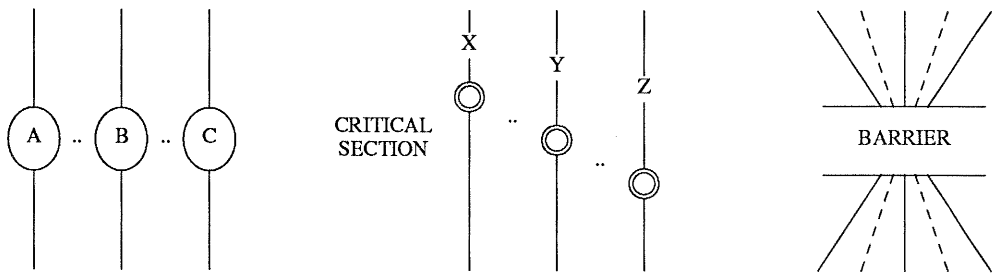
[Sch86a] C. J. C. Schauble, ''PQRSL Report'', for L. D. Fosdick, University of Colorado, Boulder, CO, Aug. 1986.

[Sch86b] B. Schlaman, ''PQRDC Report'', for L. D. Fosdick, University of Colorado, Boulder, CO, Aug. 1986.

[ScS86] B. Schlaman and C. J. C. Schauble, ''PGESL Report'', for L. D. Fosdick, University of Colorado, Boulder, CO, Aug. 1986.

6. APPENDIX A: Key to Symbols Used in Parallel Figures

There are three basic symbols used in the figures describing the parallel algorithms shown in this paper. They represent processes executing in parallel, critical sections, and barriers. These symbols are connected by lines, in the manner of sequential flow charts. Thick lines represent the flow for all the processes, as for the return to repeat a loop. The meaning of dots is much like ellipses; dashed lines are used to imply the flow of those processes not drawn, but active.



The first symbol above shows a number of processes operating in parallel. Each process may be labeled. Since this represents a Force algorithm, each process is executing essentially the same code, differing only by the process identifiers.

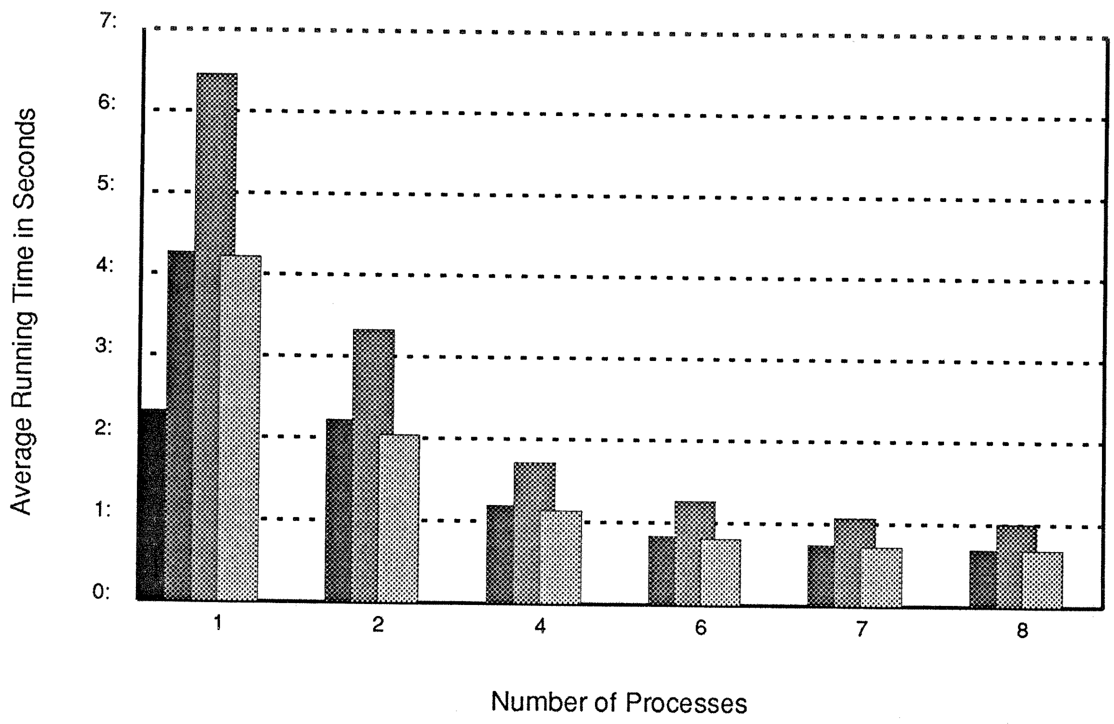
The second symbol represents a critical section. Each process, which is labeled on the arc coming into the section, has to wait its turn to enter the section of mutual exclusion. This accounts for the circles being staggered and not in a straight line.

The third and final figure shows the Force barrier. All processes must synchronize at this point. Each process pauses upon reaching the barrier. When all the processes have halted, the last process to reach this point executes the code within the barrier, if any. Then all the process are allowed to continue on.

7. APPENDIX B: Graphical Results

7.1. PGEFA

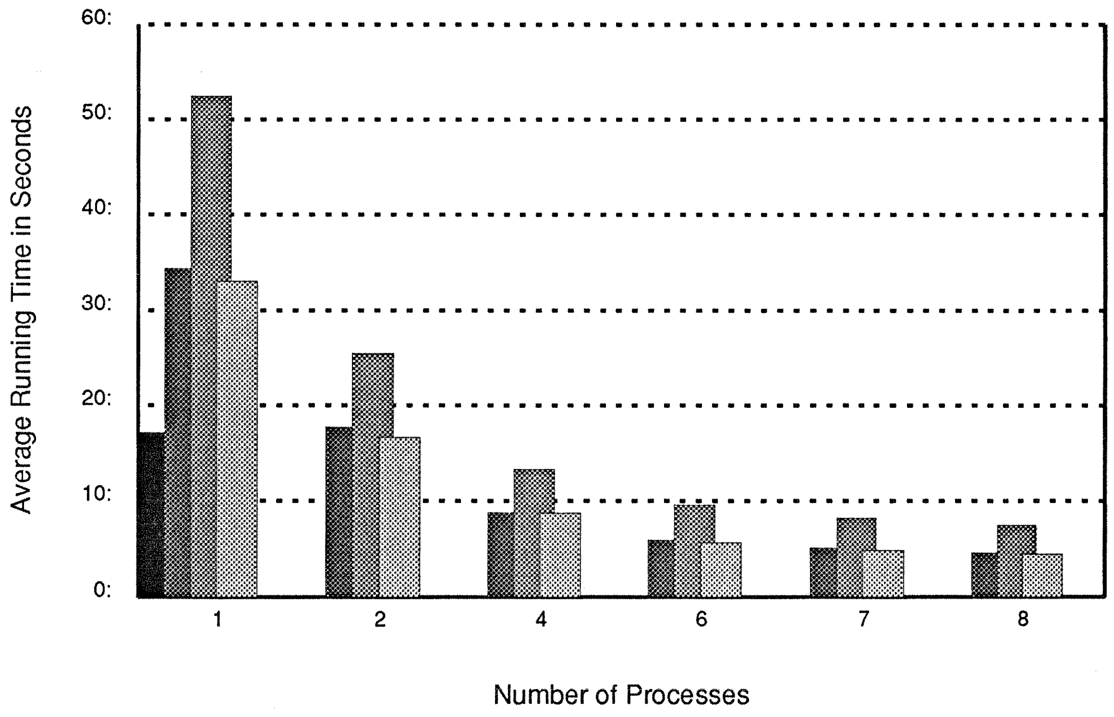
7.1.1. DGEFA vs. PGEFA, All Rows Pivoted



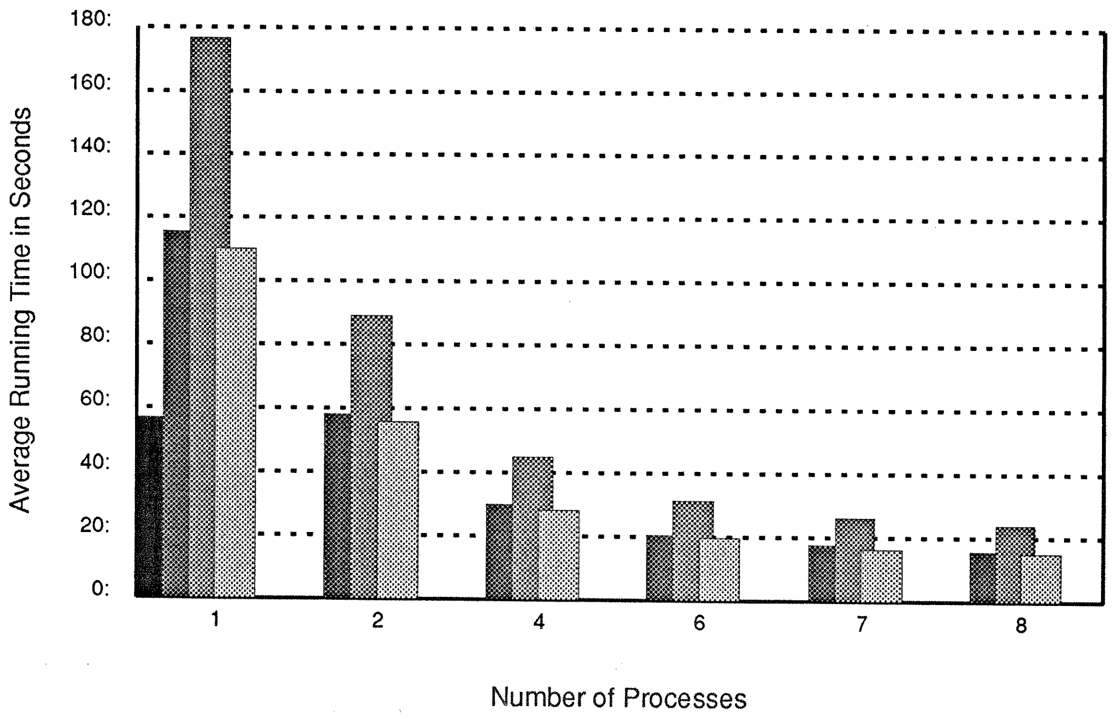
7.1.1.1. Matrix Size: 50 x 50

KEY: ▇ DGEFA ▓ Version1 ▒ Version2 ░ Version3

DGEFA vs. PGEFA, All Rows Pivoted (cont.)

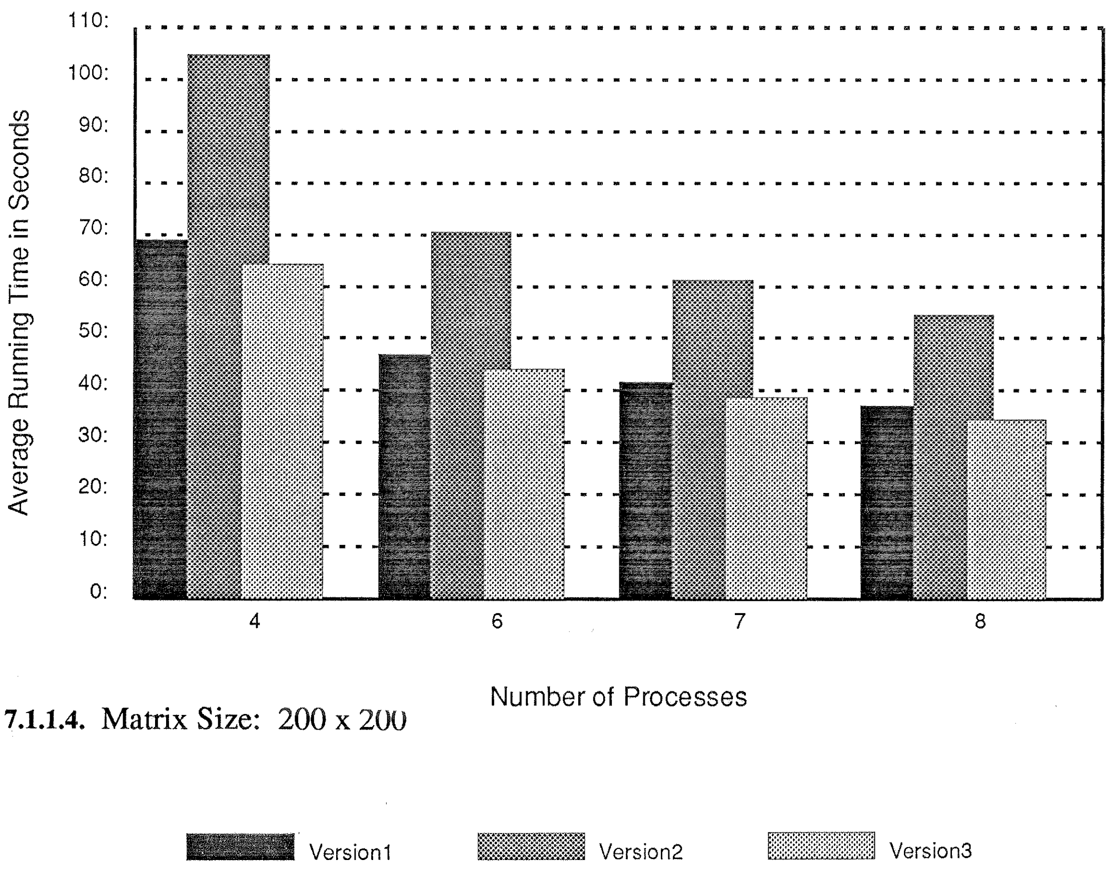


7.1.1.2. Matrix Size: 100 x 100



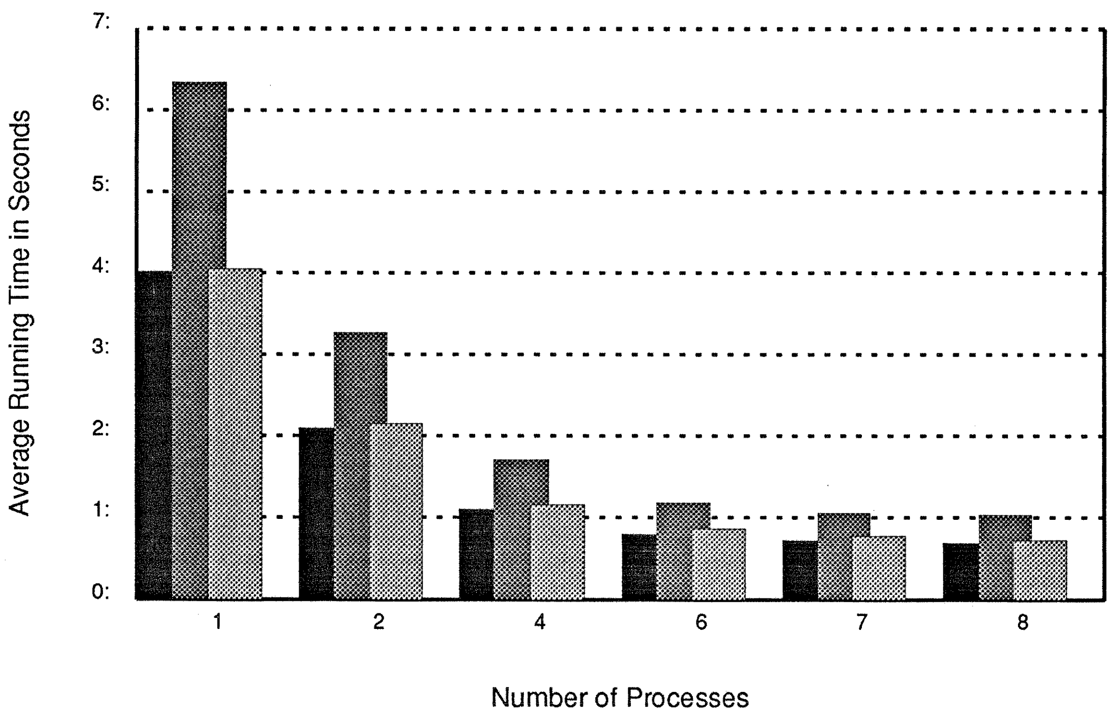
7.1.1.3. Matrix Size: 150 x 150

DGEFA vs. PGEFA, All Rows Pivoted (cont.)



7.1.1.4. Matrix Size: 200 x 200

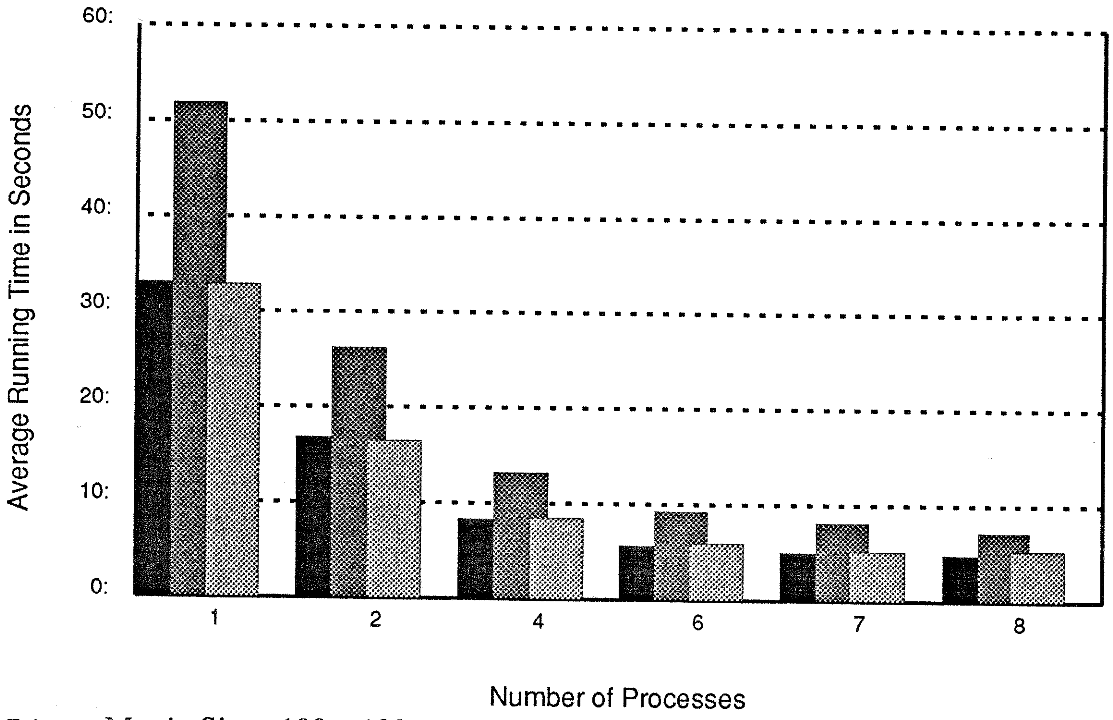
7.1.2. DGEFA vs. PGEFA, Only One Row Pivoted



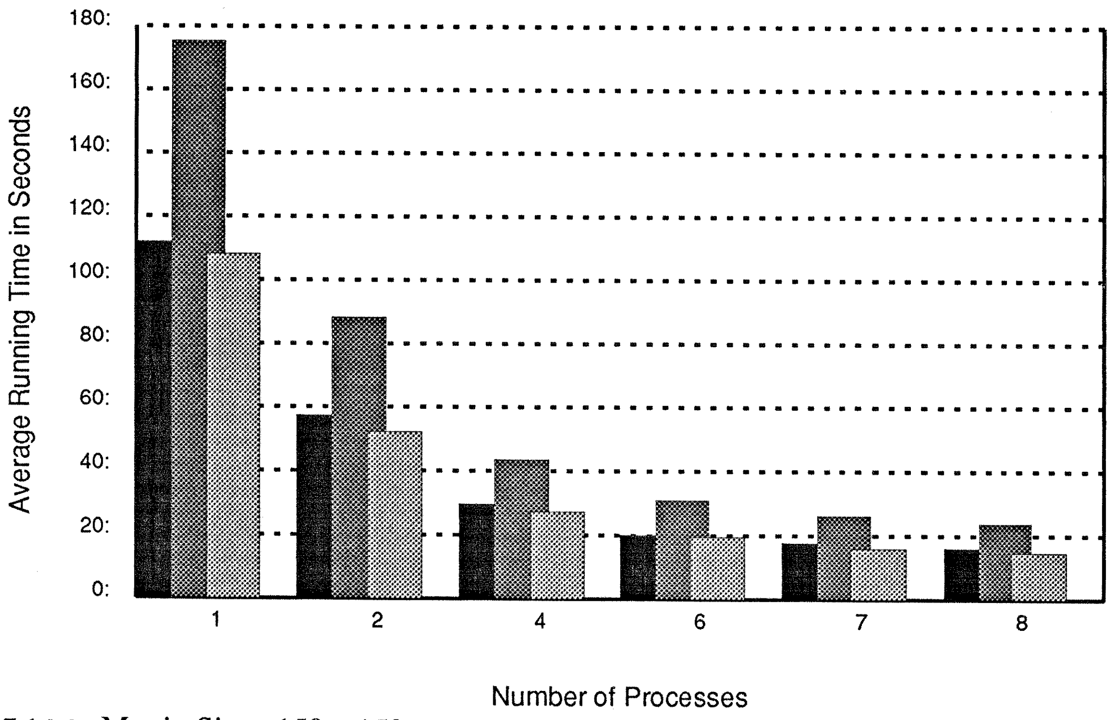
7.1.2.1. Matrix Size: 50 x 50

KEY: ■ Version1 ▩ Version2 ▨ Version3

DGEFA vs. PGEFA, Only One Row Pivoted (cont.)

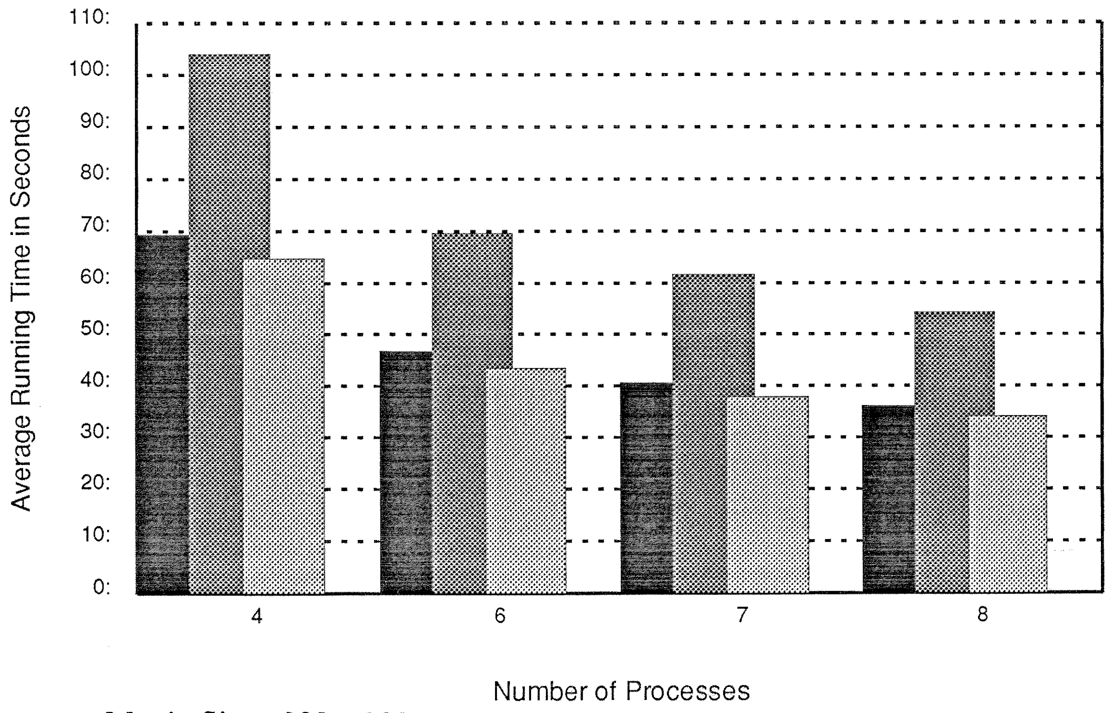


7.1.2.2. Matrix Size: 100 x 100



7.1.2.3. Matrix Size: 150 x 150

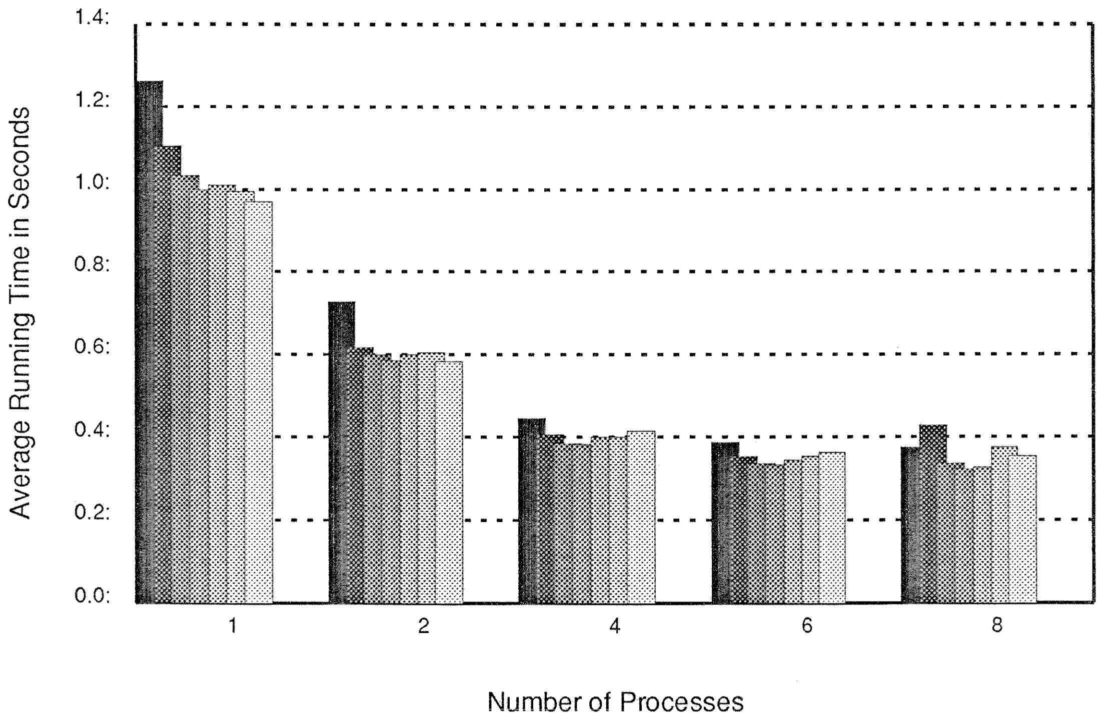
DGEFA vs. PGEFA, Only One Row Pivoted (cont.)



7.1.2.4. Matrix Size: 200 x 200

7.2. PGESL

7.2.1. Optimal Size of Submatrix for PGESL

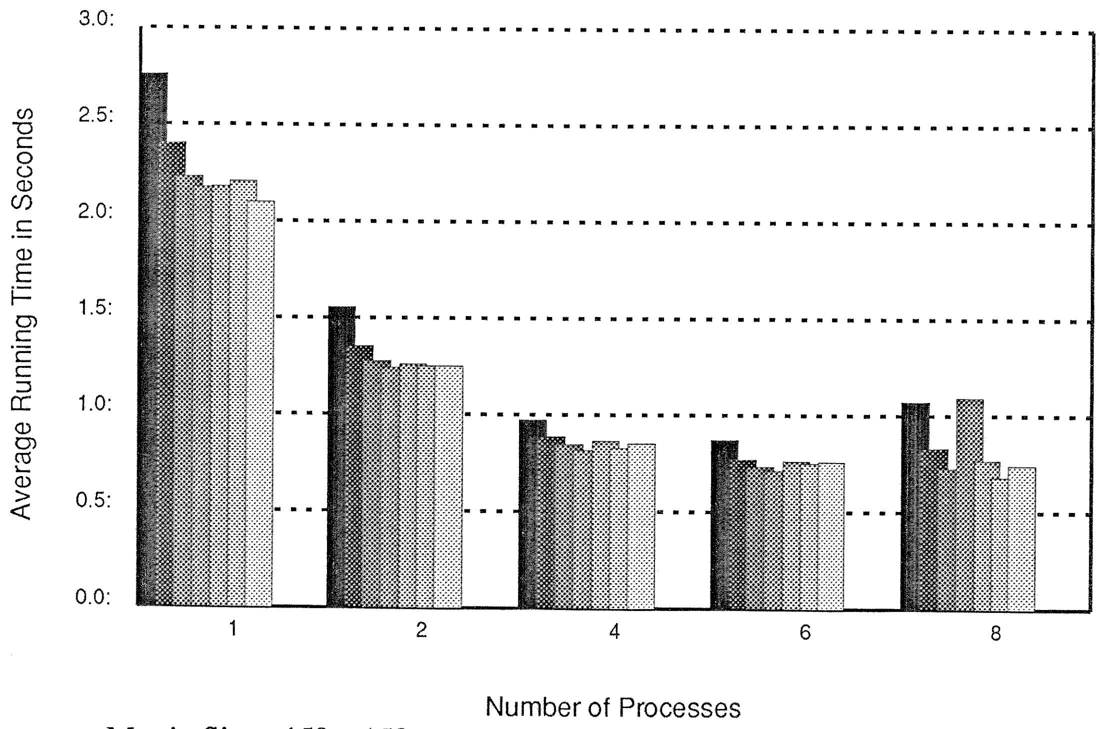


7.2.1.1. Matrix Size: 100 x 100

KEY: ▨ 1 ▨ 2 ▨ 4 ▨ 6

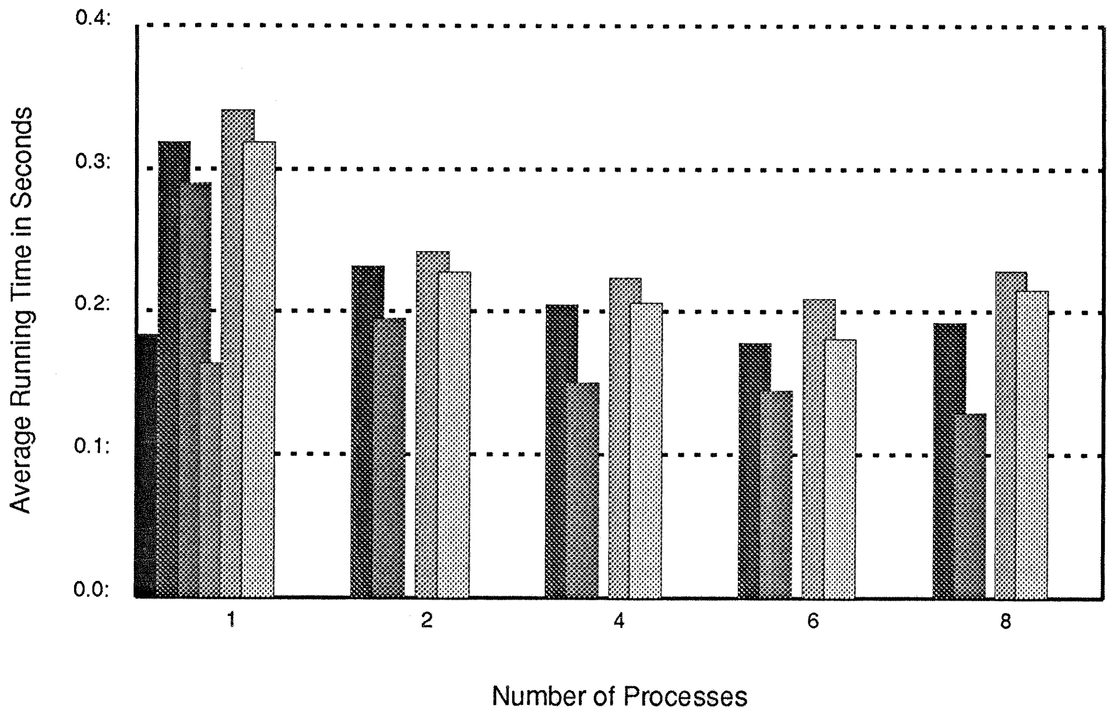
▨ 8 ▨ 10 ▨ 15

Optimal Size of Submatrix for PGESL (cont.)



7.2.1.2. Matrix Size: 150 x 150

7.2.2. DGESL vs. PGESL, With No Pivoting

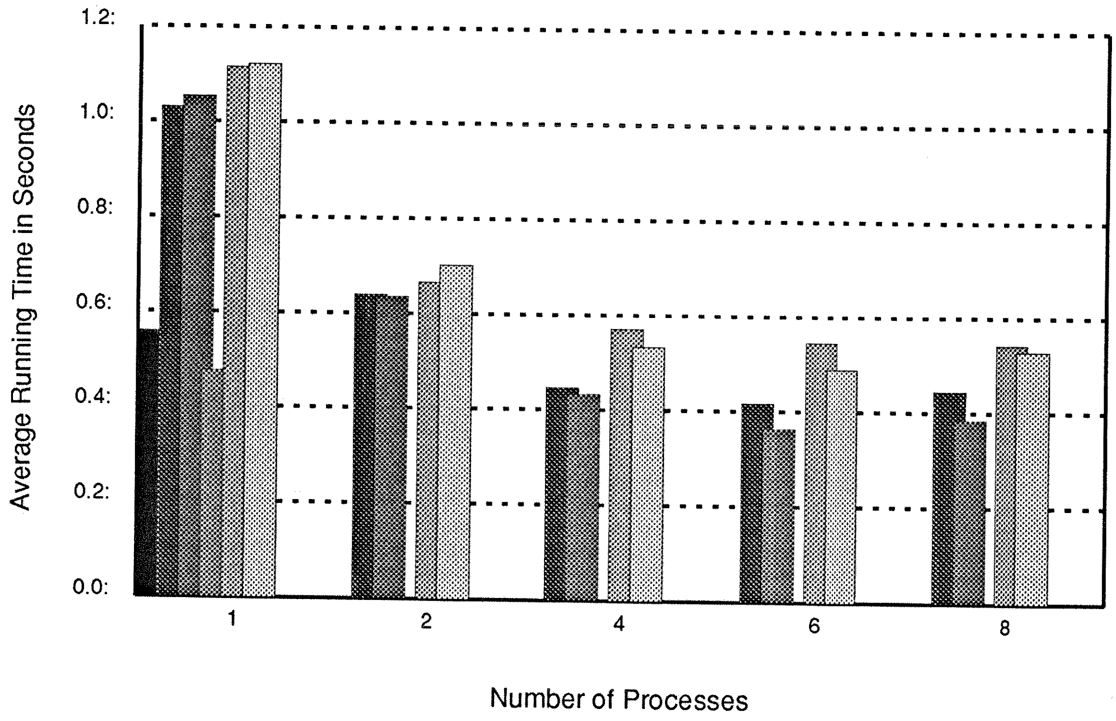


7.2.2.1. Matrix Size: 50 x 50

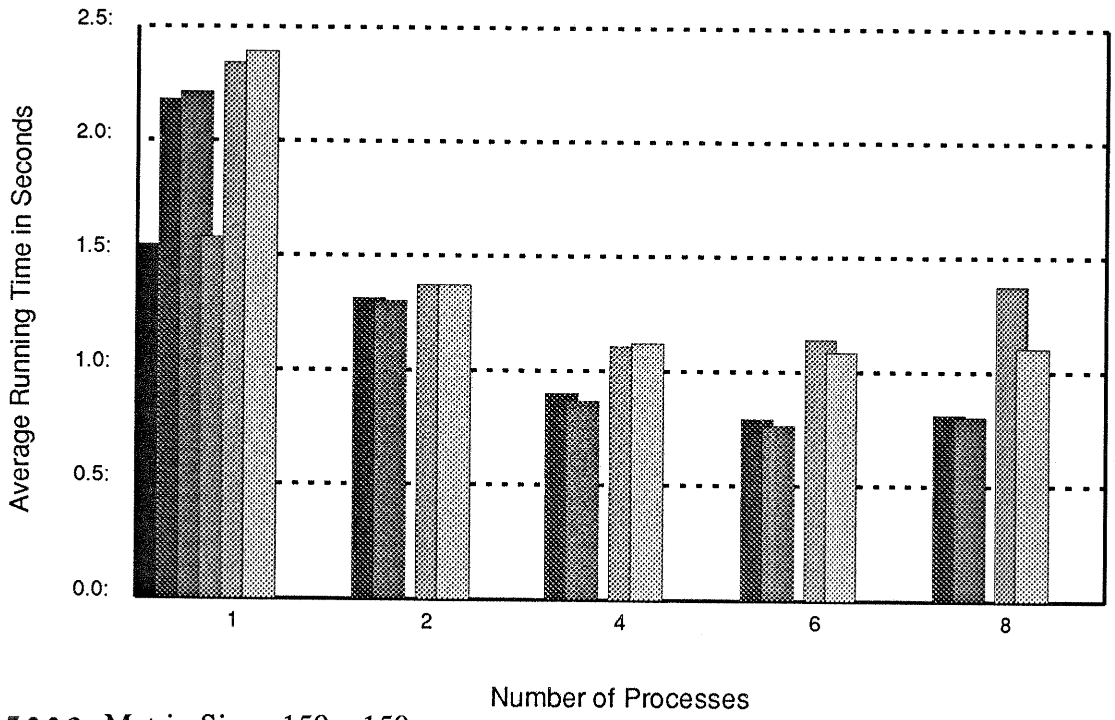
KEY:

■ DGESL(Ax=y)	▩ Version1(Ax=y)	▩ Version2(Ax=y)
▨ DGESL(ATx=y)	▨ Version1(ATx=y)	░ Version2(ATx=y)

DGESL vs. PGESL, With No Pivoting (cont.)

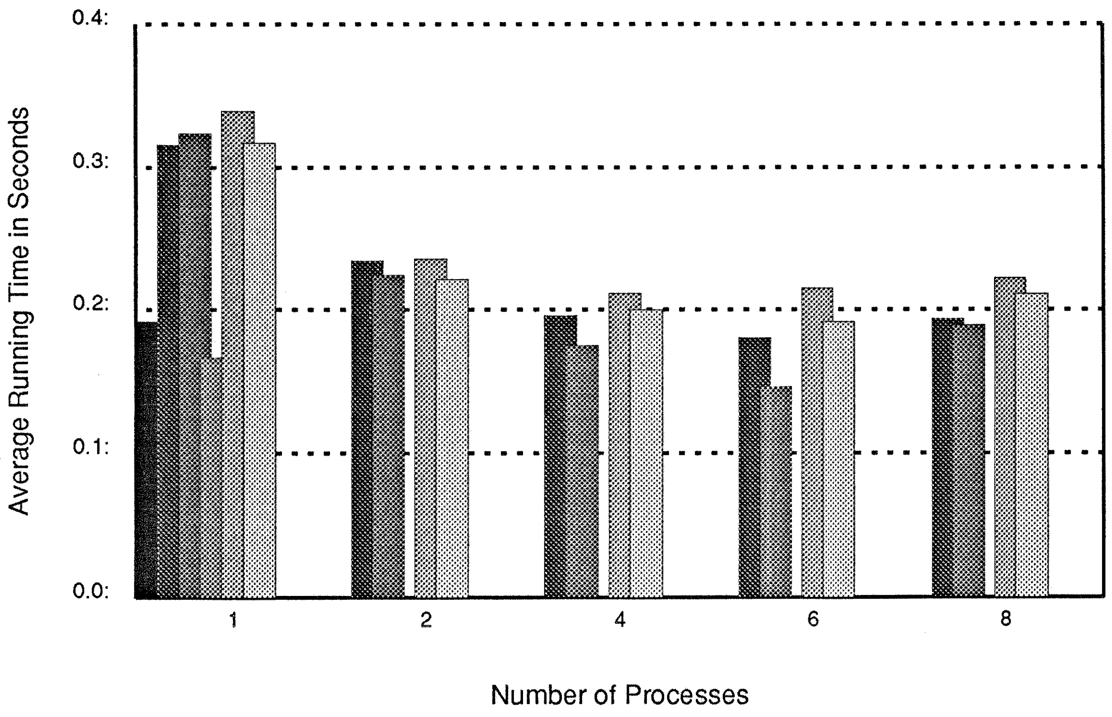


7.2.2.2. Matrix Size: 100 x 100



7.2.2.3. Matrix Size: 150 x 150

7.2.3. DGESL vs. PGESL, With Pivoting on Every Element

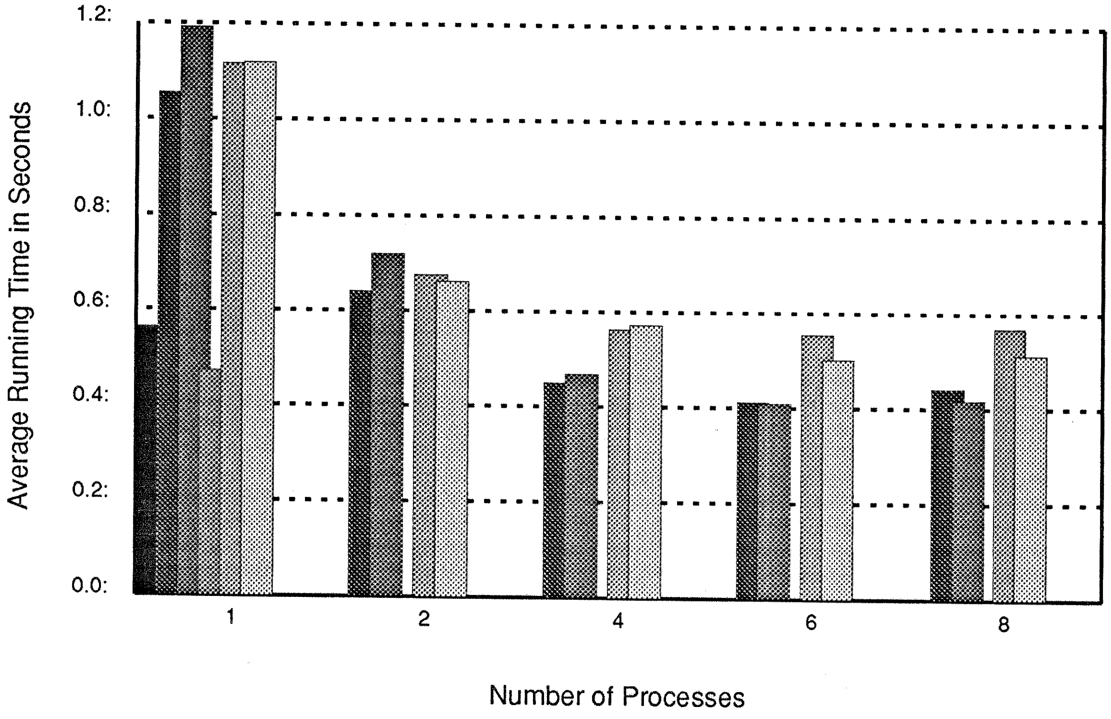


7.2.3.1. Matrix Size: 50 x 50

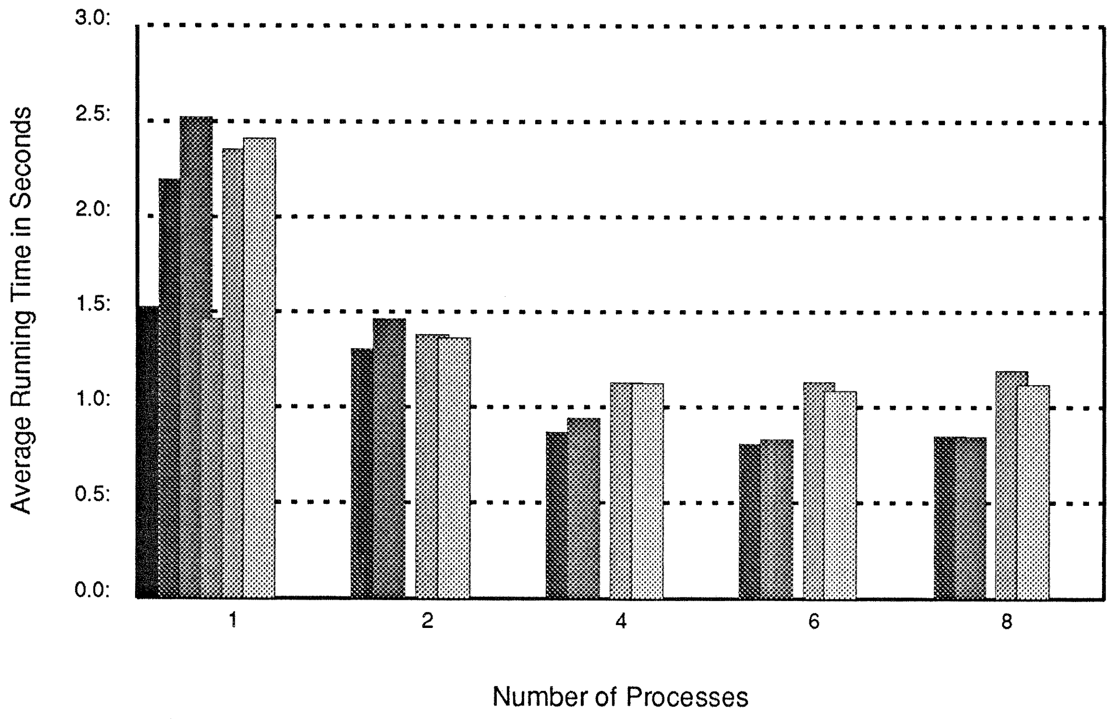
KEY:

■	DGESL(Ax=y)	▩	Version1(Ax=y)	▩	Version2(Ax=y)
▨	DGESL(ATx=y)	▨	Version1(ATx=y)	░	Version2(ATx=y)

DGESL vs. PGESL, With Pivoting on Every Element (cont.)



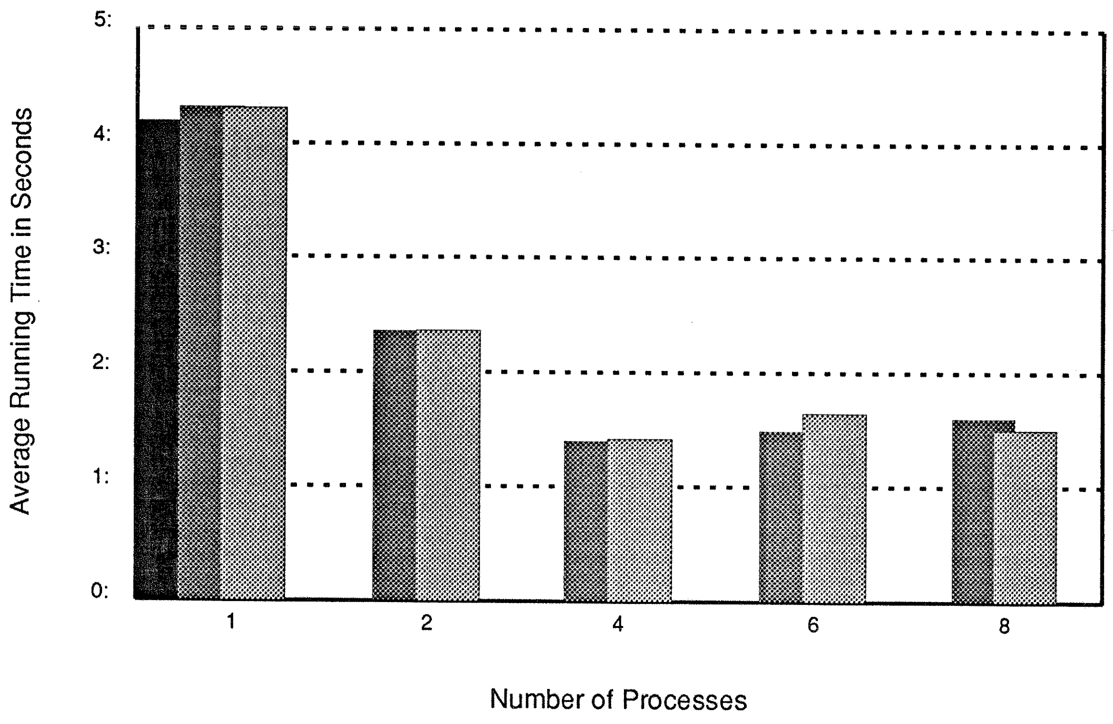
7.2.3.2. Matrix Size: 100 x 100



7.2.3.3. Matrix Size: 150 x 150

7.3. PQRDC

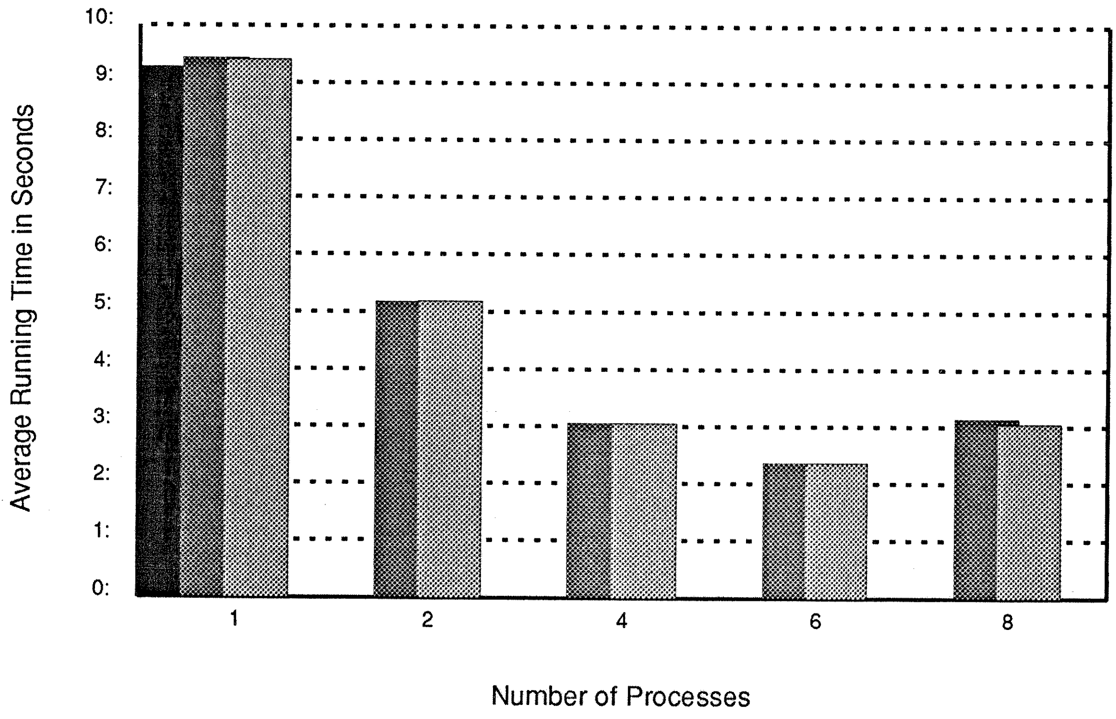
7.3.1. DQRDC Compared against Algorithm with One and Two Barriers



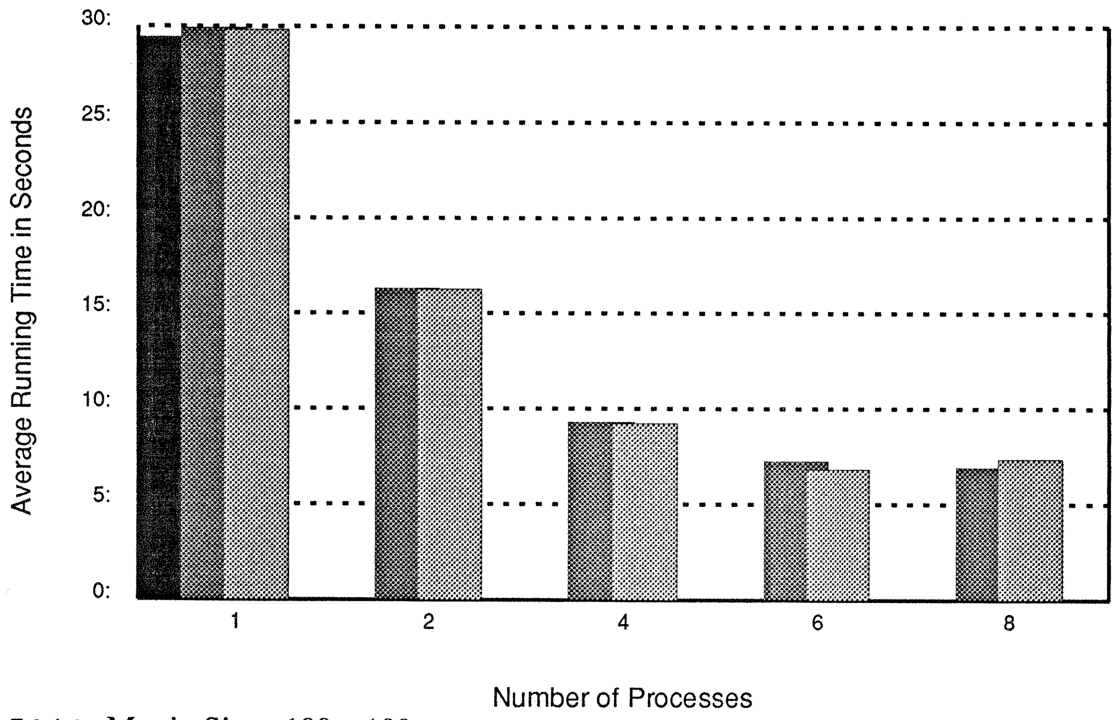
7.3.1.1. Matrix Size: 50 x 50

KEY: ■ DQRDC ▩ One Barrier ▨ Two Barriers

DQRDC Compared against Algorithm with One and Two Barriers (cont.)

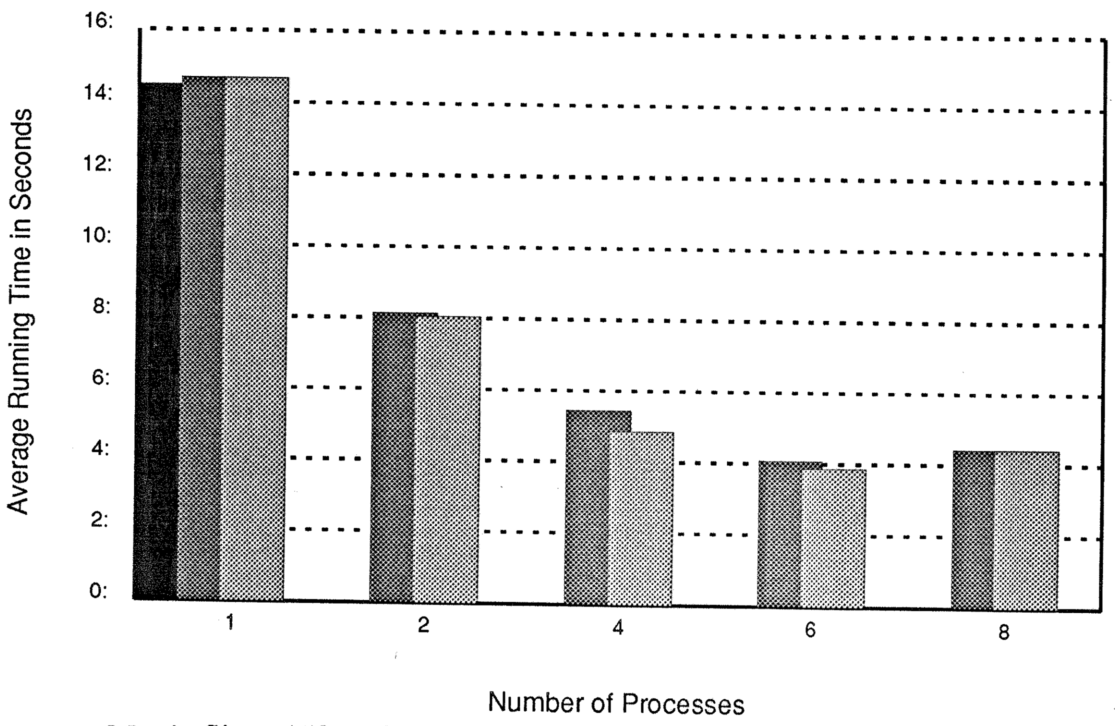


7.3.1.2. Matrix Size: 100 x 50

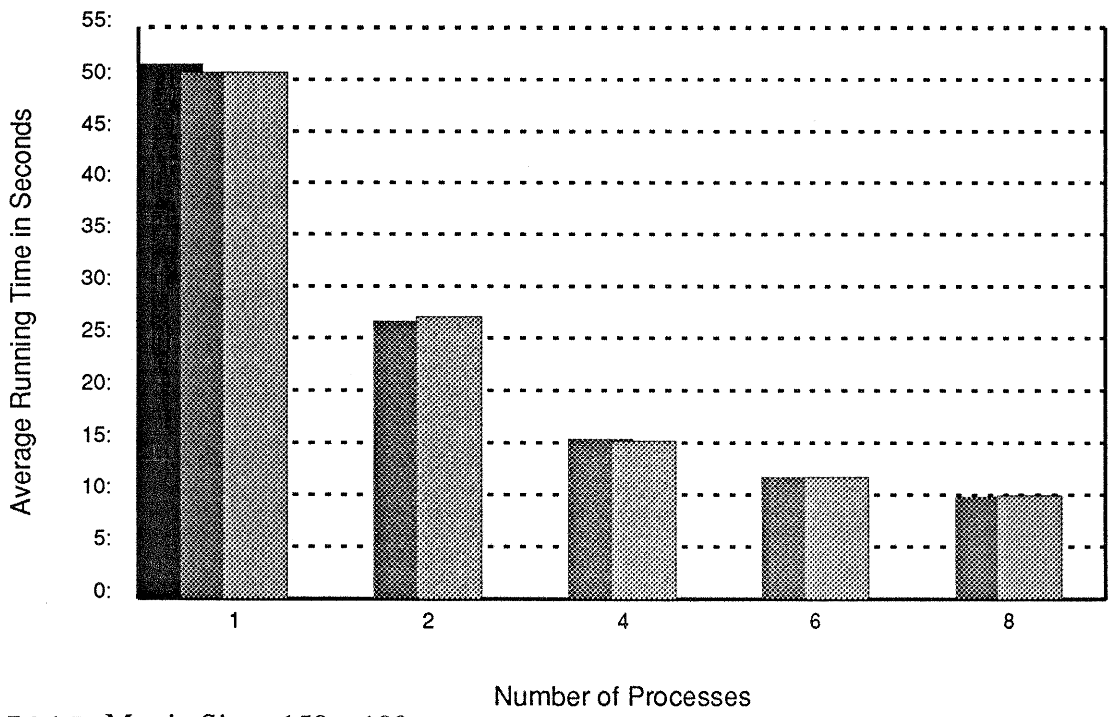


7.3.1.3. Matrix Size: 100 x 100

DQRDC Compared against Algorithm with One and Two Barriers (cont.)

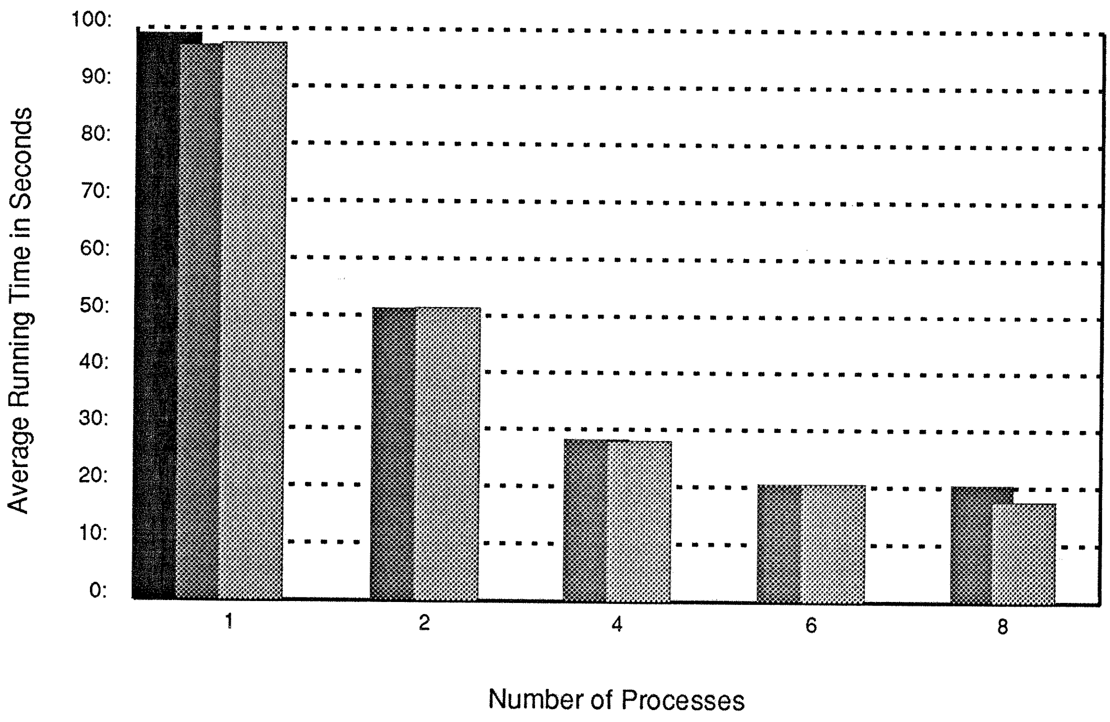


7.3.1.4. Matrix Size: 150 x 50

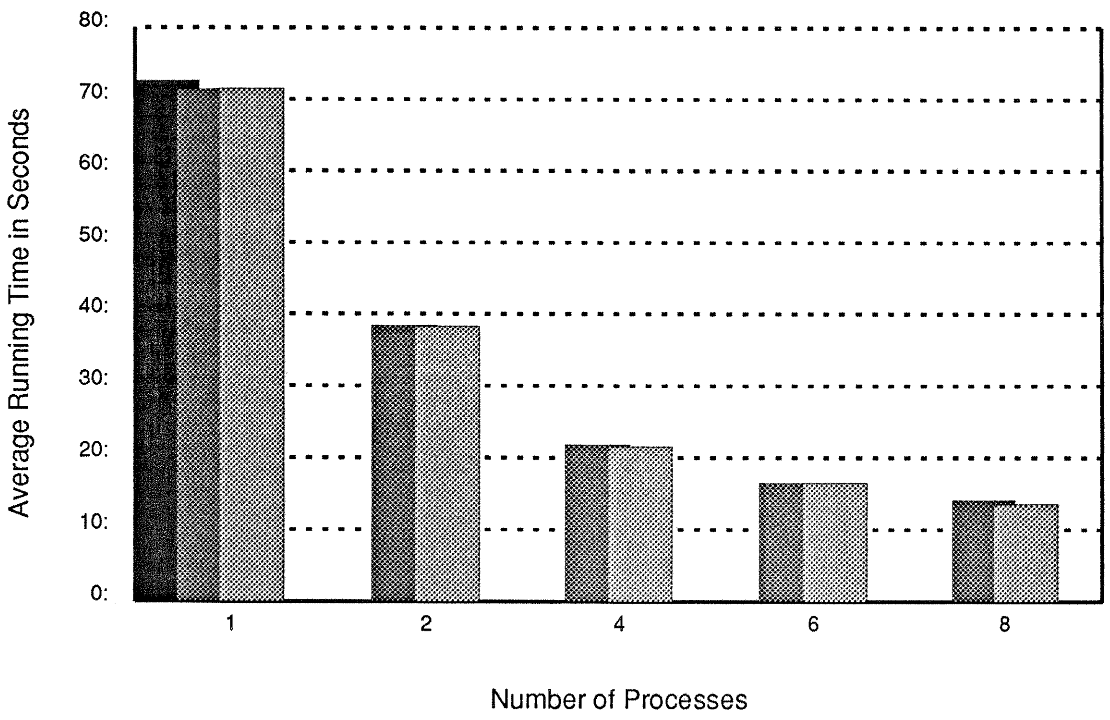


7.3.1.5. Matrix Size: 150 x 100

DQRDC Compared against Algorithm with One and Two Barriers (cont.)

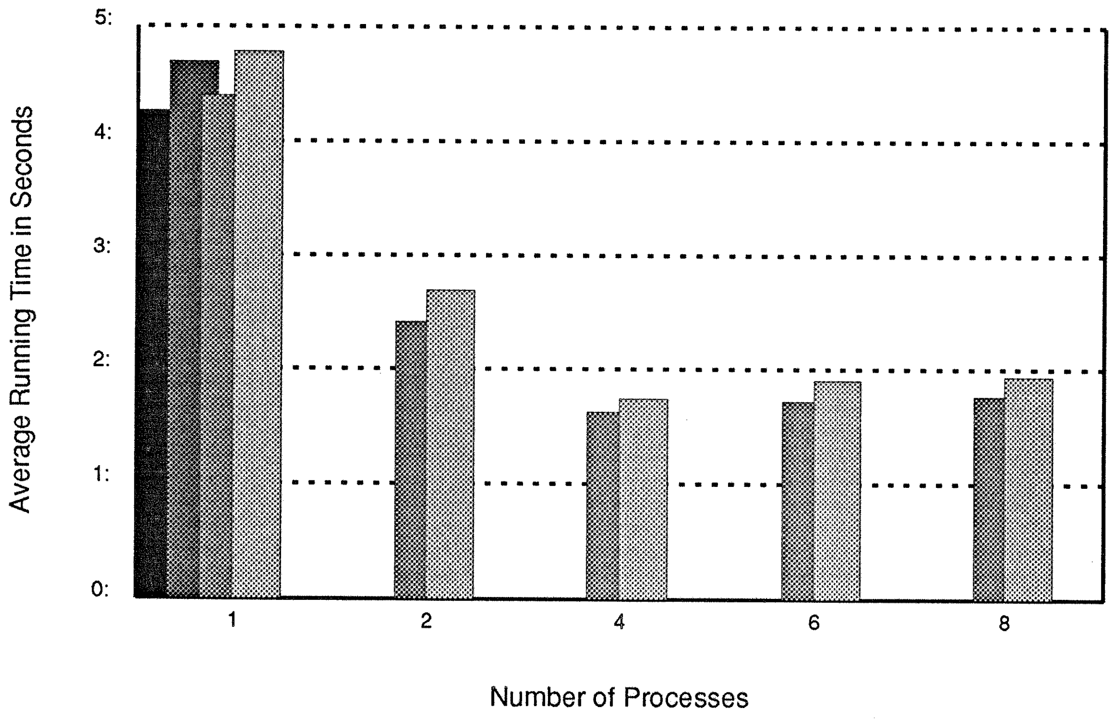


7.3.1.6. Matrix Size: 150 x 150



7.3.1.7. Matrix Size: 200 x 100

7.3.2. DQRDC vs. PQRDC, With and Without Pivoting

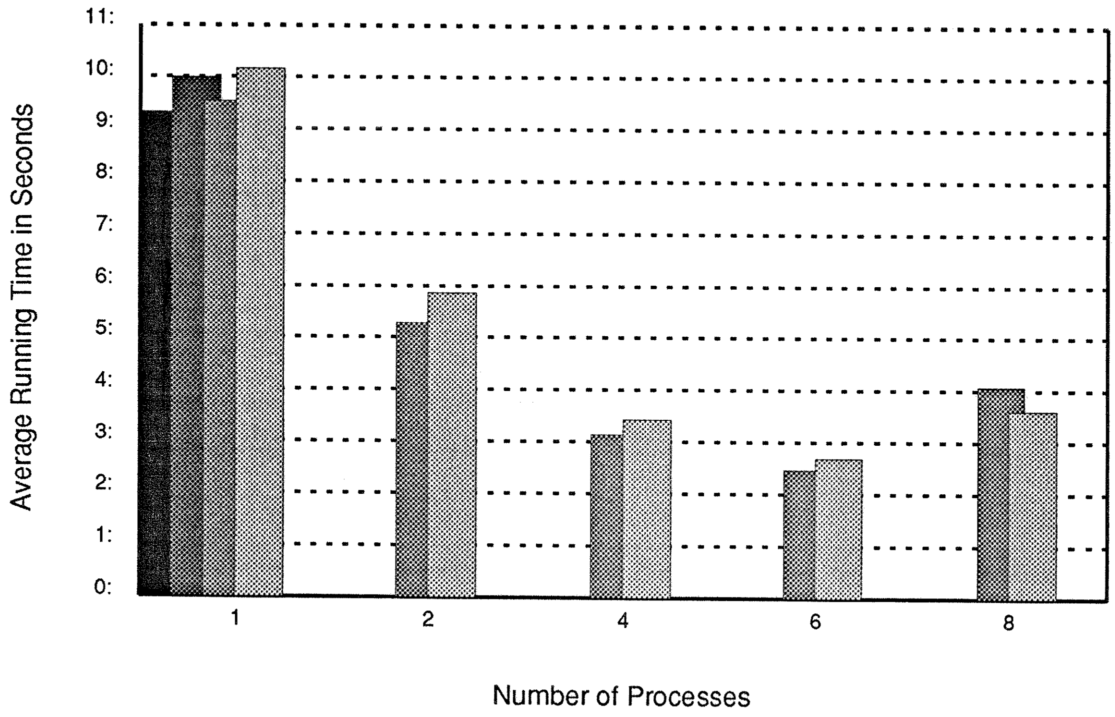


7.3.2.1. Matrix Size: 50 x 50

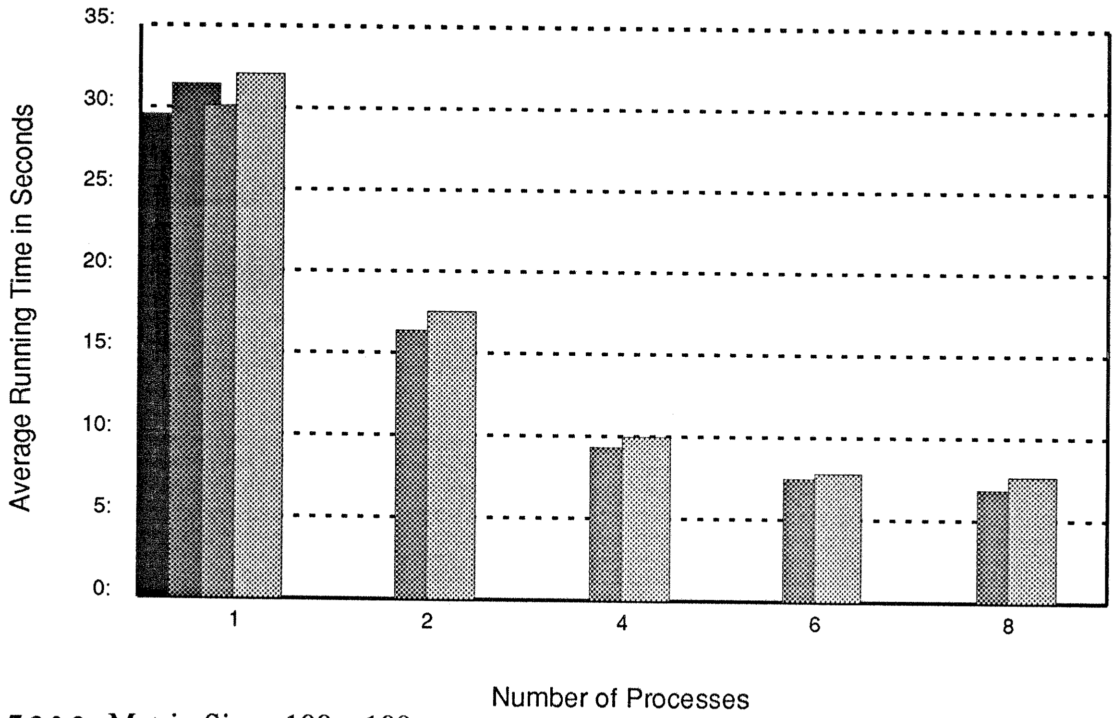
KEY:

■ DQRDC NoPiv	▩ DQRDC Pivot
▨ PQRDC NoPiv	░ PQRDC Pivot

DQRDC vs. PQRDC, With and Without Pivoting (cont.)

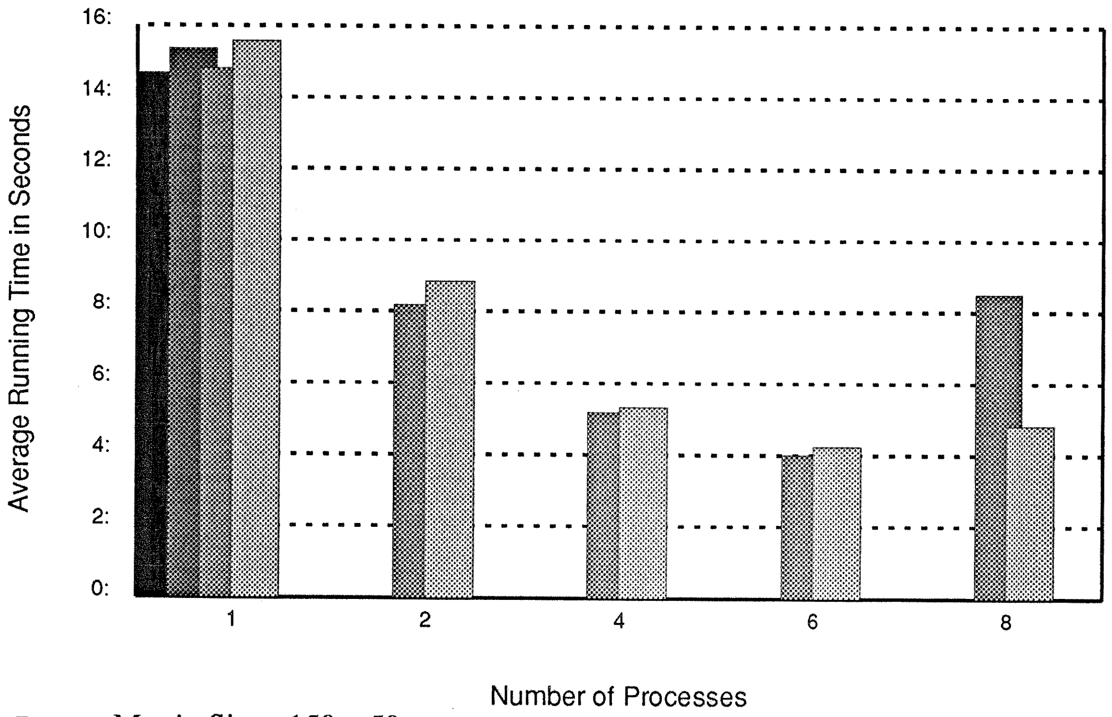


7.3.2.2. Matrix Size: 100 x 50

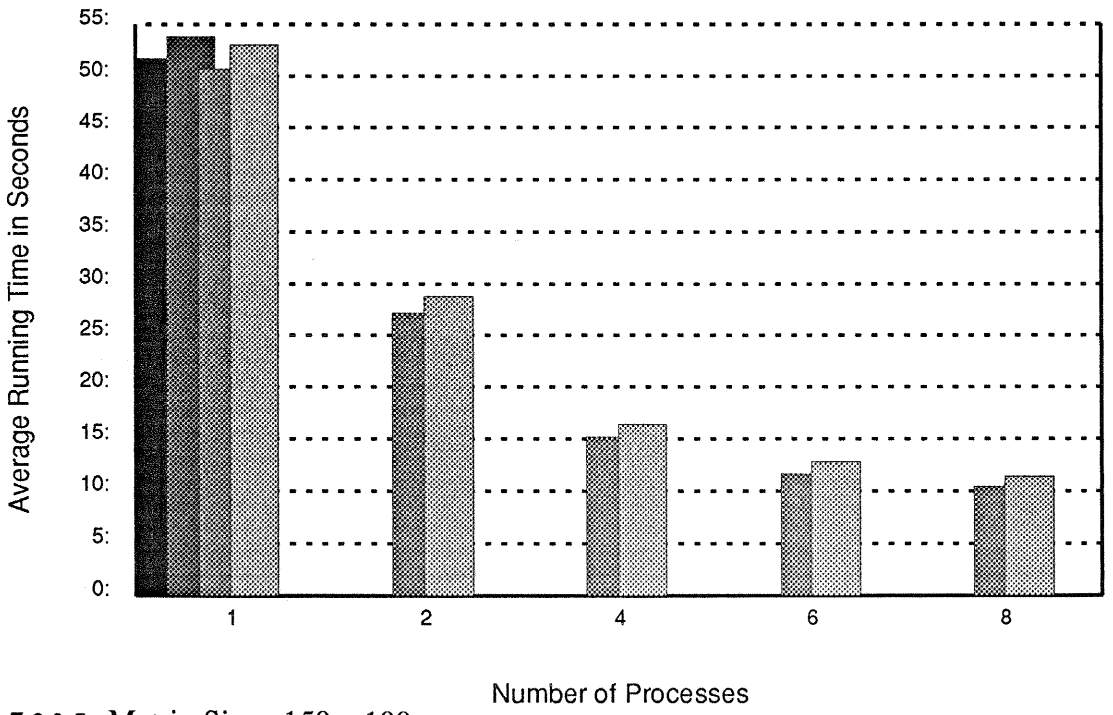


7.3.2.3. Matrix Size: 100 x 100

DQRDC vs. PQRDC, With and Without Pivoting (cont.)

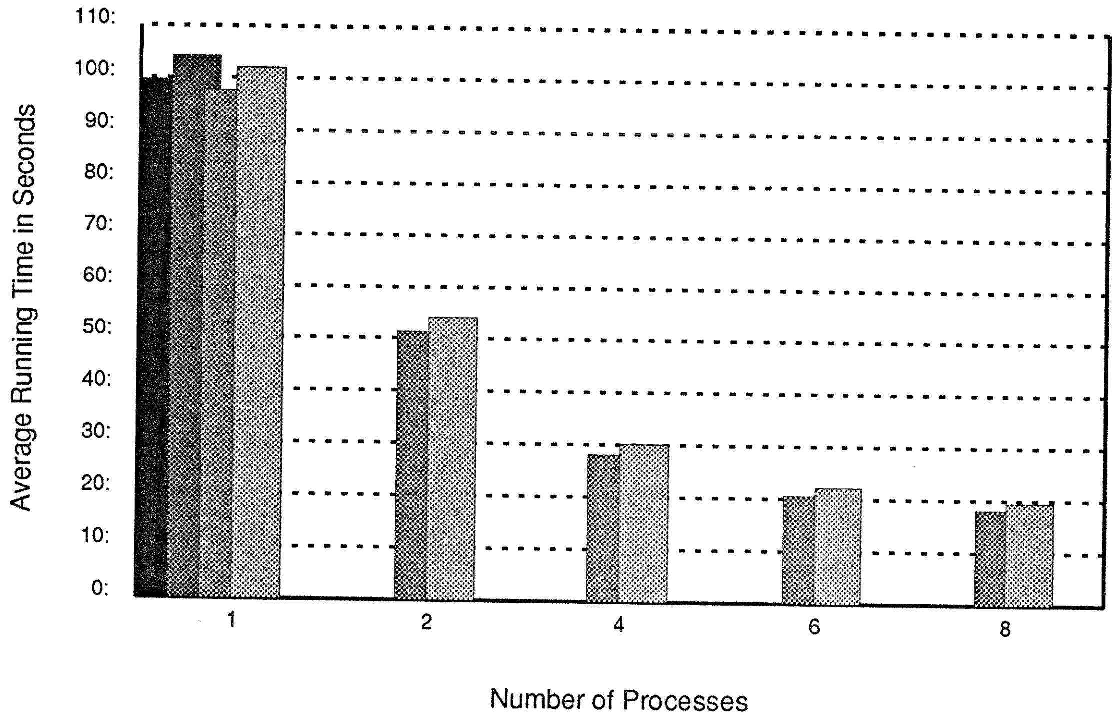


7.3.2.4. Matrix Size: 150 x 50

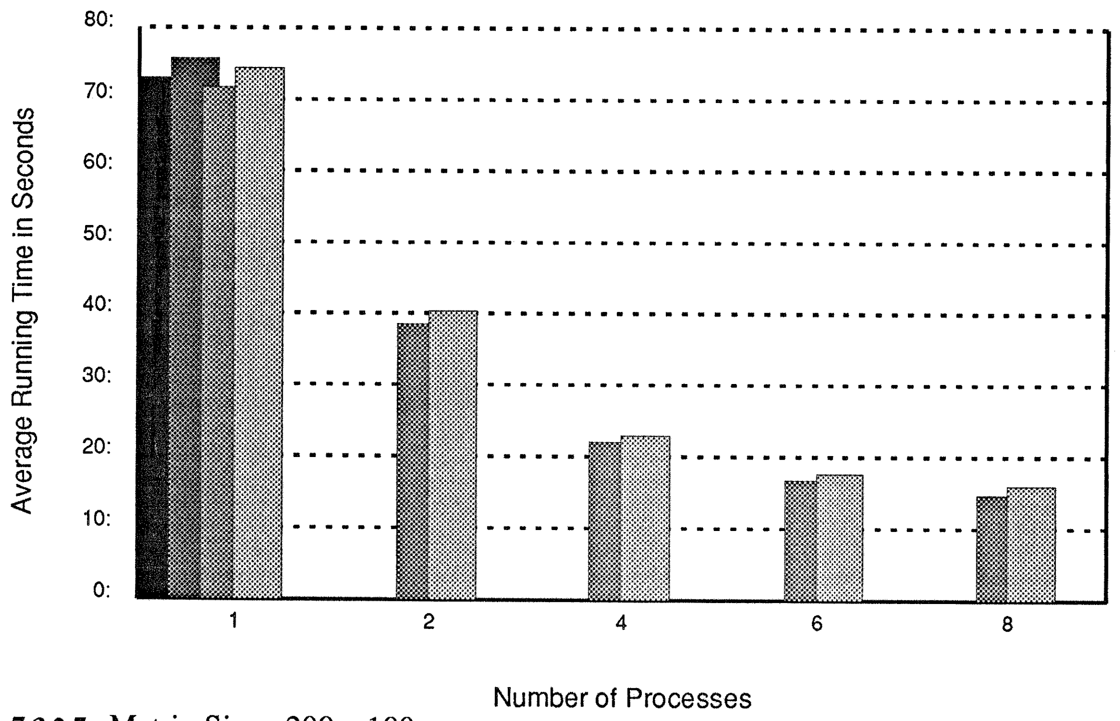


7.3.2.5. Matrix Size: 150 x 100

DQRDC vs. PQRDC, With and Without Pivoting (cont.)



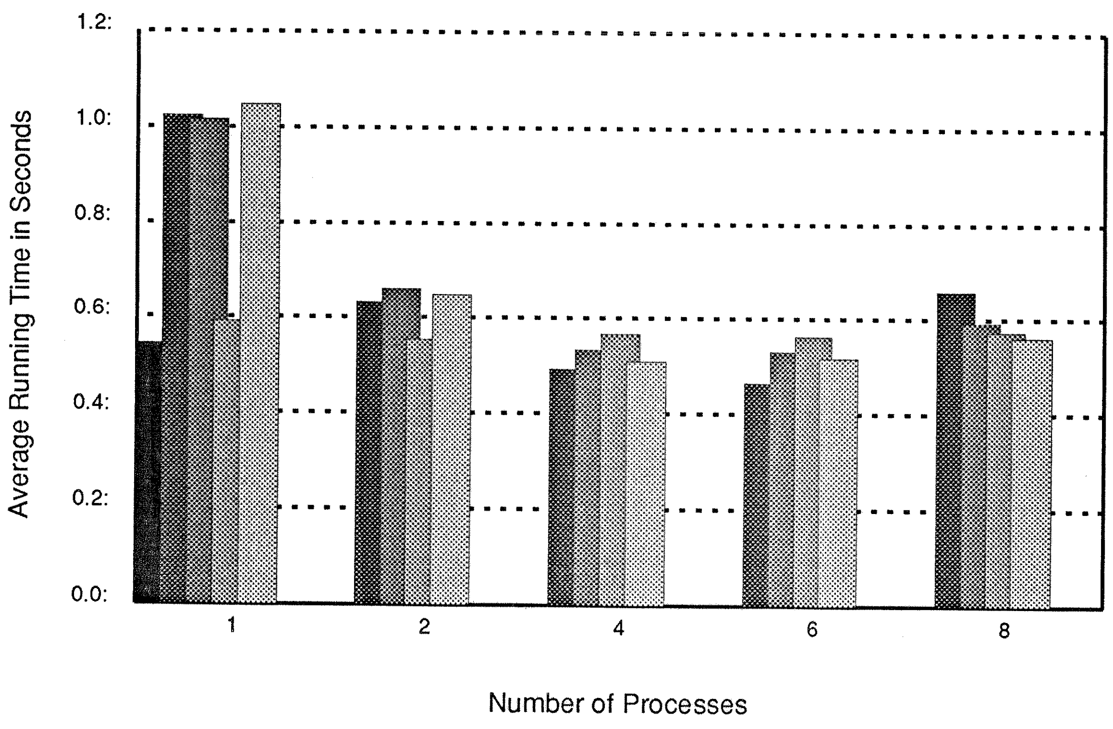
7.3.2.6. Matrix Size: 150 x 150



7.3.2.7. Matrix Size: 200 x 100

7.4. PQRSL

7.4.1. DQRSL and PQRSL Average Running Times under All Experiments

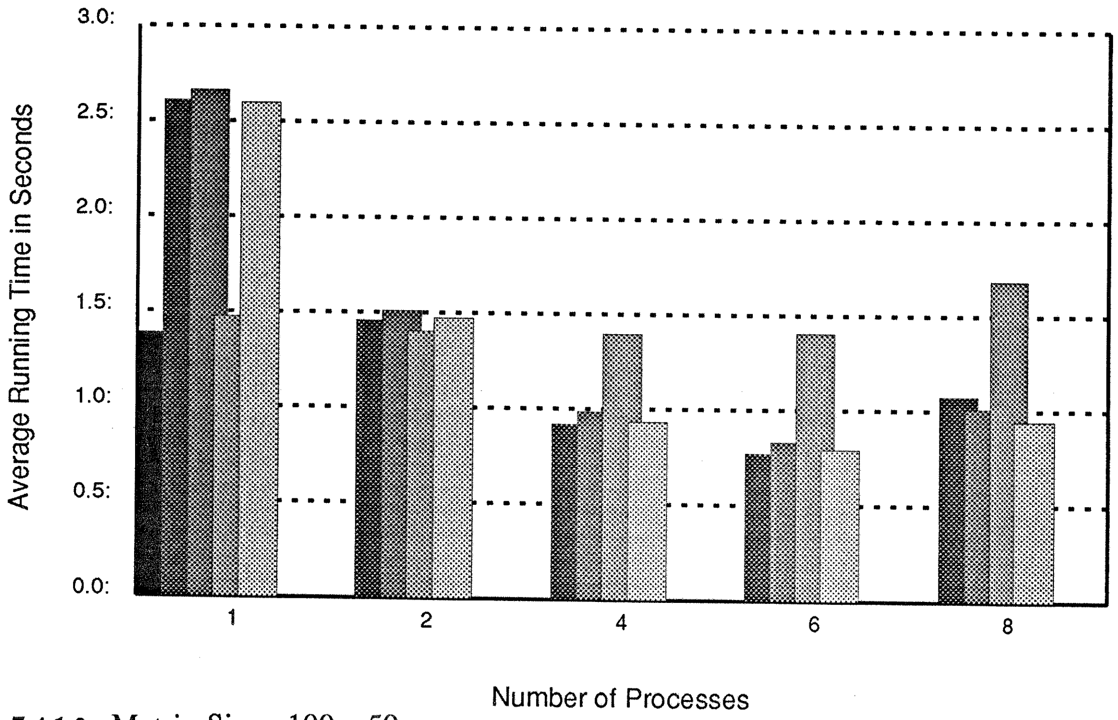


7.4.1.1. Matrix Size: 50 x 50

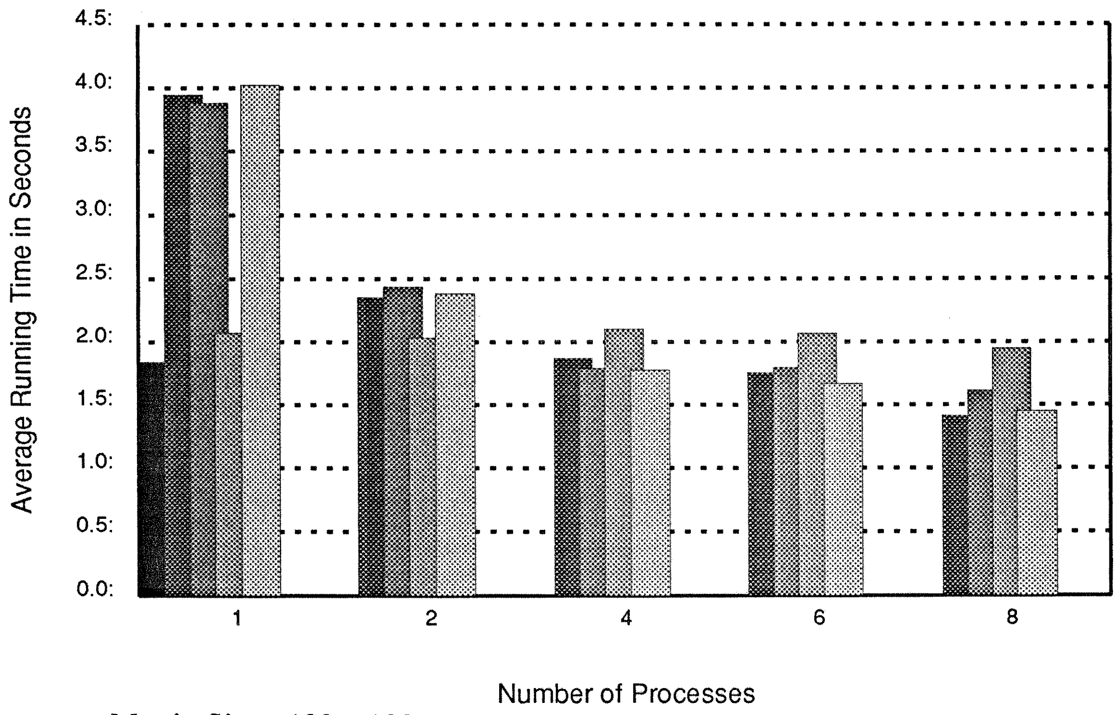
KEY: ▇ DQRSL ▓ PQRSL

▒ Nowavefront ▒ Noparcomp ░ Noparinit

DQRSL and PQRSL Average Running Times under All Experiments (cont.)

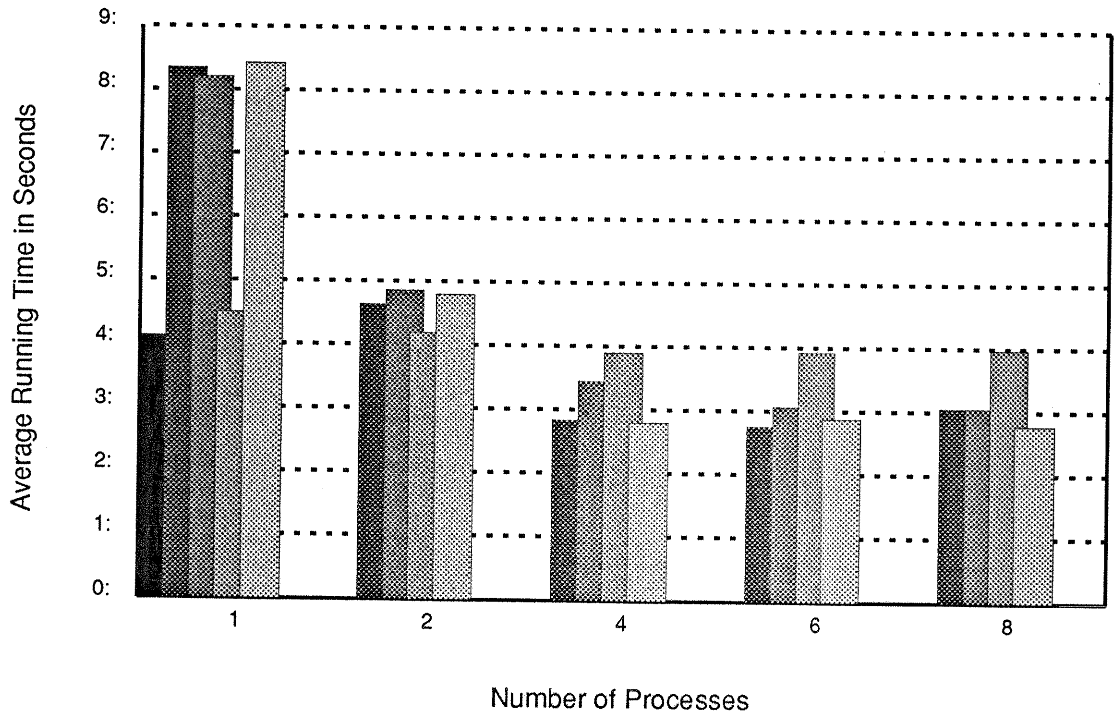


7.4.1.2. Matrix Size: 100 x 50

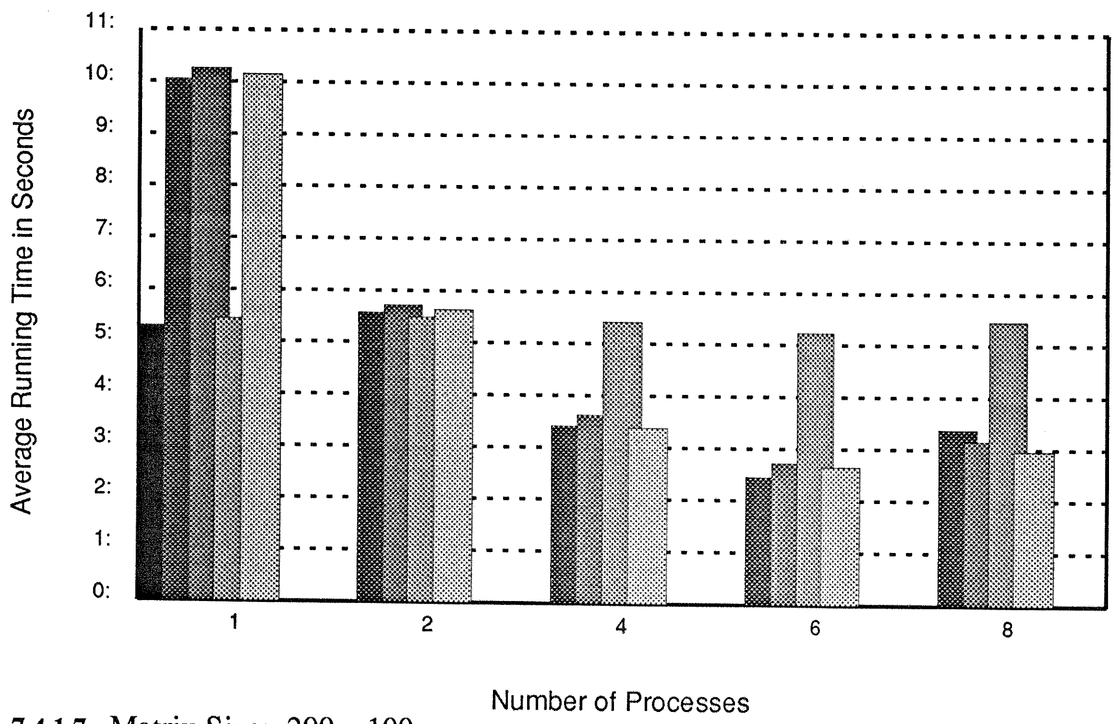


7.4.1.3. Matrix Size: 100 x 100

DQRSL and PQRSL Average Running Times under All Experiments (cont.)

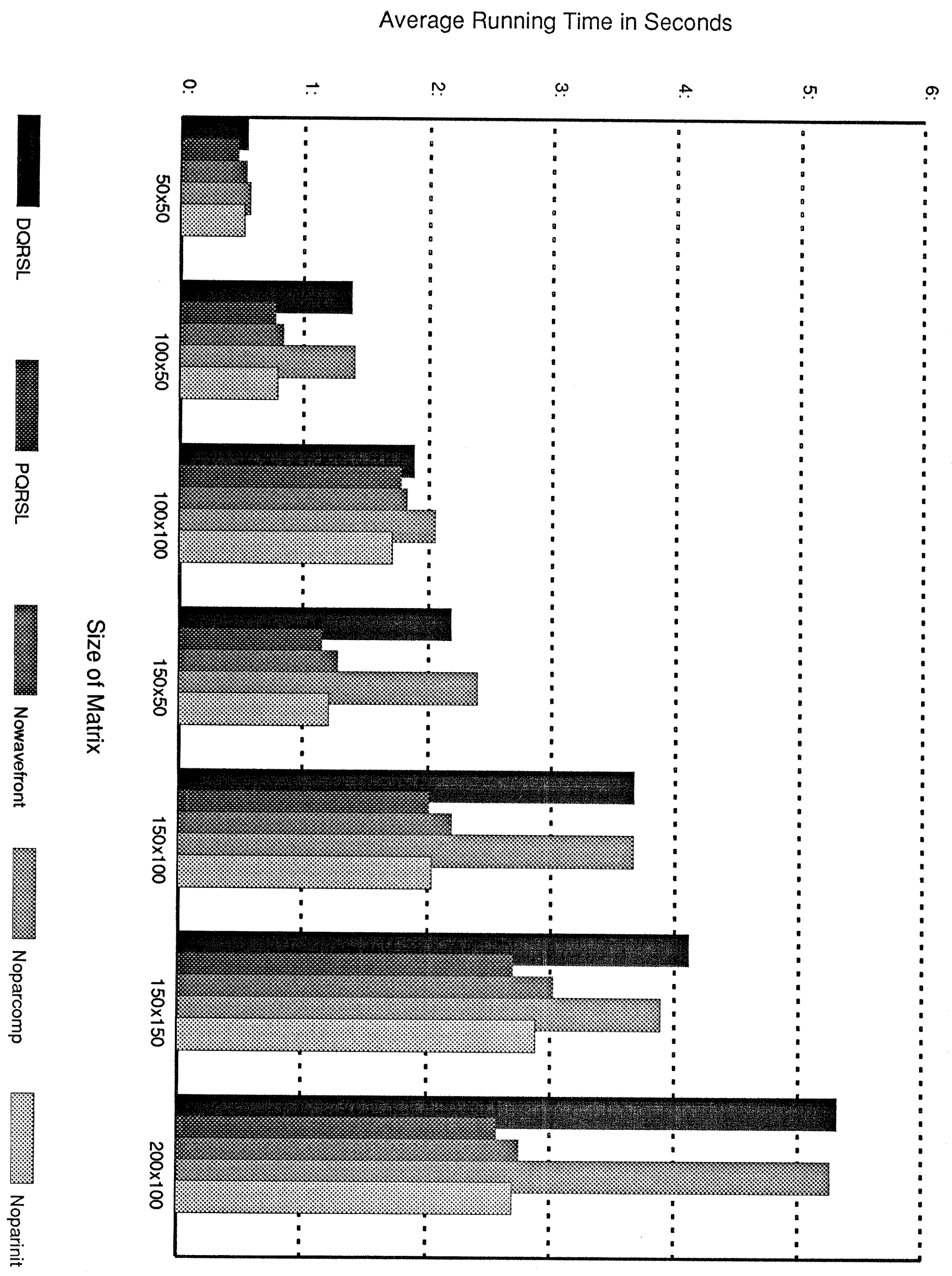


7.4.1.6. Matrix Size: 150 x 150



7.4.1.7. Matrix Size: 200 x 100

7.4.2. Gain Shown by PQRSL Versions for All Matrix Sizes


Average Running Time in Seconds
0:
1:
2:
3:
4:
5:
6:
50x50
100x50
100x100
150x50
150x100
150x150
200x100
Size of Matrix
DQRSL
PQRSL
Nowavefront
Noparcomp
Noparinit