Domain Decomposition vs. Concurrent Multigrid

Eric Van de Velde

CRPC-TR94598
August 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Also available as *CRPC-94-11* from the Center for Research on Parallel Computation at the California Institute of Technology.

Domain Decomposition vs. Concurrent Multigrid [1]

Eric F. Van de Velde
Applied Mathematics 217-50
Caltech
Pasadena, CA 91125

evdv@ama.caltech.edu

CRPC-94-11
August 29, 1994

[1]This material is based upon work supported by the NSF under Cooperative Agreement No. CCR-9120008. The government has certain rights in this material.

Abstract

We compare the performance on a concurrent computer of two domain-decomposition methods and a full-multigrid method for solving the Poisson equation on a square. Independent of the granularity of the computation, domain decomposition is almost always significantly slower than multigrid. Our largest computations use a multicomputer with 128 nodes to solve a problem on a 2048×2048 grid. In these computations, multigrid outperforms domain decomposition by at least four orders of magnitude.

1 Introduction

Although there are many variants, there are only two basic types of domain-decomposition methods. The oldest variety is due to Schwarz [7] and dates back to 1869. The Schwarz-iteration method decomposes a domain into two or more overlapping subdomains. In 1986, Bjørstad and Widlund[1] proposed a domain-decomposition method with nonoverlapping subdomains. Originally, these methods were used to reduce problems on irregular domains to problems on regular domains. More recently, domain decomposition has been used as a technique for concurrent computing. The proceedings of several international conferences [3, 4] cover these and other developments.

Multigrid methods have gained an enormous popularity since 1977, when Brandt [2] showed that they are practical fast solution methods. Concurrent implementations were developed as soon as concurrent computers became available.

Our computational experiment compares the two types of domain decomposition and a full-multigrid method. Each numerical method solves the same problem to the same error tolerance. With our implementations, we found that domain decomposition is almost always slower than multigrid. As the size of the problem increases, the performance difference increases. For our largest problems, multigrid outperforms domain decomposition by several orders of magnitude.

Section 2 summarizes our experimental results. Section 3 on performance analysis outlines the major issues that are encountered when measured execution times are used to compare and to evaluate numerical methods. Section 4 discusses the specific test problems used in our experiment.

Sections 5, 6, and 7 give a brief overview of the programs. Implementation details are important, particularly because the magnitude of the difference in performance is beyond any prior expectation. Unfortunately, only the most important aspects of each program can be discussed within the space limitations of a paper. Van de Velde [9] gives a high-level derivation of the two most important concurrent programs: multigrid (Chapter 9) and domain decomposition with nonoverlapping subdomains (Chapter 10). Readers who wish *all* the technical details may request from the author the program listings, documented by an electronic supplement [10] to this paper.

Any experiment is necessarily limited in scope; this experiment is not an exception. Section 8 examines these limitations and studies known variants of domain decomposition and their potential for bridging the observed performance gap.

2 Results of the Experiment

Speed-up and efficiency graphs are the traditional tools to display and to analyze the performance of concurrent programs. These graphs can be extremely misleading, however, because an inaccurate sequential-execution time can destroy the validity of the whole graph. Unfortunately, inaccurate sequential-execution times occur frequently, because they are often obtained with nonoptimal methods and with nonoptimized or insufficiently optimized sequential programs.

To avoid such problems, we use logarithmic execution-time plots, which plot the execution times against the number of nodes used. Although the sequential-execution time remains important to interpret some results, an inaccurate sequential-execution time does not invalidate the whole plot. A detailed discussion on logarithmic execution-time plots is found in Chapter 1 of Van de Velde [9].

Ideally, a computation using twice the number of nodes should finish in half the time. Such ideal performance is called linear speed-up. Because of the logarithmic scales, lines of linear speed-up are straight lines with a known slope. The dotted lines in our plots show this slope. Given a sequential-execution time and the slope of linear speed-up, one can draw a line of linear speed-up. The vertical distance of a computation to its line of linear speed-up is inversely related to the efficiency of that computation. Efficient computations lie in the proximity of their line of linear speed-up.

In our plots, we identify numerical methods by the following symbols:

SZ(k,SOR) Schwarz iteration with an overlap of k grid cells and a subdomain solver based on successive overrelaxation with optimal parameter.

DD(SOR) Nonoverlapping domain decomposition with a subdomain solver based on successive overrelaxation with optimal parameter.

FMG(k) Full-multigrid method with k coarser levels underneath the finest level; the total number of levels is $k + 1$.

In our computations, the number of processes always equals the number of nodes. There remains a considerable freedom in choice of process mesh. For example, 32 processes can be configured as six different two-dimensional process meshes: 32×1, 16×2, 8×4, 4×8, 2×16, or 1×32. The choice of process mesh has an impact on performance through area-perimeter considerations. In the case of domain-decomposition methods, it also has an impact on the convergence rate of the method. Preliminary studies, which are not reported here, show that all methods perform best on square meshes. Therefore, all reported execution times are obtained with one of the following process meshes:

Number of Nodes							
1	2	4	8	16	32	64	128
1×1	2×1	2×2	4×2	4×4	8×4	8×8	16×8
Process Mesh							

Figures 1 through 4 display the performance of our methods when used to solve the Poisson equation on a square. The grids range in size from 16×16 to 256×256. Each plot compares computations that solve identical problems to identical accuracy requirements. To compute the error on the numerical solution, we compare it to the known exact solution of the continuous problem. Within a plot, the only variables are the numerical method and the number of nodes. Computations of the same plot solve the identical problem. However, computations of different plots solve different problems: not only the grid size differs, but also the exact solution of the continuous problem. On finer grids, we solve a continuous problem whose solution is more oscillatory. This is also discussed in Section 4.

Figure 1. For this 32×32 problem, SZ(3,SOR) can use at most 64 nodes. To see this, consider using 128 nodes and a 16×8 process mesh. In this case, the data for the outer ghost boundary must be retrieved from a nonneighboring process in the process mesh. Our implementation does not allow this.

The coarsest grid of FMG(2) has size 8×8 and can be distributed over at most 64 nodes. As a result, our implementation of FMG(2) can use at most 64 nodes. Similarly, our implementation of FMG(3) can use at most 16 nodes because its coarsest grid has size 4×4. FMG(3) on 16 nodes is the fastest computation among all tested methods.

Multigrid with the largest number of levels performs the best among all tested methods. The multicomputer inefficiency of the coarser levels does not negate the numerical benefits of introducing the coarse levels. For all three multigrid methods, the performance curve as a function of the number of nodes is rather Ωat. Although FMG(3) is the overall fastest method, its speed-up or efficiency plot would be quite disappointing. This is to be expected for a small problem.

The slope of the performance curve of DD(SOR) is more encouraging. If one were to use the sequential-execution time of DD(SOR) to compute speed-ups and efficiencies, DD(SOR) would do very well in a speed-up or in an efficiency graph. However, it does not make sense to use DD(SOR) for a sequential computation. DD(SOR) performs two successive-overrelaxation iterations on the complete grid. The first iteration is superΩuous and computes the residual of the capacitance system, which we know to be zero. The second iteration computes the interior unknowns. The sequential-execution time of DD(SOR) is, therefore, twice that of successive overrelaxation and about ten times that of FMG(3).

Theory dictates that one compute speed-ups and efficiencies with respect to the best sequential-execution time. In practice, that minimum is not known. Figure 1 shows how dangerous it is in concurrent computing to ignore the best sequential methods. If we would have been unaware of the multigrid computations, we could easily have arrived at a completely different conclusion.

The Schwarz-iteration methods are not competitive, neither in execution time, nor in speed-up, nor in efficiency.

Figure 2. SZ(1,SOR) on more than 16 nodes did not converge within 512 Schwarz-iteration steps. FMG(3) can use at most 64 nodes and FMG(4) at most 16 nodes because of the data distribution of the coarsest grids. FMG(2) requires one multigrid-iteration step per level. However, FMG(3) and FMG(4) barely miss the accuracy requirement with one step per level and require two multigrid-iteration steps per level. FMG(4) on 16 nodes is the fastest computation.

As in Figure 1, multigrid with the maximum number of levels wins. Schwarz iteration is not competitive. DD(SOR) requires substantially more nodes to achieve a performance that is competitive with multigrid.

When the number of nodes is between 2 and 64, the performance curve of DD(SOR) is steeper than the lines of linear speed-up. This case shows how misleading speed-up and efficiency plots can be. With an erroneous or inaccurate sequential-execution time, one might easily conclude that DD(SOR) achieves "superlinear speed-up." The occurrence of superlinear speed-up is possible only because DD(SOR) computations on few nodes are particularly slow. However, when the number of nodes is increased, superlinear speed-up does recoup some of the early damage.

Figure 3. We did not time some of the Schwarz iterations on few nodes, because these computations required an excessive time. SZ(1,SOR) on more than 8 nodes does not converge within 512 Schwarz-iteration steps. As in Figure 2, DD(SOR) shows characteristics of superlinear speed-up between 2 and 64 nodes. FMG(4) can use at most 64 nodes and FMG(5) at most 16

nodes because of the data distribution of the coarsest grid. FMG(5) barely misses the accuracy requirement with one multigrid-iteration step per level and needs two steps. FMG(3) and FMG(4), on the other hand, converge in one multigrid-iteration step per level. As a result, multigrid with the largest number of levels no longer wins. FMG(4) on 64 nodes is the fastest computation. Compared with multigrid, all tested domain-decomposition methods are noncontenders.

Figure 4. For these computations on a 256×256 grid, the accuracy requirement is relaxed, because the highly oscillatory test problem is difficult to approximate to within the original strict error criterion (whether using multigrid or any of the domain-decomposition methods). DD(SOR) on 2 nodes requires 20,440 seconds (about five and a half hours), and this computation is out of range of the plot. DD(SOR) on 32 nodes does not converge within 512 conjugate-gradient-iteration steps. In fact, this iteration seemed stalled. After 128 iteration steps, the error was 1.71×10^{-3}. After 512 iteration steps, which required 3,210 seconds, the error was hardly changed at 1.70×10^{-3}. FMG(5) can use at most 64 nodes and FMG(6) at most 16 nodes because of the data distribution of the coarsest grid. Our implementation of FMG(6) loses out to FMG(5), because the latter can run on more nodes. With a different implementation, which would allow duplicating instead of distributing the coarsest grids, FMG(6) could beat FMG(5). For the present implementation, FMG(5) on 64 nodes is the fastest computation.

This 256×256 problem is large enough for multigrid to exhibit behavior of "efficient" methods: the computations lie on a line that is almost parallel to the line of linear speed-up. This is not the case in previous plots, where the problems are too small. Slopes in the logarithmic execution-time plot show that multigrid is less efficient with fewer levels. This may come as a surprise, since computations with fewer levels use finer grids, which have a more favorable ratio of arithmetic over communication. The reason for loss of efficiency is load imbalance. If the coarsest grid is fine, the coarsest-grid solver requires many Jacobi-relaxation steps. These amplify the slight load imbalance of the data distribution of the coarsest grid, which is caused by the compatibility requirements between levels.

DD(SOR) underperforms FMG(4), FMG(5), and FMG(6) by at least two orders of magnitude. However, our current implementation does not allow using FMG(5) with more than 64 nodes. Considering that DD(SOR), once again, shows characteristics of superlinear speed-up, one might raise the possibility that DD(SOR) might beat multigrid by using many nodes. Assume that DD(SOR) continues the same steep descent beyond 128 nodes.

Graphically, it is easily verified that DD(SOR) with about 2048 nodes breaks even with FMG(5) on 64 nodes. In reality, the break-even point would be further, because DD(SOR) performance would Ωatten out. Moreover, by duplicating instead of distributing the coarsest grid, one could easily increase the performance of FMG(5). With this fairly straightforward algorithmic change, FMG(5) could run on more than 64 nodes. This would put the break-even point even further out of reach.

Figures 5 and 6. In these figures, we compare the execution times of one iteration step. Note, however, that full multigrid computes the solution to prescribed tolerance in one iteration step per level. FMG(4) in Figure 5 and FMG(5) in Figure 6 coincide with FMG(4) in Figure 3 and FMG(5) in Figure 4, respectively. The execution time of DD(SOR) with one iteration step includes the computation of the residual, which occurs before the first iteration step, and the computation of the subdomain interiors, which occurs after the last iteration step.

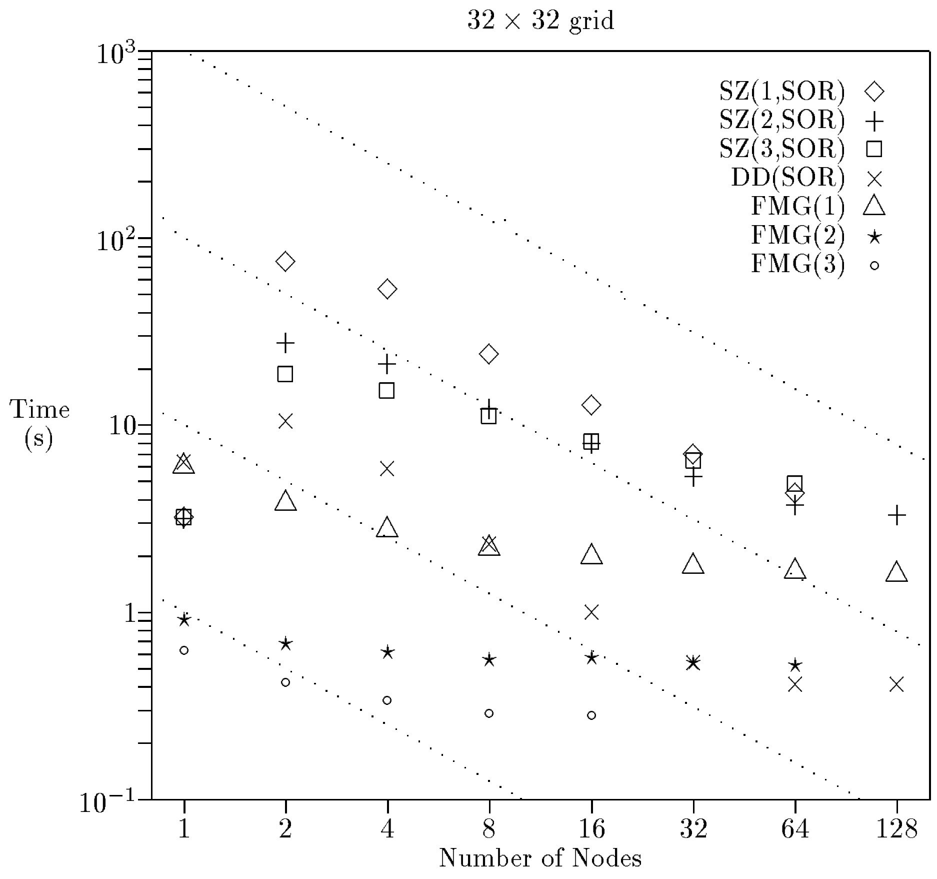

Figure 1: Logarithmic execution-time plot for problem on 32×32 grid solved to a tolerance $\tau = 9.77 \times 10^{-4}$.

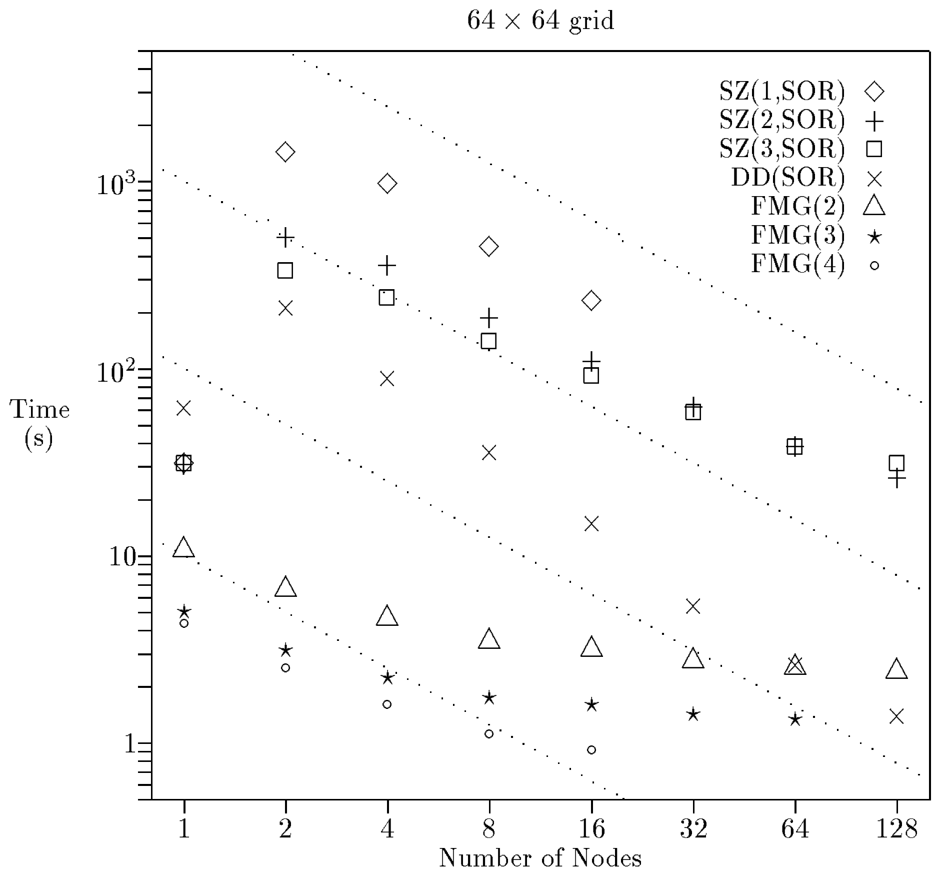

Figure 2: Logarithmic execution-time plot for problem on 64×64 grid solved to a tolerance $\tau = 2.44 \times 10^{-4}$.

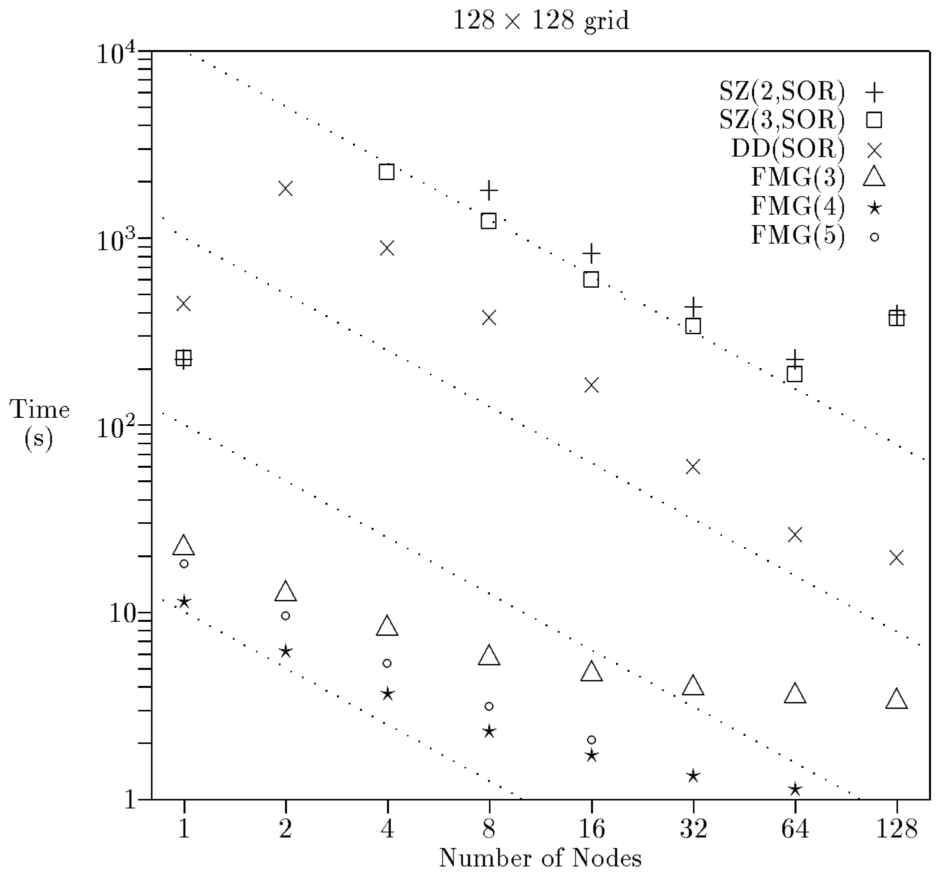

Figure 3: Logarithmic execution-time plot for problem on 128×128 grid solved to a tolerance $\tau = 6.10 \times 10^{-4}$.

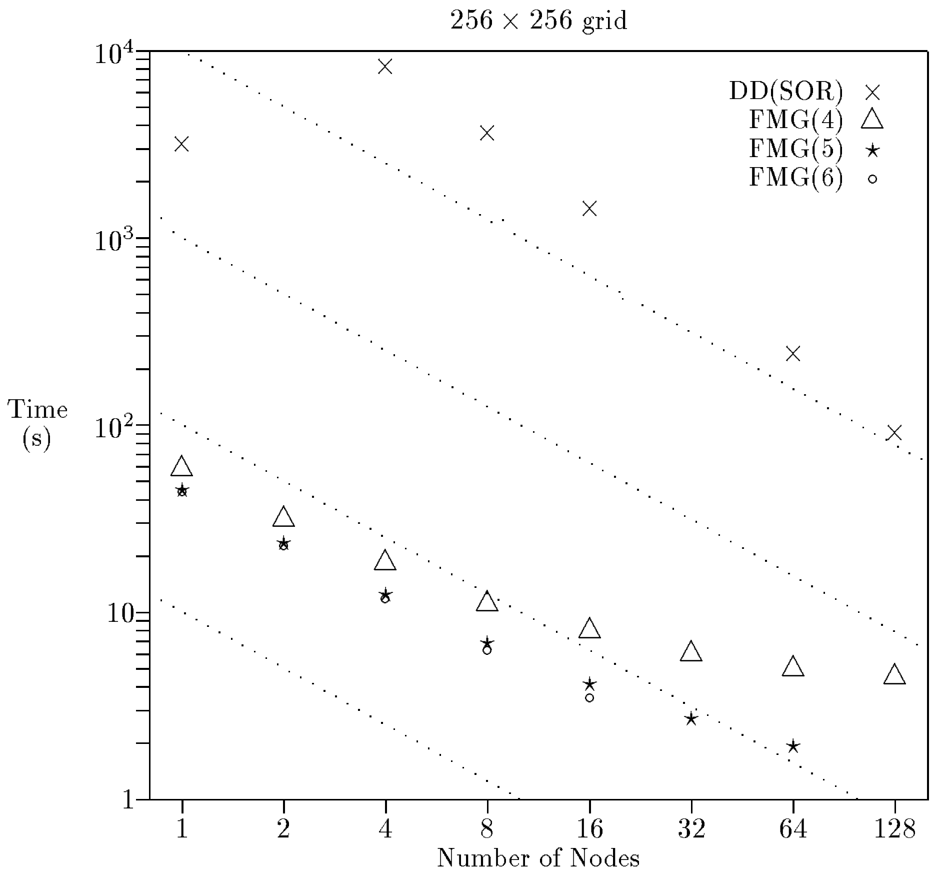

Figure 4: Logarithmic execution-time plot for problem on 256×256 grid solved to a tolerance $\tau = 1.53 \times 10^{-3}$.

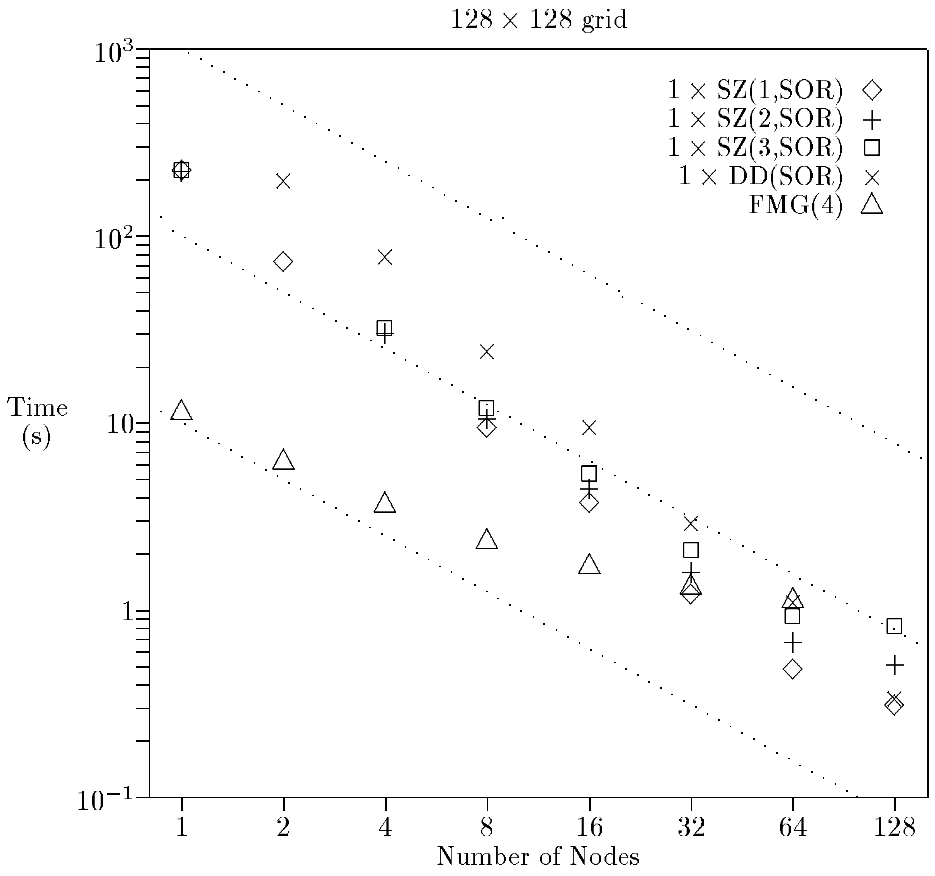

Figure 5: Logarithmic execution-time plot for one iteration step of problem on 128 × 128 grid.

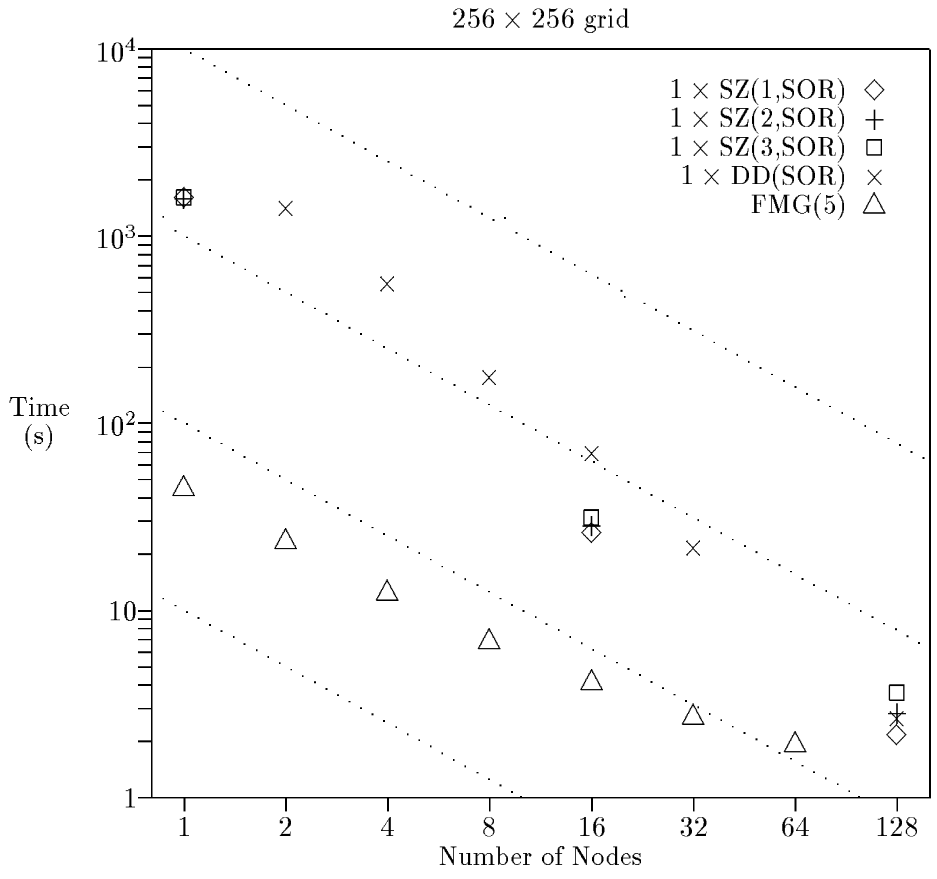

Figure 6: Logarithmic execution-time plot for one iteration step of problem on 256×256 grid.

Grid	Tolerance	Method	K	Error	Time (s)
512×512	3.81×10^{-4}	SZ(3,SOR)	1	2.60×10^{0}	26.16
		DD(SOR)	1	2.69×10^{0}	42.61
			178	3.81×10^{-4}	2653.00
		FMG(5)	1	5.19×10^{-4}	9.73
			2	2.85×10^{-4}	11.59
1024×1024	9.54×10^{-4}	DD(SOR)	1	4.24×10^{0}	309.70
			128	9.21×10^{-2}	13,900.00
			256	4.32×10^{-4}	27,580.00
		FMG(6)	1	7.72×10^{-4}	11.66
2048×2048	2.38×10^{-3}	DD(SOR)	1	?	± 1150.00
			192	2.77×10^{-1}	74,490.00
		FMG(7)	1	7.27×10^{-4}	29.65

Table 1: Large problems computed on 128 nodes.

Table 1. For concurrent computing, we are most interested in the execution times for the largest feasible problems. First, consider execution times of converged iterations, for which the actual error is less than the tolerance. For the 512×512 problem, DD(SOR) requires 178 iteration steps and about 45 minutes. The same problem with FMG(5) has converged in two iteration steps per level and in less than twelve seconds. For the 1024×1024 problem, DD(SOR) requires at least 128 and at most 256 iteration steps and an execution time between four and eight hours. Compare this with FMG(6), which converges with two steps per level and in less than twelve seconds. FMG(7) solves the 2048×2048 problem in about half a minute. DD(SOR) requires more than twenty hours to reach an accuracy that is three orders of magnitude worse. FMG(7) outperforms DD(SOR) by at least four orders of magnitude!

As in Figures 5 and 6, the execution times of DD(SOR) with $K = 1$ include the residual calculation and the interior calculation, which occur, respectively, before the first and after the last iteration step. For the 2048×2048 problem, the listed execution time is an estimate. It equals $\frac{3}{194}$ times the execution time of DD(SOR) with 192 iteration steps. This estimate assumes that the execution times of the residual calculation, the interior calculation, and one plain iteration step of DD(SOR) are the same.

Memory. Any computation uses two resources: processor time and memory. Thus far, our discussion has centered around number of nodes and execution time. Because it often determines the maximum feasible problem size, memory is a significant performance measure. For multicomputer computations with one process per node, we are primarily interested in memory use per process. Because virtual memory is either unavailable or inefficient on current multicomputers, computations are infeasible if any process requires more memory than is available on a node.

Our multigrid implementation requires about 50% more memory than the other methods, because it allocates not only a solution and a right-hand-side field but also a residual field. The extra field is convenient, because the right-hand side of the coarse problem can be computed in two easy steps. First, the residual is computed on the fine grid. Second, this fine-grid residual is restricted to the coarse right-hand side. It is possible to eliminate the residual field by combining residual calculation and restriction into one rather messy operation. In that case, memory use per node would not differ substantially from one method to the next. The coarse grids of multigrid require about the same amount of memory as the extra ghost boundaries of the domain-decomposition methods (see Section 7 and Figure 12).

3 Performance Analysis

The execution time of a computation is the most important performance-analysis measure. Under normal circumstances, this measure depends on many factors. When developing a computational experiment, one must set up the experimental conditions such that the measured execution times are relevant for the purpose in mind. In this section, we discuss the principles that guided the development of our experiment, which is intended to compare numerical methods.

A *computation* is a *program* with particular *input data* executed on a given *computer* at a particular *moment in time.* A computation occurs each time a program is executed on a computer, and every computation establishes a certain execution time. One program is always associated with an unlimited number of computations and, hence, with an unlimited number of execution times. From these, we wish to gain insights into aspects of the program, the input data, or the computer. This task is complicated, because the relationships between programs, input data, computers, and computations are complicated.

In the space of all computations, each element is defined by a set of "coordinates." This set includes the program, the input data, the computer, and the moment in time. Obviously, each of these coordinates is quite complicated just by itself. The design of a performance-analysis experiment must simplify this unwieldy space of computations. Ideally, a computation should be defined in terms of a few interesting parameters. The remaining coordinates that define each computation should be carefully controlled and, ideally, kept constant.

Classical analysis of iterative methods for linear systems as developed by Young [12] computes a convergence rate. Given this rate, one can estimate the number of iteration steps until convergence from the magnitude of the initial error and the required accuracy on the result. This type of analysis is actually an application of performance analysis as defined here. The convergence rate is inversely proportional to the execution time on an abstract computer that performs one iteration step per unit of time. The use of this abstract computer simplifies the space of computations to the point where a computation is defined by the problem, the magnitude of the error on the initial guess, and the tolerance on the computed solution. In this simplified space of computations, one can study how the abstract execution time (the convergence rate) depends on the remaining parameters. When using convergence-rate results to compare different iterative methods, one is actually comparing execution times on different abstract computers.

Performance analyses that rely on measured execution times cannot use abstraction to simplify the space of computations: an execution time is obtained on a given computer with a given program for given input data. If any of these coordinates is not fully specified, it is impossible to start a computation, let alone to obtain an execution time. A numerical method does not define the whole program. A program also incorporates an arbitrary number of implementation decisions. We must set up the experiment such that the measured execution times are relevant for comparing numerical methods and not for comparing implementation decisions.

If it is our goal to assess numerical methods and not the patience and diligence of the implementors, then we must compare execution times of comparable implementations. For example, it would be a bad idea to compare implementations in different programming languages or, more subtle, in different programming styles. For this reason, the implementations were kept as simple as possible. Low-level optimizations were avoided: all loops are written with standard index notation, not the more efficient but hardly readable pointer arithmetic of C. To maintain complete control over the implementation, we did not use any libraries except for the communication library of the Reactive Kernel/Cosmic Environment operating system.

Other issues are less obvious when assessing whether or not implementations are comparable. For example, how does one objectively assess that two programs implementing different numerical methods are equally robust? This is usually a subjective judgment call, because robustness can depend strongly on choice of error estimator and heuristics. By changing either, one may drastically change robustness as well as execution times. When error estimation could not be avoided, we used simple nonadaptive schemes.

The space of feasible input data is always extremely large. In any experiment, only a few test problems can be used. Although objective criteria can limit the choice, they cannot fully specify the set of test problems.

Finally, one must examine whether the results are valid on any other computer besides the one of the experiment. On sequential computers, one may translate many performance results from one computer to the next. Although the design of concurrent computers has not converged to as strong a degree as that of sequential computers, we are confident in wide applicability of our results. Programs may not be portable from a multiprocessor (shared-memory computer) to a multicomputer (local-memory computer), but the performance issues remain the same. Data locality, memory latency, and message latency must be addressed on all concurrent computers, albeit with different notation in different programming models. Nevertheless, we are interested in repeating the experiment on different platforms, and our programs are written with portability in mind.

4 The Test Problem

Solving the Poisson problem with Dirichlet-boundary conditions on a rectangle $\Omega = (0, L_x) \times (0, L_y)$ is a minimum requirement for any proposed solver. This standard comparison problem is defined by

$$\begin{cases} \forall (x,y) \in \Omega : & -\Delta u = f(x,y) \\ \forall (x,y) \in \partial\Omega : & u(x,y) = g(x,y). \end{cases} \tag{1}$$

We introduce an $M \times N$ grid defined by grid points $(x_m, y_n) = (md_x, nd_y)$, where $0 \le m \le M$, $0 \le n \le N$, $d_x = L_x/M$, and $d_y = L_y/N$. The exterior-boundary points are those for which $m = 0$, $m = M$, $n = 0$, or $n = N$. The continuous problem is discretized on this grid using the classical second-order difference scheme, which results in:

$$\begin{array}{l} 0 < m < M,\ 0 < n < N: \\ \quad 2(\frac{1}{d_x^2}+\frac{1}{d_y^2})u_{m,n} - \frac{1}{d_x^2}(u_{m+1,n}+u_{m-1,n}) - \frac{1}{d_y^2}(u_{m,n+1}+u_{m,n-1}) = f_{m,n} \\ 0 \le m \le M: u_{m,0} = g_{m,0},\ u_{m,N} = g_{m,N} \\ 0 \le n \le N:\ \ u_{0,n} = g_{0,n},\ u_{M,n} = g_{M,n}. \end{array}$$

In these equations, $f_{m,n} = f(x_m, y_n)$, $g_{m,n} = g(x_m, y_n)$, and $u_{m,n}$ is the numerical approximation to $u(x_m, y_n)$.

To obtain execution times, we solve the problem with exact solution

$$u(x,y) = e^{2x+y}\cos(Mx/35)\cos(Ny/25)$$

on domain $\Omega = (0,1)\times(0,1)$. Our test problem is more oscillatory for larger grids, because only difficult problems require fine grids. When comparing numerical methods, computed solutions must satisfy the same error tolerance τ. We must make sure that the computed solution $u_{m,n}$ satisfies:

$$\frac{1}{L_xL_y}\sum_{m=0}^{M}\sum_{n=0}^{N}(u_{m,n}-u(x_m,y_n))^2 d_x d_y < \tau^2. \tag{2}$$

Because the exact solution $u(x,y)$ of (1) is known, this is easily verified.

The tolerance τ should be an appropriate function of grid size. Because the discretization is second-order accurate, we choose

$$\tau = \frac{d^2}{L^2}, \tag{3}$$

where d is a representative length scale for a grid cell and L is a representative length scale for the global domain. The problems on the finest grids (256×256 and up) are very oscillatory. For these problems, none of our numerical methods can achieve the accuracy requirement (2) with the tolerance τ of (3). In these cases, we multiplied τ by 10 until the relaxed accuracy requirement could be satisfied. This explains why τ is not monotonically decreasing with grid size in our experiment.

In addition to the length scales d and L, we also need a representative length scale ℓ for a subdomain. Some formulas simplify considerably if we choose these length scales as follows:

$$d^2 = \frac{d_x^2 d_y^2}{2(d_x^2+d_y^2)},\quad \ell^2 = \frac{\ell_x^2\ell_y^2}{2(\ell_x^2+\ell_y^2)},\quad \text{and}\quad L^2 = \frac{L_x^2L_y^2}{2(L_x^2+L_y^2)}. \tag{4}$$

We only consider subdomains of Ω that are rectangles of size $\ell_x\times\ell_y$ with ℓ_x and ℓ_y integer multiples of d_x and d_y, respectively.

```
void    restrict_grid(F,C)
grid    *F, *C ;
{       int     i, j, i2, j2, Ip, Jq, sp, sq ;
        double  **c, **f ;

        sp = 2*C->Mp-F->Mp ; sq = 2*C->Nq-F->Nq ;
        Ip = C->Ip ;         Jq = C->Jq ;
        c  = C->s ;          f  = F->s ;

        exch_grid_ghost(F) ;
        for ( i=0, i2=sp ; i<Ip ; i++, i2+=2 )
        for ( j=0, j2=sq ; j<Jq ; j++, j2+=2 )
            c[i][j] = 0.25*f[i2][j2] +
                    0.1250*(f[i2-1][j2]+f[i2][j2-1]
                           +f[i2+1][j2]+f[i2][j2+1]) +
                    0.0625*(f[i2-1][j2-1]+f[i2-1][j2+1]
                           +f[i2+1][j2-1]+f[i2+1][j2+1]) ;
}
```

Figure 7: The restriction operator based on full-weight restriction.

5 Full Multigrid

Our multigrid program achieves its concurrency by pure data distribution: numerically, there is no difference between a concurrent and a sequential multigrid computation. Because we use Jacobi smoothing and a Jacobi coarsest-grid solver, even the round-off errors are identical. There is only one significant data-distribution issue: the compatibility of data distributions of different levels. Once this compatibility between levels is achieved, smoothing, restriction, and prolongation operators only need one communication operation: a boundary exchange. This is implemented by procedure `exch_grid_ghost`, which is not displayed.

With compatible data distributions, the intergrid transfers are easy. Figure 7 shows procedure `restrict_grid`, which implements full-weight restriction. This procedure takes a fine grid `F` and a compatible coarse grid `C` as parameters and computes coarse-grid values as the weighted sum of surrounding fine-grid values. Prolongation based on linear interpolation has a similar structure, but is somewhat more cumbersome, because there are more cases to consider.

```
void    smooth_Poisson_grid(K,U,F)
int     K ;
grid    *U, *F ;
{       int     k, i, j, Ip, Jq ;
        double  **u,**f,a,ax,ay,tmp, omg=2/3.0, omg1=1/3.0 ;

        ax = 1.0/(U->dx*U->dx) ; ay = 1.0/(U->dy*U->dy) ;
        a  = 0.5*omg/(ax+ay) ;
        ax = a*ax ;              ay = a*ay ;
        u  = U->s  ; f  = F->s ;
        Ip = U->Ip ; Jq = U->Jq ;
        for ( k=0 ; k<K ; k++ ) {
            for ( j=0 ; j<Jq ; j++ ) buf2[j] = u[-1][j] ;
            for ( i=0 ; i<Ip ; i++ ) {
                for ( j=-1 ; j<Jq ; j++ ) buf1[j] = u[i][j] ;
                for ( j=0 ; j<Jq ; j++ ) {
                    tmp = a*f[i][j]+ax*(buf2[j]+u[i+1][j])
                                   +ay*(buf1[j-1]+u[i][j+1]) ;
                    u[i][j] = tmp+omg1*u[i][j] ;
                }
                for ( j=0 ; j<Jq ; j++ ) buf2[j] = buf1[j] ;
            }
            exch_grid_ghost(U) ;
        }
}
```

Figure 8: The smoothing operator based on Jacobi underrelaxation.

Figure 8 displays the smoothing procedure based on Jacobi underrelaxation. Multigrid spends the majority of its execution time in this routine. The parameters of procedure `smooth_Poisson_grid` are the number of smoothing steps `K`, the solution grid `U`, and the right-hand-side grid `F`. The underrelaxation parameter is stored in variable `omg`. The Jacobi underrelaxation procedure uses two buffers `buf1` and `buf2`, which are declared and created externally to this procedure.

We use Jacobi underrelaxation with parameter `omg` $= 2/3$. For the one-dimensional Poisson problem, every such iteration step multiplies the high-frequency errors by an amplification factor of at most $1/3$. For two-

```
void    mg_Poisson_grid(K,L,U,F,R)
int     K, L ;
grid    **U, **F, **R ;
{       int     k ;

        if ( L==0 ) {
                solve_Poisson_grid(U[0],F[0]) ;
                return ;
        }
        for ( k=0 ; k<K ; k++ ) {
                smooth_Poisson_grid(3,U[L],F[L]) ;
                residual_Poisson_grid(U[L],F[L],R[L]) ;
                restrict_grid(R[L],F[L-1]) ;
                zero_grid(U[L-1]) ;
                mg_Poisson_grid(1,L-1,U,F,R) ;
                prolongadd_grid(U[L-1],U[L]) ;
        }
        smooth_Poisson_grid(3,U[L],F[L]) ;
}
```

Figure 9: The multigrid program.

dimensional problems, Wesseling [11] argues that omg = 4/5 achieves an optimal smoothing factor of 2/5. Our smoother and, hence, our multigrid method may be less than optimal.

Figure 9 displays the basic multigrid program. The five parameters of mg_Poisson_grid are the number of multigrid iterations K, the level L, and arrays of pointers to grids that represent the solution U, the right-hand-side function F, and the residual R on every level. The multigrid procedure uses L + 1 levels, level 0 being the coarsest and level L the finest.

If L = 0, procedure mg_Poisson_grid calls solve_Poisson_grid, the coarsest-grid solver. For the coarsest-grid problem, we do not worry about performance and use Jacobi relaxation, which is the easiest to implement. Isaacson and Keller [6] show that this scheme has a convergence rate

$$R_J = \frac{\pi^2}{2}\frac{d^2}{L^2}.$$

The length scales L and d are as defined in (4) but with the footnote that d is the length scale of the coarsest grid.

The coarsest-grid solver applies K Jacobi-relaxation steps to the coarsest-grid problem. We compute this number K as follows. Let ϵ_0 be the error on the initial guess. Let ϵ_K be the error after K Jacobi-relaxation steps. Given the Jacobi-convergence factor ρ_J and the Jacobi-convergence rate $R_J = -\log \rho_J$, we make the heuristic assumption that

$$\epsilon_K \approx \rho_J^K \epsilon_0.$$

Requiring this expression to be less than a tolerance τ, we obtain that

$$K \approx \frac{\log(\epsilon_0/\tau)}{R_J}.$$

As in (3), we choose $\tau = d^2/L^2$ with d the length scale of the coarsest grid. We assume that $\epsilon_0 = O(1)$. The program, rather arbitrarily, sets $\epsilon_0 = 2$.

If L $\geq$ 1, procedure **mg_Poisson_grid** performs K multigrid-iteration steps on level L. Each step smoothes level L with three steps of Jacobi underrelaxation. The residual of level L is restricted to level L $-$ 1. In V-cycle multigrid, procedure **mg_Poisson_grid** calls itself recursively to apply one multigrid-iteration step to the problem on level L$-$ 1. The prolongation operator corrects level L using the solution of level L $-$ 1.

The multigrid-iteration step performs three additional smoothing steps to eliminate high-frequency errors due to the prolongation; this is known as post-correction smoothing. Three post-correction smoothing steps is considered high. One could increase the performance of the multigrid iteration by eliminating one or two post-correction smoothing steps. (Usually, post-correction smoothing is put inside the loop. Because post-correction smoothing is immediately followed by pre-correction smoothing, our version is equivalent in all but the first and last iteration steps.)

Figure 10 displays procedure **fmg_Poisson_grid**, which implements the full-multigrid method. The second part is the most interesting. A solution is computed on the coarsest grid (level 0) by calling **mg_Poisson_grid**. The coarsest-grid solution is interpolated to level 1 and is used as the initial guess to a two-level multigrid procedure. The resulting solution on level 1 is interpolated to level 2 and used as the initial guess for a three-level multigrid procedure. This is continued until the finest level is computed.

The first part of **fmg_Poisson_grid** is somewhat awkward. Here, boundary and right-hand-side information is transmitted from the calling program to the full-multigrid procedure in such a way that the initialization of the right-hand side and the boundary is excluded from the measured execution

```
void     fmg_Poisson_grid(K,L,U,F,R)
int      K, L ;
grid     **U, **F, **R ;
{        int     l ;

         for ( l=L ; l>0  ; l-- )
             restrict_grid(F[l],F[l-1]) ;
         copy_grid_ghost_ext(F[0],U[0]) ;

         mg_Poisson_grid(1,0,U,F,R) ;
         for ( l=1 ; l<=L ; l++ ) {
             prolong_grid(U[l-1],U[l]) ;
             copy_grid_ghost_ext(F[l],U[l]) ;
             mg_Poisson_grid(K,l,U,F,R) ;
         }
}
```

Figure 10: The full-multigrid program.

times. Before calling **fmg_Poisson_grid**, the interior of grid `F[L]` is initialized by the right-hand-side function $f(x, y)$, and the exterior boundary of `F[L]` is initialized by the Dirichlet-boundary function $g(x, y)$. The first for-loop over `l` in **fmg_Poisson_grid** initializes the right-hand sides of all coarser problems by means of successive full-weight restriction. Calls to procedure **copy_grid_ghost_ext** copy the Dirichlet-boundary data from the right-hand-side grid `F[l]` to the solution grid `U[l]`.

Procedure **fmg_Poisson_grid** performs `K` multigrid-iteration steps on every level. A suitable value for `K` is chosen as follows. First, we time the full-multigrid procedure with one multigrid-iteration step per level. If the computed solution satisfies (2), we accept this computation and its execution time. Otherwise, we increment the number of multigrid-iteration steps per level until the computed solution is sufficiently accurate. For some problems, it is impossible to satisfy (2) independent of the number of multigrid-iteration steps per level. In these cases, the procedure is repeated after multiplying τ by 10. With this a-posteriori procedure, measured execution times do not include any arithmetic due to error estimation.

6 The Schwarz-Iteration Method

Domain decomposition achieves concurrency by numerically splitting up the original problem into a set of subproblems. Each subdivision of the global domain leads to a different numerical procedure. In our implementation, each subdomain is mapped to a separate process, and the number of subdomains equals the number of processes. As a result, computations with different numbers of processes use different numerical procedures. This stands in fundamental contrast to concurrent programs that achieve concurrency by pure data distribution.

The domain-decomposition method of Schwarz is based on overlapping subdomains. One Schwarz-iteration step consists of the simultaneous solution of all subdomain problems followed by a boundary exchange that initializes the Dirichlet-boundary values of the subdomain problems.

The amount of overlap can be varied. It must be at least one grid cell. The upper limit on the overlap follows from the constraint that boundary data for the subdomain problems must be available in neighboring processes. This is violated as soon as any of the local-grid dimensions is less than the amount of overlap.

It is an advantage of the Schwarz iteration that one can use sequential solvers on the subdomains. We use successive overrelaxation with optimal parameter. This procedure is almost optimal and easy to implement. Isaacson and Keller [6] show that the convergence rate is given by

$$R_{SOR} = 2\pi \frac{d}{\ell}.$$

Note that we use the length scale ℓ of the subdomain. The computation of the number of SOR-iteration steps K follows the same heuristic procedure as the one used in the coarsest-grid solver of multigrid. We find that

$$K \approx \frac{\log(\epsilon_0/\tau)}{R_{SOR}}.$$

We set $\epsilon_0 = 2$ (rather arbitrarily), and we choose $\tau = d^2/L^2$ to ensure that every subdomain is solved to the same accuracy.

The *asymptotic* convergence rate R_{SOR} underestimates the *observed* convergence rate; this fact is discussed by Young [12]. As a result, the computed number of iteration steps does not solve the subproblems to sufficient accuracy. For this reason, the program reduces the convergence rate R_{SOR} by the heuristic factor `alpha` $= 0.5$, which doubles the number of iteration

```
void    Poisson_Schwarz(K,U,F)
int     K ;
grid    *U, *F ;
{       double  RGS, omg ;
        int     k, K_1 ;

        omg = omega(...) ;
        RGS = -alpha*log1p(omg-2.0) ;
        K_1 = log(eps0/tau)/RGS ; K_1++ ;

        for ( k=0 ; k<K ; k++ ) {
            overrelax_Poisson_Schwarz(K_1,omg,U,F) ;
            exch_grid_ghost(U) ;
        }
}
```

Figure 11: Domain decomposition with overlap.

steps. Young suggested a better remedy: set the overrelaxation parameter equal to one in the first iteration step and use the optimal parameter thereafter. This suggestion came to our attention too late to be incorporated in the current experiment. However, the subdomain solver would speed up, at most, by a factor of two. Although significant, this cannot possibly bridge the performance gap between domain decomposition and multigrid.

Procedure `Poisson_Schwarz` of Figure 11 performs `K` Schwarz-iteration steps. The computation of the overrelaxation parameter `omg` is omitted for brevity. The asymptotic convergence rate `RGS` is reduced by the heuristic factor `alpha`. The variable `eps0` estimates the magnitude of the error on the initial guess, and `tau` is the error tolerance for the subdomain solver. The number of overrelaxation steps on the subdomain is computed in `K_1`. The variables `eps0`, `tau`, and `alpha` are initialized externally to the procedure. The procedure `overrelax_Poisson_Schwarz` is not displayed; it is almost identical to procedure `overrelax_Poisson_domain` of Figure 14.

The Schwarz iteration must compute the global solution to the same tolerance as multigrid. The execution time is the time required by the minimum number of Schwarz-iteration steps for which the computed solution is sufficiently near the known exact solution.

7 Domain Decomposition Without Overlap

When the subdomains do not overlap, the unknowns are split into two subsets. The first subset consists of boundary unknowns, which are associated with grid points on the boundaries between subdomains. The second subset consists of interior unknowns, which are associated with grid points located in the interior of the subdomains. As soon as the boundary unknowns are computed, it is trivial to compute the interior unknowns using a sequential Poisson solver on each subdomain. Therefore, we focus on developing a procedure for the boundary unknowns.

The boundary unknowns solve a reduced system of linear equations, called the capacitance system. Although the coefficient matrix of this system is dense, there exists a procedure to compute the product of the capacitance matrix and an arbitrary vector of boundary values. As a result, one can use Krylov-type iterations to solve the capacitance system.

Procedure `domdec_Poisson_domain` of Figure 12 solves the capacitance system using the conjugate-gradient method. (In the present case, we know that the capacitance system has a symmetric positive-definite coefficient matrix.) Once the boundary unknowns are computed to sufficient accuracy, all the other unknowns are computed in procedure `solve_Poisson_domain`, which is called in the last line of procedure `domdec_Poisson_domain`.

The conjugate-gradient iteration requires several vectors: the search direction $\vec{p}$, the residual $\vec{r}$, the vector $\vec{w} = C\vec{p}$, and the current solution $\vec{x}$. Each component of these vectors is attached to a specific grid point on the subdomain boundaries. The vector components are, therefore, stored in a set of extra ghost boundaries around the subdomains. Since boundaries are shared by up to four subdomains, the data distribution partially duplicates the boundary vectors, such that each subdomain has access to all components attached to grid points on its boundary. The solution grid `U` needs three and the right-hand-side grid `F` needs two ghost boundaries. In the program, the pair `U,0` refers to ghost boundary number `0` of grid `U`.

Procedure `capacitance_Poisson_domain_ghost` of Figure 13 applies the capacitance matrix to a search direction $\vec{p}$, which is stored in ghost boundary `U,0`. This search direction defines the Dirichlet-boundary values for a Laplace equation on each subdomain. These subdomain problems are solved by a call to procedure `solve_Poisson_domain`. After a ghost-boundary exchange performed by procedure `exch_domain_ghost`, the five-point stencil is applied to the subdomain boundaries. The resulting vector $\vec{w}$ is stored in ghost boundary `F,0`.

When procedure `capacitance_Poisson_domain_ghost` calls procedure `solve_Poisson_domain` of Figure 14, it is to solve Laplace problems on the subdomains. When `domdec_Poisson_domain` calls `solve_Poisson_domain`, it is to solve Poisson problems on the subdomains. In principle, any known sequential solver can be used to solve all subdomain problems. Here, we use successive overrelaxation, because it is near optimal and easy to implement. In Section 8.4.2, we shall consider other possible choices.

Procedure `solve_Poisson_domain` computes the number of iteration steps and the overrelaxation parameter. As discussed in Section 6, the asymptotic convergence rate of successive overrelaxation is reduced by a heuristic factor of 0.5, the magnitude of the error of the initial guess ϵ_0 is arbitrarily set equal to 2, and we choose $\tau = d^2/L^2$. In the program, the variables `alpha`, `eps0`, and `tau` are all initialized externally to the procedure.

Procedure `overrelax_Poisson_domain` has four parameters: the number of relaxation steps `K`, the overrelaxation parameter `omg`, the solution grid `U`, and the right-hand-side grid `F`. Procedure `overrelax_Laplace_domain` is identical to procedure `overrelax_Poisson_domain`, except that the right-hand side vanishes.

To obtain an execution time that can be compared with multigrid, domain decomposition must compute the solution to the same error tolerance. To avoid including error estimation in our execution times, we use a preliminary run to find the minimum number of conjugate-gradient-iteration steps to satisfy the error tolerance. Subsequently, we time a computation with the same number of iteration steps, but without computing any errors along the way. It is the execution time of this computation that we accept.

```
void    domdec_Poisson_domain(K,U,F)
int     K ;
domain  *U, *F ;
{       int     k ;
        double  xi, bt, rr0, rr1, pw, err, corr ;

        residual_Poisson_domain_ghost(U,F) ;
        copy_domain_ghost(U,0,U,2) ;
        copy_domain_ghost(F,0,F,1) ;
        copy_domain_ghost(F,0,U,0) ;
        rr0 = iprd_domain_ghost(F,0,F,0) ;
        for ( k=0 ; k<K && rr0>0.0 ; k++ ) {
            capacitance_Poisson_domain_ghost(U,F) ;
            pw = iprd_domain_ghost(U,0,F,0) ;
            xi = rr0/pw ;
            wsum_domain_ghost(U,2,U,2,xi,U,0) ;
            wsum_domain_ghost(F,1,F,1,-xi,F,0) ;
            rr1 = iprd_domain_ghost(F,1,F,1) ;
            bt = rr1/rr0 ;
            wsum_domain_ghost(U,0,F,1,bt,U,0) ;
            rr0 = rr1 ;
        }
        copy_domain_ghost(U,2,U,0) ;
        solve_Poisson_domain(U,F) ;
}
```

Figure 12: Domain decomposition without overlap.

```
void    capacitance_Poisson_domain_ghost(U,F)
domain  *U, *F ;
{       int     i, j, Ip, Jq ;
        double  **u, **f, a, ax, ay ;

        u  = U->s ;                f  = F->s ;
        Ip = U->Ip ;               Jq = U->Jq ;
        ax = 1.0/(U->dx*U->dx) ;  ay = 1.0/(U->dy*U->dy) ;
        a  = 2.0*(ax+ay) ;

        solve_Poisson_domain(U,NULL) ;
        exch_domain_ghost(U) ;
        for ( i=-1 ; i<=Ip ; i++ ) {
           j = -1 ;
           f[i][j] = a*u[i][j]-ax*(u[i+1][j]+u[i-1][j])
                              -ay*(u[i][j+1]+u[i][j-1]) ;
           j = Jq ;
           f[i][j] = a*u[i][j]-ax*(u[i+1][j]+u[i-1][j])
                              -ay*(u[i][j+1]+u[i][j-1]) ;
        }
        for ( j=0 ; j<Jq ; j++ ) {
           i = -1 ;
           f[i][j] = a*u[i][j]-ax*(u[i+1][j]+u[i-1][j])
                              -ay*(u[i][j+1]+u[i][j-1]) ;
           i = Ip ;
           f[i][j] = a*u[i][j]-ax*(u[i+1][j]+u[i-1][j])
                              -ay*(u[i][j+1]+u[i][j-1]) ;
        }
        zero_domain_ghost_ext(F,0) ;
}
```

Figure 13: Applying the capacitance matrix to a search direction stored in a ghost boundary of U.

```
void    overrelax_Poisson_domain(K,omg,U,F)
int     K ;
double  omg ;
domain  *U, *F ;
{       int     i, j, k, Ip, Jq ;
        double  **u, **f, a, ax, ay, tmp, omg1 ;

        omg1 = 1.0-omg ;
        ax = 1.0/(U->dx*U->dx) ; ay = 1.0/(U->dy*U->dy) ;
        a  = 0.5*omg/(ax+ay) ;
        ax = a*ax ;              ay = a*ay ;
        u  = U->s ;              f  = F->s ;
        Ip = U->Ip ;             Jq = U->Jq ;

        for ( k=0 ; k<K ; k++ )
            for ( j=0 ; j<Jq ; j++ )
            for ( i=0 ; i<Ip ; i++ ) {
                tmp = a*f[i][j]+ax*(u[i-1][j]+u[i+1][j])
                               +ay*(u[i][j-1]+u[i][j+1]) ;
                u[i][j] = tmp+omg1*u[i][j] ;
            }
}
void    solve_Poisson_domain(U,F)
domain  *U, *F ;
{       int     K ;
        double  R, omg ;

        omg= omega(U->dx,U->dy,U->d2,U->Ip+1,U->Jq+1) ;
        R  = -alpha*log1p(omg-2.0) ;
        K  = log(eps0/tau)/R ; K++ ;
        if (F)  overrelax_Poisson_domain(K,omg,U,F) ;
        else    overrelax_Laplace_domain(K,omg,U) ;
}
```

Figure 14: Subdomain solver based on Gauss-Seidel relaxation with optimal overrelaxation parameter.

8 Limitations of the Experiment

Any computational experiment is necessarily limited: a limited set of test problems, a limited set of variations to the basic numerical methods, a limited set of computational parameters (like tolerance, number of processes, process mesh, number of smoothing steps, V- or W-cycle, etc.), a limited number of computers on which the experiment is run. Having performed a limited experiment, we should indeed be careful before drawing a general conclusion. Obviously, it would be of great interest to repeat the experiment for a much wider variety of problems, numerical methods, etc.

The following subsections challenge the limitations of our experiment and examine whether these limitations could significantly change the outcome. Because of the significant performance difference, we are also concerned with the question whether the experiment is biased in favor of multigrid.

8.1 The Problem

Beyond the Poisson equation. The Poisson equation with Dirichlet-boundary conditions on a rectangular grid is, admittedly, a fairly straightforward test problem. Is it too simple for the purpose of our experiment?

It is almost a given that convergence factor, operation count, and other complexity measures will increase as the problem deviates from the canonical Poisson problem. Although it is far from obvious how to extrapolate performance of the Poisson problem to performance of more general problems, the Poisson problem is a widely-accepted benchmark for the evaluation of any numerical method for large sparse systems. Nevertheless, the choice of test problem remains a limitation of the experiment.

For the Poisson problem, Jacobi smoothing is one of the best smoothers available. Different problems require other carefully-chosen smoothing operators. Strongly-anisotropic problems, for example, require smoothers based on line relaxation. Other problems need other modifications of the basic method. Nevertheless, the multigrid approach is numerically effective over an impressive application range, which includes nonlinear elliptic equations and compressible and incompressible Ωuid dynamics. Multigrid methods for these problems are often significantly less efficient on multicomputers than the tested method.

The conjugate-gradient method requires a symmetric positive-definite capacitance system. This severely limits the choice of test problem for domain decomposition. If the capacitance system is not symmetric positive-definite, a general iterative solver is required. This adds substantial complications. The quasi-minimal-residual method, for example, requires at least 13 auxiliary vectors stored in ghost boundaries of the solution and right-hand-side fields. This is an extravagant memory overhead. Moreover, every iteration step computes a matrix-vector and a transpose-matrix-vector product.

Although many other solvers are available, all are substantially more complicated than the basic conjugate-gradient method. This virtually excludes any problem with a nonsymmetric or nonpositive-definite capacitance system from the application range of domain decomposition without overlap. As shown in Figures 5 and 6 and the last line of Table 1, just one step of the elementary conjugate-gradient procedure is too expensive. Therefore, one cannot expect that a more complicated outer-iteration scheme will perform any better.

The original convergence proof of the Schwarz-iteration method required the maximum principle. Although more recent proofs relax this requirement, the application range of the Schwarz method is still restricted.

When deviating from the Poisson equation, multigrid performance will suffer. The situation for domain-decomposition methods is worse, however, because problems quickly fall outside the range of feasible applications.

Grids. Domain-decomposition methods may have advantages for computations on irregular grids. This is not explored in our experiment.

On regular grids, multigrid methods have the disadvantage that grid dimensions should be a multiple of powers of two. If this is not the case, exotic restriction and interpolation operators must be developed. With these, it is difficult to find data distributions that ensure compatibility between multigrid levels.

Three-dimensional problems. We only studied two-dimensional problems. Could domain decomposition perform better when applied to three-dimensional problems? In fact, the analysis in Van de Velde [9] already indicates that domain decomposition without overlap is not suitable for three-dimensional problems. This conclusion is reached purely on grounds of memory overhead for the ghost boundaries. The same conclusion is valid for domain decomposition with large overlap. In view of our two-dimensional results, it is highly unlikely that domain decomposition with small overlap could outperform multigrid in three dimensions, but the formal tests have not been performed.

Test problems. The timings presented in this paper are obtained using a very limited set of right-hand-side functions $f(x, y)$ and boundary-value

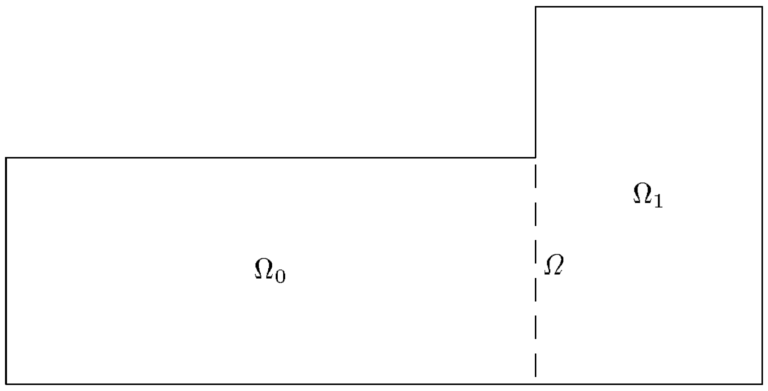

Figure 15: L-Shaped domain.

functions $g(x, y)$. Even with this limited set, running the experiment required about two weeks of dedicated multicomputer time. Although the formal study is limited to a few test functions, we experimented with several other cases in a restricted fashion. No significant variations in the results were observed.

Irregular domains. The original applications of domain decomposition centered on reducing problems on irregular domains to problems on regular domains. This is particularly interesting if the interfaces between the subdomains are small. It is a limitation of our experiment that it does not include this strength of domain decomposition.

Consider, for example, the Poisson problem defined on an L-shaped domain as in Figure 15. The two rectangular subdomains Ω_0 and Ω_1 are separated by a small interface $\mathit{\Omega}$. The capacitance system corresponding to this decomposition will have a small number of unknowns and will be symmetric positive-definite. Domain decomposition is, therefore, an appropriate solution method for this problem. One could even consider solving the subproblems in different processes. However, this domain decomposition can use at most two processes, and its maximum possible speed-up is two. In fact, the speed-up might be significantly less because of possible load imbalance between Ω_0 and Ω_1.

To achieve a higher level of concurrency, one must use a concurrent solver for the two subdomain problems. Our experiment shows that one should not further subdivide the rectangular subdomains. Instead, one could use concurrent multigrid as a subdomain solver for a domain-decomposition method defined over the two regular subdomains.

Conclusion. Expanding the application range beyond the regular Poisson problem requires considerable effort for any method. However, the obstacles seem significantly greater for domain decomposition.

8.2 The Computer

All timings were obtained on the Symult S2010 [8]. This multicomputer is a collection of 168 Motorola 68020 processors connected in a rectangular communication network with worm-hole routing. In many concurrent-computing circles, this computer is classified as antique furniture. However, it does have advantages. Our main motivation for using this system is access. At Caltech, the Symult is used mostly for educational purposes, and its load is very light when not being used by classes. In spite of its age and the demise of the company that built this computer, the Symult system software is the most stable and error-free of any multicomputer available. As a result, this system is a friendly environment for performing an elaborate computational experiment. Although the Symult is too slow for many applications, it is sufficiently fast for our experiment. Experiments with three-dimensional problems would require a faster platform, however. Our main concern should be whether Symult performance is relevant for current multicomputers and multiprocessors.

Vectorization. Most current computers are designed around vector processors; the Symult is based on scalar processors. One could argue that this slants the experiment in favor of multigrid. Because domain decomposition performs more arithmetic between successive communication calls, it will use vector processors more effectively than multigrid. This advantage does not show in our experiment. It is virtually inconceivable, however, that a higher vectorization efficiency could be a significant help in bridging the performance gap.

Communication. Another difference between the Symult and more recent computers is the ratio of communication time over arithmetic time. In this respect, the Symult performs better than most current multicomputers, because communication technology has not kept pace with advancing processor technology. The communication-arithmetic ratio of modern multicomputers is often disappointing, if not at the hardware level then at the user level. The Symult was built with a favorable hardware ratio, and its operating system provides simple low-latency communication primitives that keep the ratio favorable at the user level.

The lower communication overhead of the Symult favors multigrid over domain decomposition, because multigrid performs fewer arithmetic operations between successive communication calls. This slant in favor of multigrid is most relevant for fine-grained computations, when the communication time is the most significant fraction of the execution time. However, none of the tested methods are highly suitable for fine-grained computations, and the advantage in favor of multigrid is a slight one.

Conclusion. Although the choice of computer favors multigrid somewhat, this slant does not threaten the validity of the experiment.

8.3 Computational Parameters

Problem size. Having examined a substantial range of problem sizes, we are confident that our experiment is valid for problems of feasible size.

Number of nodes. Current high-end multicomputers have at least 500 nodes, a few have more than a thousand nodes. Although this is a significant difference with the maximum of 128 nodes used in our experiment, our performance results should translate to computations of similar granularity. It should be noted, however, that neither multigrid nor domain decomposition are well suited for fine-grain concurrency. The coarser levels of multigrid and the ghost boundaries of domain decomposition force a medium- to coarse-grain programming style.

Number of processes. It is standard practice on current multicomputers to choose the number of processes always equal to the number of nodes. The most important argument in favor of using several processes per node is communication hiding: instead of keeping a processor waiting until a message arrives, control is switched to another process ready to perform useful work. It remains an open question whether communication hiding can offset the inefficiencies of process scheduling. However, since multigrid spends a larger fraction of its execution time on communication, it holds better promise than domain decomposition when it comes to exploiting communication hiding.

Process mesh and process placement. Process mesh $(P \times Q)$ and process placement (on which node to place each process) have only a minor impact on multicomputer performance. However, they do have a substantial impact on numerical performance of domain-decomposition methods. We even observed several examples of nonconvergence on highly rectangular process meshes. We avoid these robustness problems by choosing $P \approx Q$.

Conclusion. Our choice of computational parameters does not bias our experiment in favor of any particular method.

8.4 Numerical Variants

The most significant limitation of our experiment is that we test only the most basic version of each numerical method. In this section, we evaluate whether known variants of the basic methods can lead to a different conclusion of the experiment. Of course, no experiment can *prove* the nonexistence of methods that outperform multigrid. For this reason, our experimental set-up will remain useful as a benchmark for new methods.

Coding complexity. The basic multigrid method and the basic domain-decomposition method without overlap contain about an equal number of lines of code in the C programming language. The individual routines are of about the same complexity. The Schwarz-iteration method is, by far, the shortest and the simplest program and is comparable in complexity with the smoothing operator of multigrid. Some of the proposed variations require a substantial coding effort. We do not address the question whether or not coding efforts are worth the potential benefits.

8.4.1 Full Multigrid

Adaptive control. The simplest variants of multigrid adaptively determine the number of smoothing steps and adaptively switch between V, W, and F cycle. Adaptive control does not necessarily increase performance, because the required error estimation can be expensive. Adaptivity of this type is more important for robustness than for performance. In production implementations, all numerical methods require some amount of error estimation and adaptive control. Because of their heuristic nature, we avoid these strategies in this experiment.

Data distribution and duplication. Coarse-level computations are overwhelmed by communication. It is a viable option to duplicate the coarsest levels in every process and to avoid coarse-level data distribution and communication. Consider, for example, the performance of FMG(4) on a 64×64 grid in Figure 2. FMG(4) on 16 nodes is the best performer. Unfortunately, it is impossible to run FMG(4) on 32 nodes because of the coarsest grid. Duplicating the coarsest grid in every process would increase the efficiency of FMG(4) on 8 and 16 nodes. Moreover, this variant of FMG(4) would probably run efficiently on 32 and 64 nodes. If the coarsest grid is sufficiently coarse, as in the above example, the memory overhead of duplication is an acceptable price to pay for the increased performance. Advanced implementations of the same idea could decide on the appropriate amount of duplication and distribution for every level. Substituting duplication for communication is inherently computer dependent, because the decision depends on the ratio of communication time over arithmetic time.

Conclusion. The performance of concurrent multigrid can be increased further by the purely algorithmic technique of substituting duplication for communication.

8.4.2 Domain Decomposition Without Overlap

Adaptive control. The theory requires that one solve the subdomain problems exactly. We violate this theoretical requirement and solve only up to discretization-error accuracy. This reduces the amount of work in the subdomain solver considerably. With inexact subdomain solvers, the conjugate-gradient iteration is applied to a perturbed capacitance system. At best, the iteration converges to a solution of the perturbed system. As shown by a limited number of nonconverging cases, inexact subdomain solvers lead to a loss of robustness. This problem can be avoided either by using exact subdomain solvers based on direct methods or by using better stopping criteria for the iterative subdomain solvers.

At the start of the conjugate-gradient iteration, the values on the subdomain boundaries are inaccurate initial guesses. One might consider it pointless to solve the subdomain problems with this boundary data to high accuracy. As the outer iteration progresses and the boundary values become more accurate, the accuracy of the subdomain solver should increase. However, the intuition leading to such adaptive accuracy-control strategies is difficult to translate into rigorous theory. Adaptive accuracy control changes the perturbation of the capacitance system from one iteration step to the next. Different iteration steps compute, therefore, search directions that belong to slightly different coefficient matrices. This may result in a reduced convergence rate or, at worst, in loss of convergence. Accuracy strategies merely trade off robustness for speed.

Production codes require some adaptive error control to determine the number of outer-iteration steps. It is a mathematical challenge to do this without computing the global solution estimate (which consists not only of values on the subdomain boundaries, but also of values in the subdomain interiors). Unfortunately, it is too expensive to compute the interior values every iteration step. In our experiment, error estimation and adaptive control are omitted, and the interior values are computed only after the outer iteration has converged.

Method	2-D	3-D
Dense LU	$O(N^3)$	$O(N^3)$
Banded LU	$O(N^2)$	$O(N^{2.33})$
Sparse LU	$O(N^{1.5})$	$O(N^2)$
Fast Poisson Solver	$O(N \log N)$	$O(N \log N)$
Jacobi	$O(N^2 \log N)$	$O(N^{1.67} \log N)$
Gauss-Seidel	$O(N^2 \log N)$	$O(N^{1.67} \log N)$
Overrelaxation	$O(N^{1.5} \log N)$	$O(N^{1.33} \log N)$
Conjugate Gradient	$O(N^{1.5} \log N)$	$O(N^{1.33} \log N)$
Preconditioned Conjugate Gradient	$O(N^{1.25} \log N)$	$O(N^{1.17} \log N)$
Multigrid	$O(N \log N)$	$O(N \log N)$
Full Multigrid	$O(N)$	$O(N)$

Table 2: Complexity of candidate subdomain solvers.

Subdomain solvers. Table 2 was adapted from Holst [5]. It lists several possible subdomain solvers and their theoretical complexity when applied to the Poisson equation in two and three dimensions. The number N is the number of unknowns. In the text, we focus on the complexity of two-dimensional solvers.

As noted above, inexact subdomain solvers may reduce convergence and robustness of the method. This is an argument in favor of direct solvers, which solve the subdomain problems to an accuracy near machine precision. However, the cost of just one iteration step is already prohibitively large, as was shown in Figures 5 and 6 and the last line of Table 1. Clearly, we cannot afford to increase the complexity of the subdomain solver. This immediately excludes using dense and banded LU-decomposition.

Among the direct solvers, fast Poisson solvers are the optimal choice, and they have a complexity better than that of optimal overrelaxation. Consider, for example, a fast Poisson solver based on the fast Fourier transform. This solver requires that the dimensions of all subdomain problems be powers of two, unless a complicated general-radix transform is used. More importantly, fast Poisson solvers are only applicable if the domain-decomposed solver is applied to the Poisson problem. This is a tight restriction on the application range of the method. Finally, if solving the Poisson problem is the only goal, one can use a concurrent implementation of the fast Poisson solver for the whole domain.

Sparse LU-decomposition has a larger application range and, for two-dimensional problems, a lower complexity than optimal overrelaxation. However, the order-of-magnitude estimates of the table omit the constants in front of the expressions. For small to moderate values of N, one should expect that the reduction of $O(N^{1.5}\log N)$ to $O(N^{1.5})$ is more than offset by a larger constant. Moreover, the direct solver requires the coefficient matrix explicitly and carries a substantial memory overhead.

The preconditioned conjugate-gradient method can be used to solve the subdomain problems with a reduction of $O(N^{0.25})$ in complexity (compared with overrelaxation). This method carries a substantial memory overhead, because the conjugate-gradient iteration requires at least four fields to store search directions, residuals, etc.

Multigrid as a subdomain solver seems rather pointless: if one uses multigrid on the subdomains, why bother writing a capacitance solver around it? Why not merely distributing multigrid on the global domain? If one persists, a technical difficulty arises. To construct coarse grids, the subproblems must have dimensions that are multiples of a power of two. If this is not the case, subdomain problems must use nonconforming grids, which lead to additional complications. One must also contend with substantial memory overhead for all the levels of multigrid as well as for the ghost boundaries necessary for solving the capacitance system.

All of the above alternatives to overrelaxation with optimal parameter require a substantial coding effort and carry a substantial memory overhead. Moreover, our execution-time plots strongly indicate that one cannot significantly improve performance by choosing a different subdomain solver. Consider, for example, Figure 3 for problems on 128×128 grids. The subdomain problems range in size from 64×128 to 8×16 as the number of nodes is increased from 2 to 128. When the subdomain problems are as small as 8×16, the performance difference between any of the considered alternatives is minimal.

The outer iteration. The number of conjugate-gradient iteration steps can be significantly reduced if one is able to construct a preconditioner for the capacitance matrix. This is a considerable mathematical challenge, particularly if one's ambitions exceed solving the Poisson equation. Moreover, many preconditioners are difficult to integrate into a concurrent program.

However, Figures 5 and 6 and the last line of Table 1 indicate that preconditioners are unlikely to have a significant impact in the present case. Assume that there exists a preconditioner that reduces the number of outer-iteration steps to one. Moreover, assume that this preconditioner does not introduce any computational overhead. The resulting preconditioned method would have an execution time equal to that of one iteration step of our nonpreconditioned method. Multigrid outperforms even this ideal domain-decomposition method by a significant margin.

Conclusion. Although marginal improvements to the tested domain-decomposition program are possible, there is no hope to make it competitive with multigrid.

8.4.3 The Schwarz-Iteration Method

Subdomain solvers. For the same reasons as above, other subdomain solvers cannot significantly improve performance of the Schwarz iteration.

Overlap. Increasing the area of overlap beyond three grid points will improve the convergence rate somewhat. However, the area of overlap also represents a significant duplication of effort between neighboring processes, which may wipe out the convergence-rate improvement.

Multiplicative methods. Multiplicative Schwarz methods achieve faster convergence than the additive Schwarz method of our experiment. Unfortunately, multiplicative methods are less concurrent.

The outer iteration. The number of Schwarz-iteration steps can also be reduced by using preconditioning techniques. Once again, Figures 5 and 6 and Table 1 show that multigrid outperforms one Schwarz-iteration step for a wide range of computations. This is a strong indication that preconditioners will not have a significant impact on the conclusion.

Conclusion. None of these variants are likely to deliver the orders-of-magnitude improvement required to make Schwarz iteration competitive.

9 Conclusion

"Classical" measures like speed-up and efficiency can be quite misleading. If we were to choose the sequential-execution time carelessly, our best-performing methods would have the lowest speed-up and the lowest efficiency. Logarithmic execution-time plots do not depend on a good sequential-execution time. For this reason, they are much more reliable than speed-up and efficiency graphs to analyze the performance of concurrent programs.

We used two fundamentally different techniques to derive a concurrent Poisson solver. Domain decomposition numerically splits the original problem into several subproblems. Data distribution, on the other hand, is a purely algorithmic technique and does not alter any aspect of the numerical method. For a significant range of computational parameters, the tested domain-decomposition methods are several orders of magnitude slower than data-distributed full multigrid. For the Poisson problem, there are no obvious remedies to bridge this performance gap.

Nevertheless, domain decomposition remains an important technique for certain problems, like those defined on irregular domains composed of regular subdomains. However, based on our results for the Poisson solver, concurrency alone does not seem to be a sufficient justification for considering domain decomposition.

Acknowledgments

I thank Dr. Joel N. Franklin, Dr. Michael J. Holst, Dr. Herbert B. Keller, and Dr. Stefan Vandewalle for their critical comments on several early versions of this paper.

I would also like to acknowledge valuable correspondence with Dr. Petter E. Bjørstad, Dr. Paul F. Fischer, Dr. Stephen F. McCormick, Dr. Barry F. Smith, Dr. Olof B. Widlund, and Dr. David M. Young.

References

[1] P. E. Bjørstad and O. B. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM Journal on Numerical Analysis*, 23:1097–1120, 1986.

[2] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31:333–390, 1977.

[3] T. F. Chan, R. Glowinski, J. Periaux, and O. B. Widlund, editors. *Domain Decomposition Methods for Partial Differential Equations.* SIAM, 1990.

[4] R. Glowinski, G. Golub, G. Meurant, and J. Periaux, editors. *Proceedings of the First International Symposium on Domain Decomposition Methods for Partial Differential Equations.* SIAM, 1988.

[5] M. J. Holst. *The Poisson-Boltzman Equation.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[6] E. Isaacson and H. B. Keller. *Analysis of Numerical Methods.* John Wiley and Sons, 1966.

[7] H. A. Schwarz. Ueber einige Abbildungsaufgaben. *Journal für die reine und angewandte Mathematik*, 70:105–120, 1869.

[8] C. L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martin, J. Seizovic, C. S. Steele, and W.-K. Su. The architecture and programming of the Ametek series 2010 multicomputer. In G.C. Fox, editor, *Hypercube Concurrent Computers and Applications.* ACM Press, 1988.

[9] E. F. Van de Velde. *Concurrent Scientific Computing.* Number 16 in Texts in Applied Mathematics. Springer-Verlag, 1994.

[10] E. F. Van de Velde. Program documentation for "Domain Decomposition vs. Concurrent Multigrid". Report CRPC-94-11a, California Institute of Technology, 1994.

[11] P. Wesseling. *An Introduction to Multigrid Methods.* John Wiley and Sons, 1992.

[12] D. M. Young. *Iterative Solution of Large Linear Systems.* Academic Press, 1971.