Performance Comparison of Three Parallel Implementations of a Schwarz Splitting Algorithm

Jim Gamble and Calvin J. Ribbens

TR 89-37

Performance Comparison of Three Parallel Implementations of a Schwarz Splitting Algorithm

Jim Gamble*
Calvin J. Ribbens*
Virginia Polytechnic Institute & State University

October 23, 1989

Abstract

We describe three implementations of a Schwarz splitting algorithm for the numerical solution of two-dimensional, second-order, linear elliptic partial differential equations. One implementation makes use of the SCHEDULE package. A second uses the language extensions available in SEQUENT Fortran for creating and controlling parallel processes. The third implementation is a hybrid of the first two—using explicit (non-portable) calls to create and control parallel processes, but using data structures and overhead similar to the implementation that uses SCHEDULE. We report results from several experiments with a typical test problems on a Sequent Symmetry S81 shared-memory multiprocessor. We measure the overhead involved in using SCHEDULE, and discuss advantages and disadvantages of this approach.

1 Introduction

Determining which of various parallel computational methods will be the most efficient for solving a specific problem on a given computer is rarely possible a priori. The experiment described herein was an attempt to discover factors which might influence the performance of a given method on a Sequent Symmetry S81 shared-memory parallel computer. The parallel methods differed in how each coordinated its sub-processes. One approach (represented in two implementations) used system calls and programmed logic. The other method used the SCHEDULE software package. The factors of interest included overhead of process coordination, and contention on the shared-memory bus.

Testing of these factors involved three separate programs which all solved second-order linear elliptic PDE's over square two-dimensional domains with varying grid sizes and different numbers of overlapping Schwarz subdomains. The Schwarz algorithm is described in Section 2. One program (which will be referenced as Program X—for eXplicit) had in its code the logic to allocate processors and coordinate the work on different subdomains through explicit calls to language extension software. This program used local memory wherever possible, on the assumption that use of local memory minimizes conflicts on the shared-memory bus.

A second program used the SCHEDULE software package, developed at Argonne National Labs (we will refer to this program as Program S). SCHEDULE software gives the programmer access

*Supported by DOE grant DE-FG05-88ER25068.

to the parallel features of a machine through subroutine calls. Different SCHEDULE subroutines allow the user to specify how many processors are to execute a job in parallel, which subroutine(s) and what calling sequences will execute, and in what sequence these subroutines should execute. The design objective of SCHEDULE is to allow programmers to write programs which will execute in parallel, and be portable to any machine which has a SCHEDULE library. SCHEDULE is currently available for Alliant, Encore, Sequent, Vax, Flex, Sun, Cray2, and NCUBE machines. Standard references for SCHEDULE are Dongarra and Sorenson [1,2,3].

In general, processes executed by SCHEDULE are scheduled upon entry into the SCHEDULE subroutine. An extra SCHEDULE provision allows any process already executing under SCHEDULE to initiate one or more processes dynamically. If scheduling is totally determined upon entry into SCHEDULE, the program is said to be running "static" SCHEDULE. In the case that one or more processes spawns children dynamically, the program is said to be running "dynamic" SCHEDULE. In all cases, there is some process whose execution is determined statically—that is, upon entry into SCHEDULE. The terms "static" and "dynamic" may also be applied to individual processes, and they indicate whether the process in question was pre-scheduled (static) or spawned (dynamic). Dynamic SCHEDULE processes have additional mechanisms available for synchronization of and re-entry into spawned processes. These mechanisms are not available to static processes.

Program S used a smaller granularity for the tasks which it ran in parallel as compared to Program X. The SCHEDULE software itself forced this decision because of limitations in the SCHEDULE software package. Four features which we found particularly frustrating were: (1) a lack of any means by which static processes might be synchronized during execution; (2) a limit on the total number of jobs which may be controlled by a single invocation of SCHEDULE (1000 jobs); (3) a limit on the number of times spawned processes may be re-entered (the actual number depends upon the entire SCHEDULE load); and (4) a deficient job sequencing language which allows neither conditional execution nor iterative execution.

Our implementation required that all parallel processes be synchronized at several points in their execution. The synchronization mechanism available to dynamic processes would be adequate for our needs, but unfortunately, point (3) above prevented us from using dynamic processes. Subsequently, point (1) above forced us to break the parallel processes into four smaller processes, each of which would be synchronized through separate calls to SCHEDULE from the main program. This implementation also spared us problems with the job sequencing language mentioned in point (4).

The drawback was that now Program S was forced to initiate two invocations of the SCHEDULE subroutine for each iteration of the numerical method (see description below). As the two invocations are temporally and logically disjoint, data that were needed by both invocations were forced to be stored in shared memory, so that they would not be lost upon termination of either SCHEDULE call. Because of this unfortunate implementation of SCHEDULE, we anticipated a decrease in performance. The reasons for the decrease would be (1) the added overhead of spawning child processes twice per iteration of Program S (compared to once per run of Program X); and (2) contention on the shared-memory bus for data which were now required to be stored in shared memory.

Finally, we wrote a hybrid program in an attempt to isolate the effects of memory contention on the shared-memory bus. Ideally, the hybrid program (which we will call Program H) was to use the logic of Program X, and the data structures of Program S. The actual Program H had some data structures unique unto itself; however, these structures were used infrequently compared to

the computational data structures, and we ignored their effect on program performance.

2 Procedure

The numerical method for this experiment was Schwarz splitting. Schwarz splitting is an approach to solving two-dimensional partial differential equations (PDEs) that has attracted considerable interest in recent years, primarily because of the ease with which it can be parallelized. The basic idea of Schwarz splitting is to split the domain into overlapping subdomains, and to repeatedly solve the PDE on each of the subdomains independently (thus the potential for parallelism). Boundary conditions for subdomain boundaries which correspond to portions of the original boundary are given by the original boundary conditions; boundary conditions for subdomain boundaries which lie inside a neighboring subdomain are typically derived from the approximate solution at a previous iteration. For a more extensive treatment of Schwarz splitting, see Tang [9] and Lions [5,6].

There are many variations to the basic Schwarz splitting scheme. The performance of the method is significantly affected by the degree of overlap between subdomains, the choice of stopping criteria for the Schwarz iteration, the method used to solve the individual subdomain problems, and the means by which new boundary conditions are derived. The main purpose of this study is not to experiment with all possible variations of Schwarz splitting. We are simply interested in methods of parallelizing algorithms such as Schwarz splitting for a shared memory architecture.

Consequently, we simply make reasonable choices for the alternatives just mentioned, and proceed with the experiment. In particular, the overlap ratio between subdomains is set at 0.25. This means, for example, that in the horizontal direction, one-fourth of a subdomain is overlapped on both the left and right. Each subdomain must be a rectangle, and we split in both the horizontal and vertical direction. We halt the Schwarz iteration when the maximum relative change in the solution is less than $(errest/100)$, where *errest* is an estimate of the discretization error. The individual subdomain solves are done using ELLPACK modules 5-POINT STAR and LINPACK SPD BAND (see Rice and Boisvert [7] for a description of ELLPACK). These software modules implement standard second-order finite differences and band Cholesky factorization, respectively. Since only the right-hand sides of the discrete systems change from Schwarz iteration to iteration, we only form and factor the linear systems once. At each Schwarz iteration we then simply do a forward- and back-solve to compute the new approximate solution vector for each subdomain. Finally, we refer to the iteration scheme used here as a "Jacobi" iteration because we use the neighboring solution at the previous iteration as boundary condition for subdomains interior to the original region. There are more complicated schemes (SOR, conjugate gradient, etc.), but for our purposes it is sufficient to consider the simplest approach.

All of the data reported below is based on numerically solving a single linear, self-adjoint elliptic problem (Problem 4 from Rice, et. al [8]).

The pseudo-code for Program X and Program S are given in Figures 1 and 2. The logic of Program H is identical to that of Program X, except for some inconsequential details. Note also that child processes in Program X are responsible for *nsubdomains/nprocs* pieces while child processes in Program S are responsible for only one piece (but may be called more often). Each processor works on one or more Schwarz subdomains at each iteration. Note that for the data reported below, each processor is responsible for the same number of Schwarz subdomains; that is, the work load across all processors is even. The method described above requires that the program coordinate the work between all processors at critical points: after each matrix solve, and after each answer store. Thus, each program has some logic which prevents any processor from advancing to

Parent Process

```
parameter nprocs
begin
  define subdomains and grids
  get_time(start)
  spawn nprocs child processes
  get_time(end)
  compute error
  print results
end
```

Child Process

```
parameter npieces = (nsubdomains / nprocs)
begin
  for each of my npieces do
    generate initial (zero) solution
  for each of my npieces do begin
    discretize (5-point star)
    factor matrix
    save factorization
    end
  while (not converged) do begin
    for each of my npieces do begin
      compute rhs
      back solve
      end
    for each of my npieces do
      copy solution
    if (my_id = 1) test convergence
    end
end
```

Figure 1: Psuedo-code for Program X.

Parent Process
```
parameter nsubdomains
begin
  define subdomains and grids
  get_time(start)
  SCHEDULE nsubdomains Inits
  SCHEDULE nsubdomains Factors
  while (not converged) do begin
    SCHEDULE nsubdomains Iters
    SCHEDULE nsubdomains Copys
    test convergence
    end
  get_time(end)
  compute error
  print results
end
```

Child Init
```
begin
  generate initial (zero) solution
end
```

Child Factor
```
begin
  discretize (5-point star)
  factor matrix
  save factorization
end
```

Child Iter
```
begin
  compute rhs
  back solve
end
```

Child Copy
```
begin
  copy solution
end
```

Figure 2: Psuedo-code for Program S.

Table 1: Processors used for a given configuration.

Grid Per Subdomain	Number of Subdomains		
	2×2	3×3	4×4
5x5	1,2,4	1,3,9	1,2,4,8
9x9	1,2,4	1,3,9	1,2,4,8
13x13	1,2,4	1,3,9	1,2,4,8
17x17	1,2,4	1,3,9	1,2,4,8
25x25	1,2,4	1,3,9	1,2,4,8
33x33	1,2,4	1,3,9	1,2,4,8
49x49	1,2,4	1,3,9	
65x65	1,2,4		

the next step until all processors have completed the current step. Failure to do this would allow a fast processor to corrupt the "current" answer with his "new" answer before slower processors have computed their "new" answers (which depend upon the integrity of the "current" answer).

We ran each program on a series of test cases, and measured the time needed to reach convergence. The problems were configured by the number of Schwarz subdomains, the grid size on each subdomain, and the number of processors used to compute the solution. Table 1 lists the problem configurations which we actually ran. The table indicates how many Schwarz subdomains are used in each direction (so that "2x2" means a total of four subdomains, for example). For each run, we completely recompiled the program so that smaller tasks would not be penalized by large data structures which they did not need. The reason for not running the finest grids on the 3x3, and 4x4 subdomain cases is that the data structures became too large to run on our Sequent. Another hardware limitation (only 10 processors) prevented us from running the 4x4 subdomains with 16 processors. The results averaged from five runs are given in Appendix A.

By inspection of program logic, we developed a model for the time required to complete each problem configuration. Our model is based on the assumption that the time needed for the factor and solve steps depends only on the size of the grid. A corollary to this assumption is that the time is independent of the problem data for the class of problems we considered. Both assumptions are convenient for our needs. Neither is completely realistic. We bypass the question by further assuming that for the precision of our clock, the differences are not significant. Actually, the time needed to factor and/or back-solve the subdomains will be affected to a certain degree by the location of the subdomain. Interior subdomains (the 2x2 case has no interior subdomains, the 3x3 case has one, and the 4x4 case has four) require less time because the boundary look-up simply reads a value from an array. The side and corner subdomains must evaluate a function to determine boundary values. Discounting these differences, we write, for all grids,

$$\begin{aligned} \text{Time} \quad = \quad & (\text{nsubdomains} + 1) * (2 * \text{ iters } + 2) * \text{ init time} \\ & + (\text{nsubdomains} \ / \ \text{nprocs}) * \text{ factor time} \\ & + \text{ iters } * (\text{nsubdomains} \ / \ \text{nprocs}) * \text{ iter time} \end{aligned}$$

where

nprocs = number of processors for a configuration

Table 2: Program X times (least-squares fit).

grid	init time	factor time	iter. time
5x5	1.0E-4	2.9E-2	2.8E-3
9x9	1.2E-4	7.7E-2	1.6E-3
13x13	1.5E-4	2.4E-1	4.3E-2
17x17	2.9E-4	5.6E-1	8.9E-2
25x25	8.6E-4	1.0E+0	2.7E-1
33x33	1.6E-3	4.9E+0	6.0E-1
49x49	5.4E-3	2.2E+1	1.8E+0

Table 3: Program H times (least-squares fit).

grid	init time	factor time	iter. time
5x5	9.5E-5	2.9E-2	2.8E-3
9x9	1.2E-4	8.2E-2	1.6E-3
13x13	1.5E-4	2.5E-1	4.4E-2
17x17	2.8E-4	5.9E-1	8.9E-2
25x25	7.5E-4	2.0E+0	2.7E-1
33x33	1.7E-3	5.4E+0	5.9E-1
49x49	5.0E-3	2.0E+1	1.9E+0

init time	= the initialization/coordination time needed for one forked process
nsubdomains	= the number of Schwarz subdomains into which the problem domain is divided
factor time	= the time needed for factorization on a subdomain
iters	= the number of iterations computed before convergence
iter time	= the time needed to solve for and copy the next iterate on one subdomain.

We ran each configuration five times, and processed the resulting data using the above model and an IMSL least-squares subroutine. Our objective was to determine how factor time and iteration time compared in each of the three programs. The results from the least-squares fit to the data are found in Tables 2, 3, and 4. For convenience, the least-squares data are plotted in graphs which may be found in Appendix B. To verify the results given by the least-squares model, we modified the programs to directly time the factorization and iteration parts of the code. The initialization time was smaller than clock resolution, so we did no measurements on that part of the programs. Comparison of the two timings shows correlation in the most significant digit. The largest degree of accuracy occurs in the finer grids, which is to be expected. The times obtained by direct measurement are found in Tables 5, 6, and 7.

3 Follow-up

We found unexpectedly that Program H and Program X performed almost identically. As the major difference between the two programs was the usage of shared memory for most of the data,

Table 4: Program S times (least-squares fit).

grid	init time	factor time	iter. time
5x5	1.4E-3	3.2E-1	5.0E-4
9x9	1.4E-3	4.3E-1	1.4E-2
13x13	1.4E-3	5.8E-1	4.2E-2
17x17	1.6E-3	9.8E-1	8.7E-2
25x25	2.1E-3	2.5E+0	2.7E-1
33x33	2.9E-3	5.9E+0	5.9E-1
49x49	7.1E-3	2.1E+1	1.9E+0

Table 5: Program X times (direct timings).

grid	factor time	iter. time
5x5	1.5e-2	4.1e-3
9x9	8.4e-2	1.7e-2
13x13	2.6e-1	4.5e-2
17x17	6.0e-1	9.2e-2
25x25	2.1e+0	2.8e-1
33x33	5.3e+0	6.1e-1
49x49	2.1e+1	1.9e+0
65x65	5.9e+1	4.3e+0

Table 6: Program H times (direct timings).

grid	factor time	iter. time
5x5	1.5e-2	4.1e-3
9x9	8.4e-2	1.7e-2
13x13	2.6e-1	4.5e-2
17x17	6.0e-1	9.2e-2
25x25	2.1e+0	2.8e-1
33x33	5.3e+0	6.1e-1
49x49	2.1e+1	1.9e+0
65x65	5.8e+1	4.3e+0

Table 7: Program S times (direct timings).

grid	factor time	iter. time
5x5	3.2e-02	2.1e-02
9x9	1.0e-01	3.4e-02
13x13	2.8e-01	6.2e-02
17x17	6.2e-01	1.1e-01
25x25	2.1e+00	3.0e-01
33x33	5.4e+00	6.3e-01
49x49	2.1e+01	1.9e+00
65x65	5.9e+01	4.4e+00

Parent Process

```
parameter n
begin
  get_time(start)
  spawn n child processes
  get_time(end)
  time = end - start
  print time
end
```

Child Process

```
begin
  fill local array with data
  barrier_wait
  for i = 1 to 20 do
    copy local array to global array
end
```

Figure 3: Psuedo-code for program to saturate shared resources.

we wanted to test exactly what load would be needed to impact program performance because of contention on the shared-memory bus. The following experiment was designed to test this.

We wrote a program which spawned one or more child processes. The child processes then simultaneously and repeatedly over-wrote a large, common, shared-memory array. The pseudo-code for this program is shown in Figure 3. We expected that adding more child processes would create congestion on the shared-memory bus, resulting in slower performance. The results showed a decrease in performance, as expected, although the decrease was much less than we expected. With 9 child processes running, shared-memory throughput was still at 89% of the speed at which a process ran without competition.

In an effort to see when the overhead of SCHEDULE would become unacceptable, we did one additional experiment involving granularity. The job we chose repeatedly updated each element of a 100x999 matrix by taking the average of its four nearest neighbors. The number of columns in the array was chosen as a programming concession to the limit in SCHEDULE of 1000 jobs. We ran the job at three levels of granularity. The smallest granularity made the computation of each point a separate process (requiring 100 calls to SCHEDULE, total of 99,900 processes). An intermediate granularity updated all points in a row (1 call to SCHEDULE, 100 processes). The largest granularity updated all points in several rows, the number of rows being determined by the

number of available processors. This version required 1 call to SCHEDULE and 1 process for each processor.

The smallest granularity compared dismally to the other two. This is not surprising, since the smallest granularity has so many more processes to track and coordinate (by 3 orders of magnitude). The two larger granularity problems were close enough in performance for 5 or more processors that they could be termed indistinguishable.

4 Conclusions

Based on our own and others' experience with programming on sequential and parallel machines, we expected the following results. First, for a given problem, the time would be smaller for configurations which used more processors, and that the speedup would be nearly linear. Second, the performance of the hybrid program would suffer because the shared-memory bus would act as a bottle-neck. Finally, because the only way to coordinate an iterative computation through SCHEDULE is to make at least one call to SCHEDULE per iteration, we expected that the SCHEDULE program would perform poorly compared to the other two. This performance would be most notable on the coarse grids, (5x5, 9x9, ...) where the work/coordination ratio is the smallest.

Indeed, the times returned by the different parallel programs did exhibit an inverse relation to the number of processors used by that program. The effect was best seen on the large problems where the ratio of work to initialization and coordination was the greatest.

However, our second assumption appears to be wrong. On the Sequent, there is no noticeable difference in performance between shared memory and local memory—even when most of the processors are working—for the kind of memory throughput our programs required. The hardware that comprises the shared-memory bus appears to be intelligent enough to cache and schedule data flow quite efficiently for a modest number of processors on our Sequent.

Lastly, the overhead caused by the use of SCHEDULE is not that significant. As expected, the relative differences are most notable on the smaller jobs. The most telling examples come, of course, from the 4x4 subdomain cases. Program X takes anywhere from one-half (1 processor) to one-fifth (8 processors) as long as Program S. But for large problems, Program S tends to be at most 5% slower than Program X, and in many cases, the difference is even smaller. For example, with 3x3 subdomains, a 49x49 grid, and 3 processors, Program X takes 589 seconds and Program S takes 593 seconds, a difference of less than 1%. For this case running only one processor, the absolute difference in time was 14 seconds. We attribute this overhead almost entirely to SCHEDULE, but relative to the time needed to run the whole program, this is not an outrageous amount of time when viewed relative to the total amount needed to reach convergence.

Programming for both Program X and Program S required some special expertise. In the case of Program X, we needed to learn the language-extension system calls which allowed user access to the parallel features of the Sequent. The functionality of the system calls was straight-forward. Such entities as barriers and spawned processes have appeared often enough in recent articles that they are no longer mysterious. However, as programmers using these system calls, we suddenly were concerned with unfamiliar issues—namely, data-flow and data allocation. Correct, reproducible results of a program run required that we identify sequences of logic which depend upon other sequences, and that data items be accessible to all processes which need them.

Learning SCHEDULE was less threatening, since many of the data flow issues were hidden in the SCHEDULE code itself, or ignored completely. The tutorial guide for using the Sequent Fortran parallel system calls describes five different kinds of variables which may appear in a DO-

loop. These variables are classified according to their usage, and the tutorial indicates how each must be handled—whether said variable should be in local or global memory, and whether access to that variable should be exclusive, which is to say, gated by a lock-unlock mechanism. Failure to analyze the DO-loop variables correctly can cause non-deterministic results.

SCHEDULE does not ask for such effort on the part of the programmer, but this neglect allows the programmer to make errors which the system calls (used properly) avoid. In our case, we avoided these errors in Program S mostly because we had already written Program X, and were consequently aware of the data flow issues.

SCHEDULE does have some rather glaring limitations. In addition to (1) the lack of barriers, or other coordinating logic; (2) the limit of 1000 jobs per invocation of SCHEDULE; (3) a limit on the number of times spawned processes may be re-entered; and (4) the job dependency language which disallows conditional and iterative execution, we would add the following.

(5) SCHEDULE has no means by which a child process can establish its identity as different from any other child process. This is an annoyance in the case where one child only needs to do a certain piece of work (such as testing for convergence). The only solution at present is to declare a global common array of integers such that A(1) = 1, A(2) = 2, ..., and to pass an array element through the calling sequence of the process. This leads to the next problem.

(6) All procedure parameters must be stored in global common; otherwise, SCHEDULE cannot find and manipulate them. Program S used much more global memory than Program X in order to conform to this SCHEDULE requirement.

There is a version of SCHEDULE which will soon be available which allows jobs to re-use job-slots (see Hanson and Sorensen [4]). In effect, the limit of 1000 jobs per invocation becomes a limit of 1000 jobs at any instant per invocation. It should be noted that SCHEDULE has subroutine calls which allow the programmer to add jobs to the queue during execution. Therefore, this change to SCHEDULE is more significant than it might appear.

SCHEDULE does suffer performance problems when asked to coordinate several small jobs, but any logic for coordinating parallel processes would be expected to perform similarly.

Overall, the overhead cost of SCHEDULE is low enough that the programmer should choose between system calls and SCHEDULE software based on personal preference. The language-extension system calls will require special scrutiny of the program logic and data flow logic. SCHEDULE will cause more problems with data organization and job sequencing. All things equal, the trade-off is between a greater effort in program analysis, and awkward data structures. If portability is an important factor however, then SCHEDULE has an obvious advantage.

References

[1] J. J. Dongarra and D. C. Sorensen, "SCHEDULE User's Guide", ANL-MCS-TM-76, Argonne National Laboratory, May 1986.

[2] J. J. Dongarra and D. C. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs", ANL-MCS-TM-86, Argonne National Laboratory, November 1986.

[3] J. J. Dongarra and D. C. Sorensen, "A Portable Environment for Developing Parallel Fortran Programs", *Parallel Computing* 5 (1987), pp 175–198.

[4] F. B. Hanson and D. C. Sorensen, "The SCHEDULE Parallel Programming Package with Recycling Job Queues and Iterated Dependency Graphs", *Concurrency: Practice and Experience*,

(to appear).

[5] P. L. Lions, "On the Schwarz alternating method I", in *Domain Decomposition Methods for Partial Differential Equations*, (R. Glowinski, G. Golub, G. Meurant and J. Periaux, eds.), SIAM, Philadelphia, 1988, pp. 1–42.

[6] P. L. Lions, "On the Schwarz alternating method II: stochastic interpretation and order properties", in *Domain Decomposition Methods*, (T. Chan, R. Glowinski, J. Periaux and O. Widlund, eds.), SIAM, Philadelphia, 1989, pp. 47–70.

[7] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York, 1985.

[8] J. R. Rice, E. N. Houstis and W. R. Dyksen, "A population of linear, second order elliptic partial differential equations on rectangular domains", *Math. Comp.*, 36(1981), pp 479–484.

[9] W. P. Tang, "Schwarz splitting and template operators", Ph.D. thesis, Stanford University, 1987.

Appendix

A Average of times recorded from runs of all programs

Program X

2 x 2 Subdomains

grid	iters	time (p=4)	time (p=2)	time (p=1)	error
5	31	0.19	0.28	0.49	1.7e-02
9	36	0.73	1.37	2.61	3.2e-03
13	39	2.02	3.91	7.77	9.7e-04
17	41	4.32	8.52	16.96	4.4e-04
25	44	14.27	28.42	56.60	1.9e-04
33	46	32.97	65.72	130.97	1.0e-04
49	48	111.32	221.79	442.48	4.3e-05
65	50	272.10	544.02	1082.12	2.3e-05

3 x 3 Subdomains

grid	iters	time (p=9)	time (p=3)	time (p=1)	error
5	59	0.35	0.73	1.84	7.4e-03
9	68	1.38	3.67	10.53	1.1e-04
13	74	3.79	10.55	31.31	3.9e-04
17	78	7.95	22.93	67.97	2.0e-04
25	83	26.04	74.26	223.37	8.9e-05
33	87	59.96	172.76	513.57	4.7e-05
49	93	203.37	587.80	1744.08	2.0e-05

4 x 4 Subdomains

grid	iters	time (p=8)	time (p=4)	time (p=2)	time (p=1)	error
5	97	0.92	1.53	2.71	5.14	3.9e-03
9	113	4.23	7.87	15.34	30.36	5.3e-04
13	123	11.83	22.84	45.18	89.89	2.0e-04
17	129	25.40	49.41	97.50	193.90	1.2e-04
25	139	84.24	163.55	324.39	643.60	5.0e-05
33	145	191.26	373.01	738.21	1470.74	2.9e-05

Program H

2 x 2 Subdomains

grid	iters	time (p=4)	time (p=2)	time (p=1)	error
5	31	0.17	0.29	0.48	1.7e-02
9	36	0.73	1.37	2.65	3.2e-03
13	39	2.03	3.94	7.83	9.7e-04
17	41	4.30	8.57	17.03	4.4e-04
25	44	14.31	28.50	56.69	1.9e-05
33	46	33.12	65.80	131.00	1.0e-04
49	48	111.78	221.48	441.77	4.3e-05
65	50	273.28	541.92	1081.53	2.3e-05

3 x 3 Subdomains

grid	iters	time (p=9)	time (p=3)	time (p=1)	error
5	59	0.36	0.75	1.86	7.4e-03
9	68	1.39	3.66	10.57	1.1e-03
13	74	3.80	10.63	31.48	3.9e-04
17	78	7.97	23.04	68.27	2.0e-04
25	83	26.20	75.66	224.38	8.9e-05
33	87	60.22	173.32	515.90	4.7e-05
49	93	204.47	588.61	1752.73	2.0e-05

4 x 4 Subdomains

grid	iters	time (p=8)	time (p=4)	time (p=2)	time (p=1)	error
5	97	0.90	1.54	2.71	5.20	3.9e-03
9	113	4.21	7.94	15.36	30.46	5.3e-04
13	123	11.85	22.99	45.46	90.37	2.0e-04
17	129	25.32	49.63	97.75	194.43	1.2e-04
25	139	84.29	163.59	324.49	645.86	5.0e-05
33	145	191.44	372.99	738.62	1469.55	2.9e-05

Program S

2 x 2 Subdomains

grid	iters	time (p=4)	time (p=2)	time (p=1)	error
5	31	1.34	1.47	1.69	1.7e-02
9	36	2.12	2.72	4.06	3.2e-03
13	39	3.47	5.44	9.33	9.7e-04
17	41	5.83	10.10	18.60	4.4e-04
25	44	16.03	30.23	58.38	1.9e-04
33	46	34.86	67.54	132.82	1.0e-04
49	48	113.64	223.71	444.21	4.3e-05
65	50	275.44	543.85	1084.52	2.3e-05

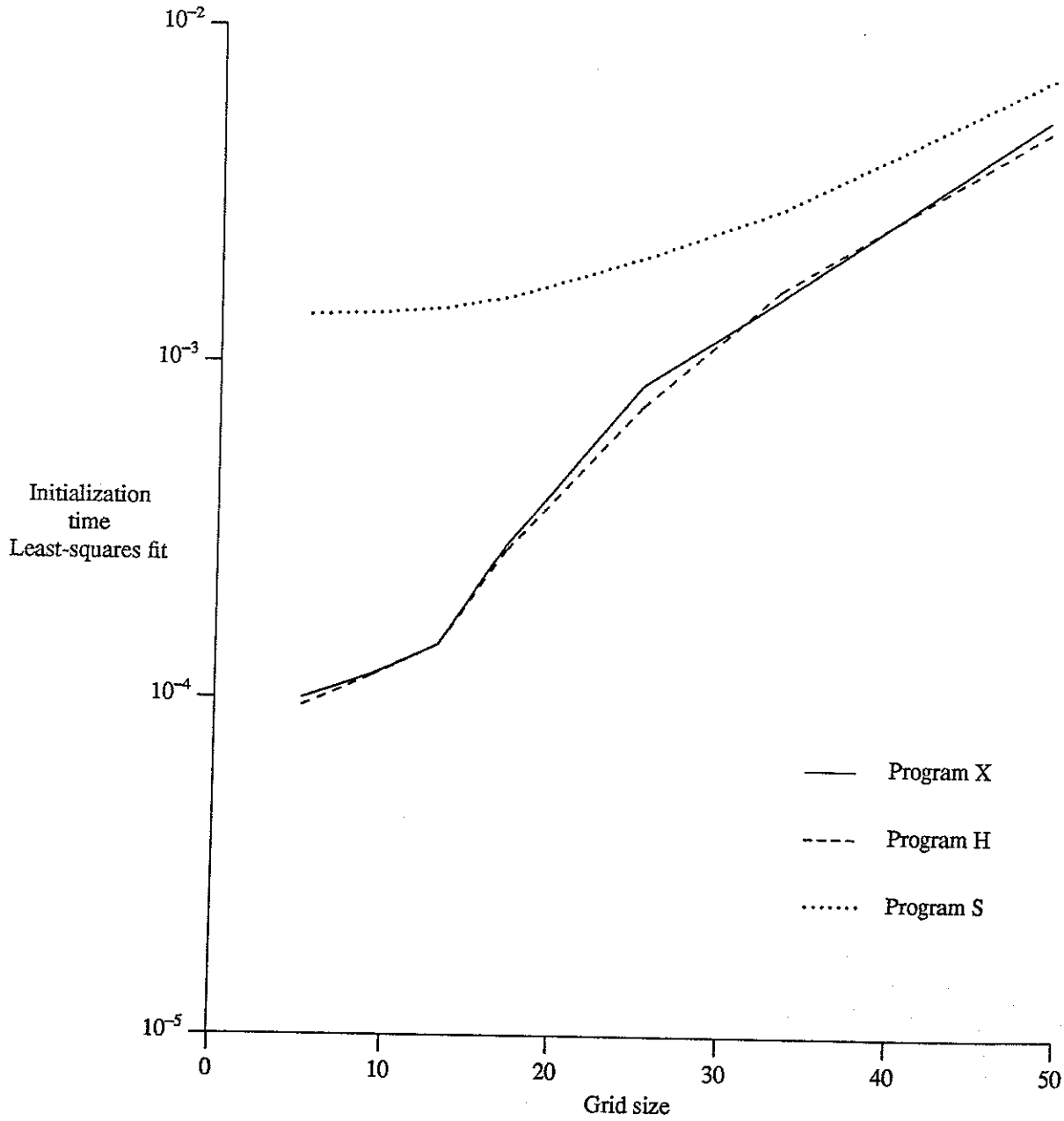
3 x 3 Subdomains

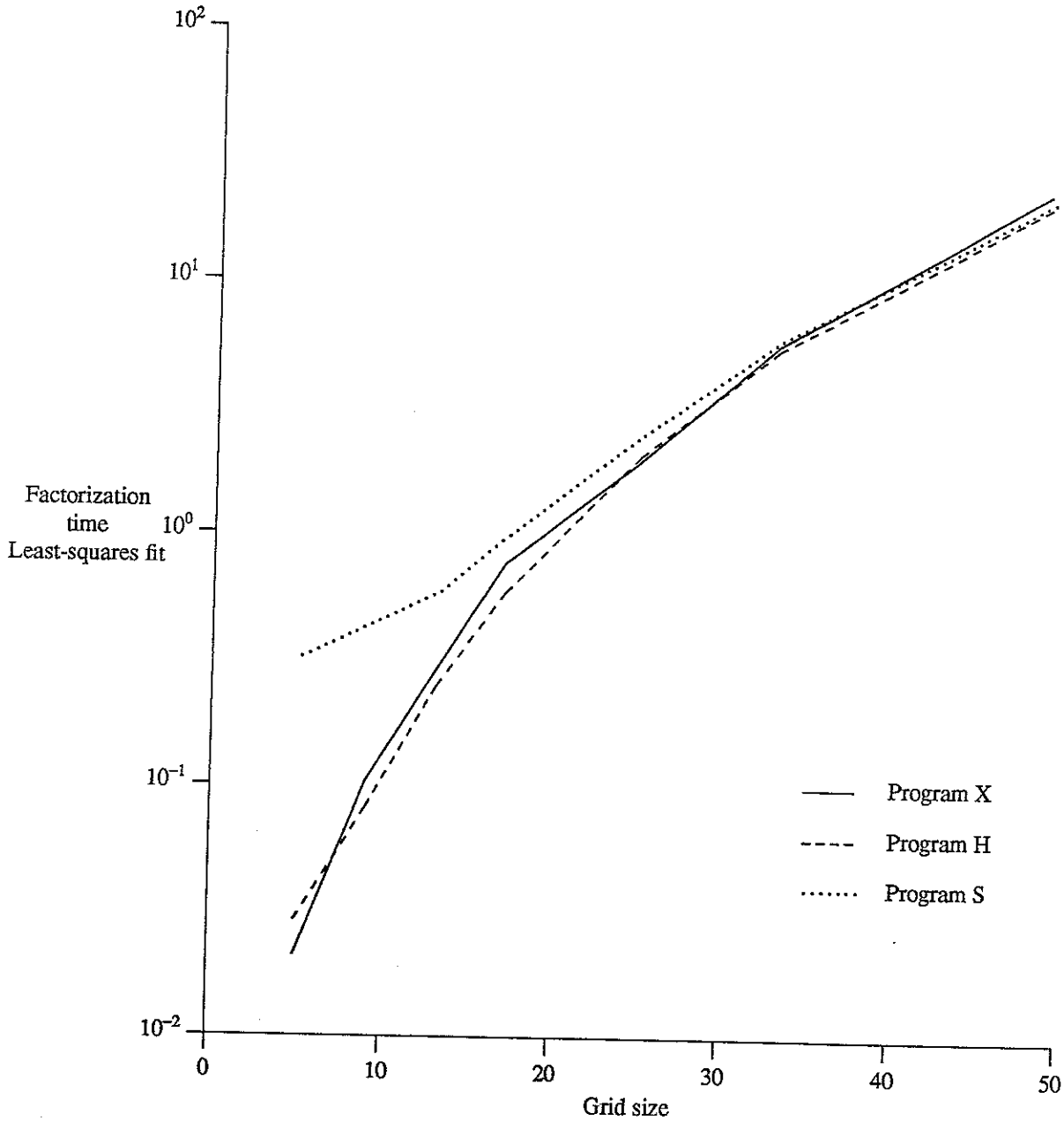
grid	iters	time (p=9)	time (p=3)	time (p=1)	error
5	59	2.67	3.14	4.52	7.4e-03
9	68	4.06	6.40	13.66	1.1e-03
13	74	6.73	13.66	34.74	3.9e-04
17	78	11.10	26.19	71.67	2.0e-04
25	83	29.45	79.19	228.30	8.9e-05
33	87	63.55	176.95	520.11	4.7e-05
49	93	208.70	593.27	1758.05	2.0e-05

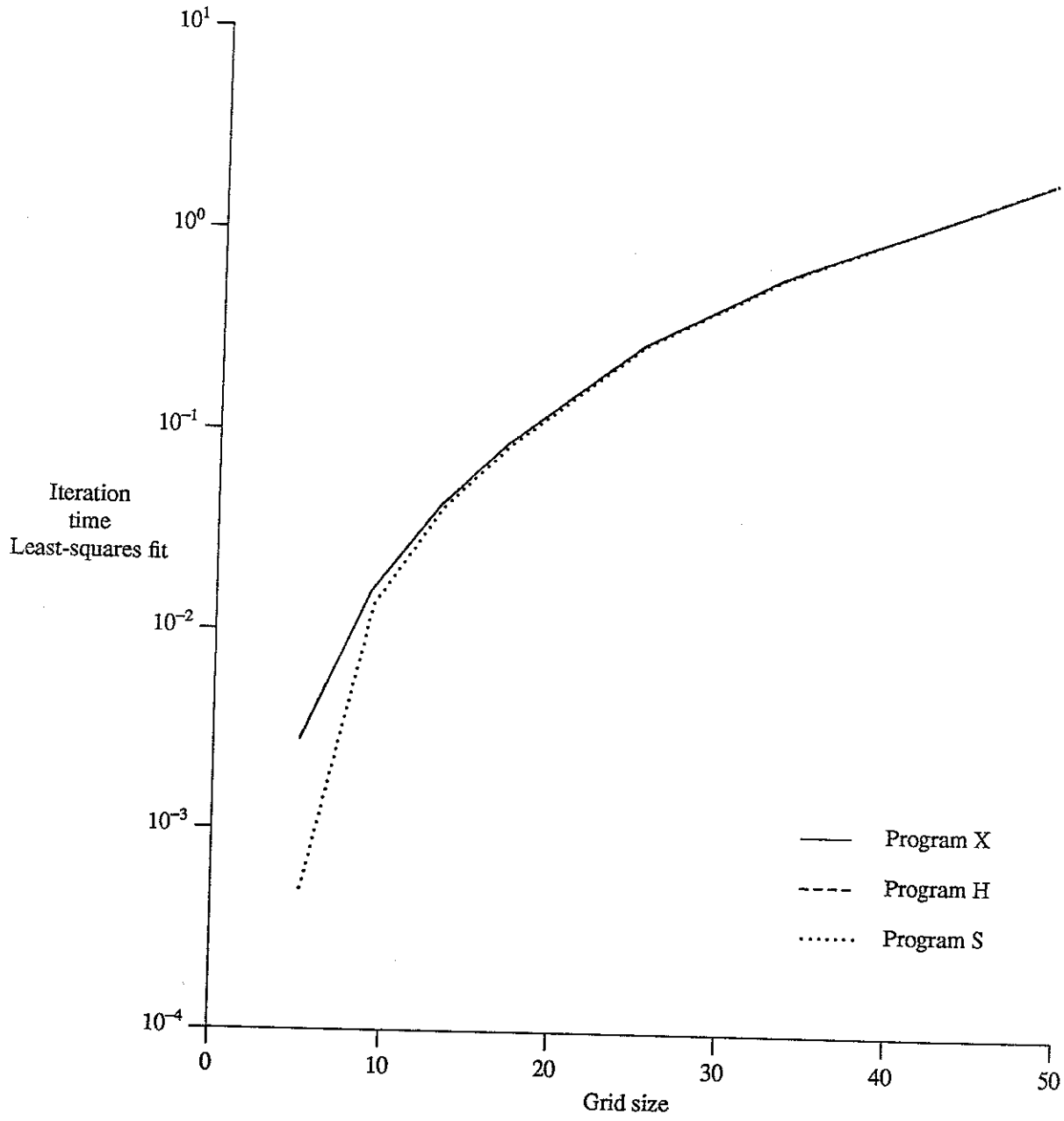
4 x 4 Subdomains

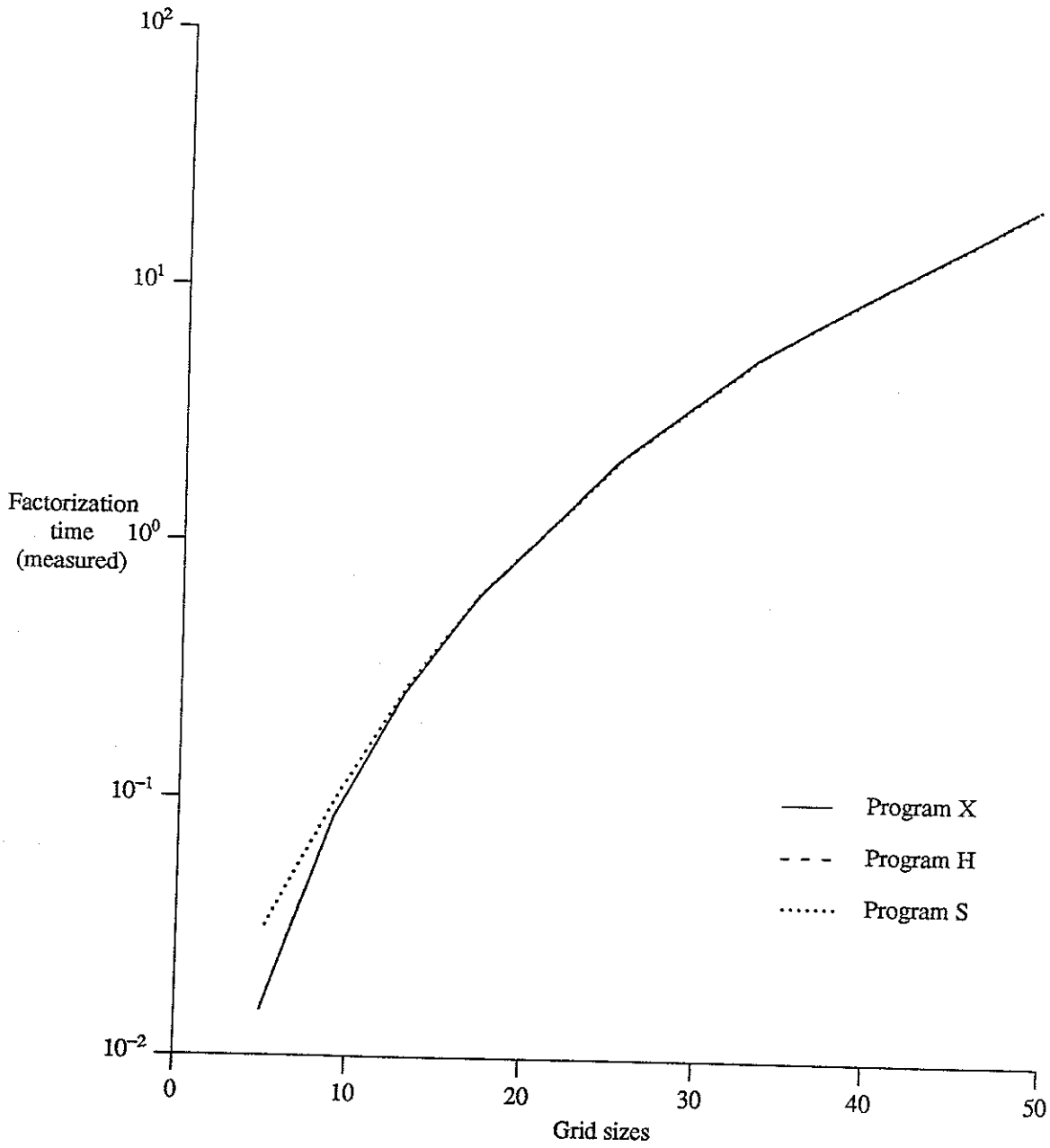
grid	iters	time (p=8)	time (p=4)	time (p=2)	time (p=1)	error
5	97	5.09	5.82	7.52	10.59	3.9e-03
9	113	9.08	12.97	20.87	36.78	5.3e-04
13	123	17.32	28.81	51.85	97.95	2.0e-04
17	129	31.05	55.23	104.30	201.51	1.2e-04
25	139	90.16	170.09	331.45	653.59	5.0e-05
33	145	198.17	379.81	747.32	1480.58	2.9e-05

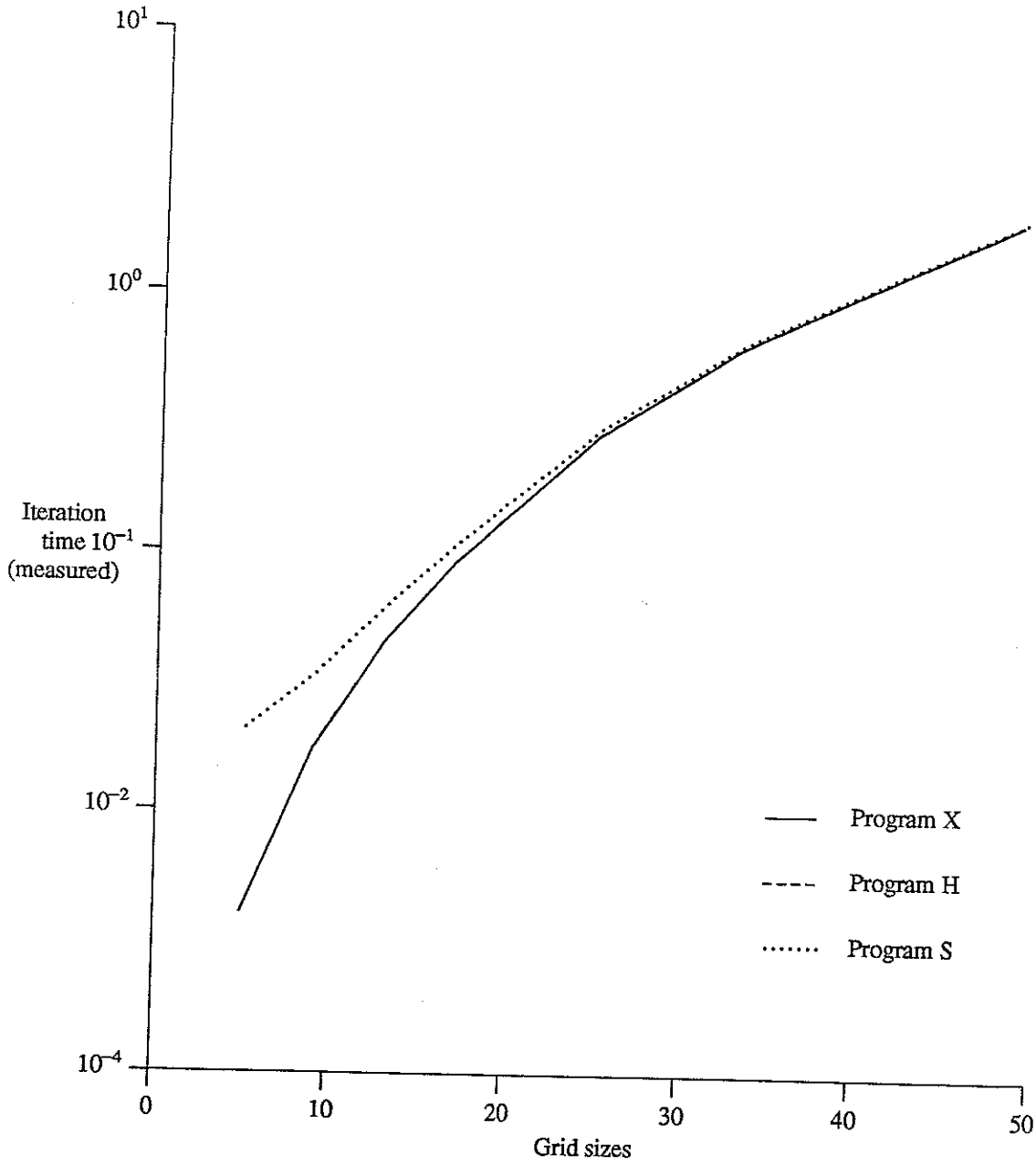
B Plots of log(time) vs. grid size for initialization, factorization, iteration.




10^2
10^1
10^0
10^{-1}
10^{-2}
Factorization time
Least-squares fit
0
10
20
30
40
50
Grid size
Program X
Program H
Program S


10^1
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
Iteration
time
Least-squares fit
0
10
20
30
40
50
Grid size
Program X
Program H
Program S


10^2
10^1
Factorization time (measured)
10^0
10^{-1}
10^{-2}
0
10
20
30
40
50
Grid sizes
Program X
Program H
Program S


10^1
10^0
Iteration time 10^{-1} (measured)
10^{-2}
10^{-4}
0
10
20
30
40
50
Grid sizes
Program X
Program H
Program S