Parallel Implementation of a Multigrid Method on the Experimental ICAP Supercomputer

Raphaele Herbin*

IBM Corporation
Department 48B, MS 428
Neighborhood Road
Kingston, New York 12401

Stephane Gerbi†

Ecole Nationale des Travaux Publics de l'Etat
Rue Maurice Audin
69120, Vaulx en Velin, France

and

Vijay Sonnad

IBM Corporation
Department 48B, MS 428
Neighborhood Road
Kingston, New York 12401

Transmitted by John Casti

ABSTRACT

A parallel multigrid method for the resolution of elliptic partial differential equations has been implemented on the loosely Coupled Array of Processors system at IBM Kingston, an experimental MIMD machine with communications occurring via shared memories. We consider various smoothers, restrictions, and prolongations, using multicolor ordering of the grid points. The parallelization is obtained by an *a priori* decomposition of the finest grid; each step of the multigrid method is then

*Present address: Ecole Polytechnique Federale de Lausanne, GASOV/ASTRID, Lausanne, 1015, Switzerland.
†Present address: Yale University, Computer Science Department, Box 2158, Yale Station, New Haven, Conn. 06520.

52 Vanderbilt Ave., New York, NY 10017 0096-3003/88/$03.50

parallelized, up to the direct solve on the coarse grid, which is done sequentially; data communication via the shared memory occurs at the boundaries of the subregions thus defined. We present the numerical results that we obtain for the parallel code, along with the efficiencies according to the number of processors that we use. A finer analysis of the parallel code is carried out by a precise timing of the various procedures, leading to a number of conclusions as to the optimal way of parallelizing the multigrid method with this kind of approach.

I. INTRODUCTION

The solution of very large problems, such as 3D fluid-dynamics simulations, generally requires more computing power than a standard sequential computer can provide. Parallel computing appears to be a way of circumventing this limitation. Parallelizing efficient sequential numerical algorithms, however, is not always possible and does not automatically lead to the best parallel algorithm.

A very fast sequential method is the multigrid method. The convergence speed does not deteriorate when the discretization is refined, whereas classical iterative methods slow down with finer mesh discretization. Our purpose here is to find out whether this very fast sequential method can be efficiently implemented on a parallel supercomputer such as the experimental ICAP system.

The parallel implementation of multigrid methods has been studied by several authors, for different architectures (see [22] for a survey). In a multigrid code, the unknowns arise from a sequence of nested grids, which can create some difficulties for data communication in a mesh-connected array of processors. In [6], Brandt studies the parallelization of the multigrid method with one or more points of a particular grid assigned to one processor, for a perfectly shuffled nearest-neighbor array. Chan and Schreiber [9] consider large networks of simple processors with local communication. In these parallel codes, processors are assigned to one grid only, and thus remain idle during the computation on other grids. In [13] and [14], new versions of multigrid are introduced in order to allow simultaneous relaxation on all grids. We also mention recent work by Frederickson and McBryan [27].

Our approach here is to divide the domain of definition into as many subregions as the number of processors (up to 10 for the ICAP1 system), thus dividing all grids between processors. Data communication occurs at the boundaries of each subregion from one processor to its neighbor and is done via the shared bulk memory (see [10] for a detailed description of the ICAP system). Therefore, no processor is idle at any given step of our parallel

multigrid algorithm. This approach was previously taken for a parallel implementation of PDE solvers on the Denelcor HEP-1 [19] and on hypercubes [20].

One of the main features of the multigrid algorithm is that it requires a "smoothing" step on each grid. The smoothing step consists of one or more iterations of a linear iterative method (see e.g. [16]). This is the reason why our starting point is the study of the parallelization of a linear iterative solver. This also explains why we have chosen to parallelize the SOR method with red-black ordering: it is indeed, as we shall see, very efficient as a smoother for the multigrid method.

In Section II, we present the basic multigrid concept and give the fundamental algorithm; various possible smoothers are also presented. The convergence results and the computational cost are outlined. In Section III, we present some sequential tests that were carried out before parallelization; the experimental ICAP system is described; the parallel implementation of the multigrid method and the data flow analysis are studied. Parallel numerical results are finally presented in Section IV; a precise analysis of the parallel efficiency of each of the components of the method is performed, leading to a number of conclusions as to the parallelization of the method.

II. THE MULTIGRID METHOD

II.1. Problem Definition and Notation

II.1.1. The Continuous Boundary-Value Problem. We consider here the following second-order, positive definite self-adjoint Dirichlet problem:

$$\begin{aligned} Lu &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{II.1}$$

where

$$Lu = -\sum_{i,j=1}^{2} \frac{\partial}{\partial x_i}\left(a_{ij}\frac{\partial u}{\partial x_j}\right) \tag{II.2}$$

and Ω is a polygonal domain of $\mathbb{R}^2$.

II.1.2. Finite-Difference Discretization. Let h be a real number. We introduce L_h, the discrete operator obtained by approximating L by the

second-order Taylor's formula (5-point difference scheme) and the following discrete problem:

$$\begin{aligned} L_h u_h &= f_h \qquad \text{in } \Omega_h, \\ u_h &= 0 \qquad \text{on } \partial\Omega_h. \end{aligned} \tag{II.3}$$

Let us consider as a model problem Poisson's equation with homogeneous Dirichlet boundary condition on the unit square:

$$\begin{aligned} -\Delta u &= f \qquad \text{in } \Omega =]0,1[\times]0,1[, \\ u &= 0 \qquad \text{on } \partial\Omega. \end{aligned} \tag{II.4}$$

The 5-point difference scheme leads to Equation (II.3), where the spatial stencil can be represented as

$$L_h = -\Delta_h = \frac{1}{h^2}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}. \tag{II.5}$$

Eliminating the discrete boundary conditions leads to a 4-point scheme at the edges and to a 3-point scheme at the corners.

II.2. The Multigrid Algorithm

For the multigrid approach, one needs a sequence of grids (discretized domains) numbered from l_0 (coarse grid) to l (fine grid): $\Omega_{l_0}, \ldots, \Omega_l$, with respective mesh sizes $h_{l_0}, \ldots, h_l$, where $h_i = (\frac{1}{2})^{i+1}$. So we simply write

$$L_l u_l = f_l \qquad \text{in } \Omega_l, \tag{II.6}$$

where L_l is the discrete operator taking into account the boundary conditions.

The multigrid method can easily be described by a recursive program [16, 23], given below, where S_l^ν denotes ν iterations of the smoother, which is a linear iterative method; r is the restriction from a grid to the next coarser one; d is the defect on this new grid (and thus the right-hand side for the next smoothing step); and p is the prolongation from a grid to the next finer one.

procedure MGM (l, u, f): integer l; array u, f;
comment: l level of the fine grid, l_0 level of the coarse grid, $u = u_l^j$ is a given iterate, $u = u_l^{j+1}$ is the result.
begin
if $l = l_0$ then
$u = L_{l_0}^{-1} f$
else
begin integer j; array v, d;
$u = S_l^{\nu_1}(u, f)$
$d = r(f - L_l u)$
$v = 0$
for $j = 1$ to γ step 1 do MGM$(l-1, v, d)$
$u = u + pv$
$u = S_l^{\nu_2}(u, f)$
end
endif
end

Several restriction and prolongation operators have been proposed, according to the type of equation and of discretization (see e.g. [16, 23, 5] for a discretization with finite differences, and [16, 4, 18] for the finite-element approach). For simplicity, we have restricted ourselves to the case of finite differences, and we consider the 9- and 7-point prolongation operators and the 9-, 7-, and 5-point restriction operators [26, 12].

We shall see in the next subsection that if red-black relaxation is used as a smoother, the defect, $d_l = f_l - L_l u_l$, vanishes at all black points. Thus, if the defect $d_{l-1} = r d_l$ is computed by the 5-point restriction, it coincides with $\frac{1}{2} r_{\text{inj}} d_l$, where r_{inj} is the trivial restriction between Ω_l and Ω_{l-1}. That is the reason why, in this case, it is also called the half-injection. Note that this type of restriction is a very good candidate for parallelization, since it is a completely local process.

II.3. Smoothers and Smoothing Properties

The well-known damped Jacobi method is generally considered to be a very easily parallelizable method, since within one iteration, each point is computed independently from the others. The method may thus be parallelized quite simply (see e.g. [22]). The convergence rate of the Jacobi method, however, is not nearly as good as the SOR method, and in [1], for instance, its parallelization is only considered for use as a preconditioner in a parallel preconditioned conjugate-gradient method. However, we are not interested in the convergence rate of the methods but in their smoothing properties. The SOR method is a good candidate as a smoother, but it is inherently sequential.

43_{red}	44_{black}	45_{red}	46_{black}	47_{red}	48_{black}	49_{red}
36_{black}	37_{red}	38_{black}	39_{red}	40_{black}	41_{red}	42_{black}
29_{red}	30_{black}	31_{red}	32_{black}	33_{red}	34_{black}	35_{red}
22_{black}	23_{red}	24_{black}	25_{red}	26_{black}	27_{red}	28_{black}
15_{red}	16_{black}	17_{red}	18_{black}	19_{red}	20_{black}	21_{red}
8_{black}	9_{red}	10_{black}	11_{red}	12_{black}	13_{red}	14_{black}
1_{red}	2_{black}	3_{red}	4_{black}	5_{red}	6_{black}	7_{red}

FIG. 1. Red-black ordering for $h = \frac{1}{8}$ (49 nodes).

II.3.1. The SOR Method with Red-Black Ordering. We consider here a way of parallelizing SOR by a reordering of the nodes, which leads to independent steps when the system of linear equations arises from the discretization of a partial differential equation. Let us start by considering the model problem (II.4) and its discretization (II.3). We partition the grid points by the classical red-black ordering, as shown in Figure 1. Using the lexicographical (rowwise) ordering and assuming an odd number of nodes, we split Ω_h, the set of grid points, into two sets of nodes: the red nodes with odd numbers, and the black nodes with even numbers. Let N be the total number of nodes $[N = (2^l - 1)^2$ for a grid of level l as defined in Section II.2], and $M = E(N/2)$ be the number of black nodes. It is well known that the 5-point discretization (II.5) of the model problem (II.4) leads to the following partitioned matrix form (see e.g. [25]):

$$\begin{bmatrix} D_r & E \\ E^T & D_b \end{bmatrix} \begin{bmatrix} U_r \\ U_b \end{bmatrix} = \begin{bmatrix} F_r \\ F_b \end{bmatrix}, \tag{II.7}$$

where $U_r = (u_1, u_3, \ldots, u_N)^T$, $U_b = (u_2, u_4, \ldots, u_{N-1})^T$, $F_r = (f_1, f_3, \ldots, f_N)^T$, $F_b = (f_2, f_4, \ldots, f_{N-1})^T$, $D_r = (4/h^2)I_{M+1}$, $D_b = (4/h^2)I_M$, and E is an $(M+1) \times M$ matrix.

The Gauss-Seidel iteration on the system (II.7) leads to

$$\begin{aligned} D_r U_r^{k+1} &= -EU_b^k + F_r, \\ D_b U_b^{k+1} &= -E^T U_r^{k+1} + F_b. \end{aligned} \tag{II.8}$$

The system (II.8), which defines the red-black relaxation, can be implemented in parallel, since the red and black unknowns are completely decoupled. Of course, like the Gauss-Seidel method, the red-black relaxation method can be accelerated by a relaxation parameter ω.

Note that if the partial differential equation to be solved involves cross derivatives, then the matrices D_r and D_b are no longer diagonal, so that the unknowns U_r and U_b are now coupled. In this case, our method can easily be generalized by using multicolor schemes which have been proposed for the 9-point and 13-point finite-difference schemes, as well as for some finite-element schemes [1, 2, 21].

II.3.2. Smoothing Properties of Linear Iterative Methods. We are now going to review the smoothing properties of a linear iterative method and in particular the damped Jacobi method, the SOR method, and finally the red-black relaxation.

The coarse-grid correction step as an iteration by itself is nonconvergent; therefore, the crucial step when developing multigrid solvers is the design of linear iterative methods with a high error-smoothing rate. Namely, the question is how to reduce nonsmooth error components for as little computational work as possible. In the case of the model problems, the accuracy of the methods can be measured precisely by local mode analysis [5, 23]: for the 5-point discretization (II.5) of the model problem (II.4), the algebraic error $u_h - \tilde{u}_h$ (where u_h is the exact solution to the discrete problem and $\tilde{u}_h$ is the approximation computed by one step of the multigrid method) is a combination of Fourier components:

$$\exp\left(i\theta_1 \frac{x}{h}\right)\exp\left(i\theta_2 \frac{y}{h}\right)$$

for $$(\theta_1, \theta_2) \in]-\pi, \pi] \times]-\pi, \pi], \quad (x, y) \in \Omega_h.$$

These components can be separated into low and high frequencies; the low-frequency Fourier components on the grid Ω_h are also the Fourier components on the grid Ω_{2h}. The smoothing factor on the grid Ω_h is defined by $\mu(h) = \max\{\mu(\theta_1, \theta_2, h)\}$, for (θ_1, θ_2) defining a high frequency, where $\mu(\theta_1, \theta_2, h)$ is the factor by which the amplitude of the Fourier component

$\exp(i\theta_1 x/h)\exp(i\theta_2 y/h)$ is multiplied by a relaxation sweep. This is the bound for the factor by which all high-frequency error components are reduced, and of course we want $\mu(h) < 1$. In [23], Stüben and Trottenberg define also the uniform smoothing factor as

$$\mu^* = \max\{\mu(h),\ h \leqslant \tfrac{1}{4}\}.$$

For appropriate methods, smoothing factors are smaller than 0.5 (see e.g. [16, 5, 23]).

In Section 3 of [23], Stüben and Trottenberg show that the smoothing factor for the damped Jacobi iteration is $\mu^*_J = \max\{|1 - \omega/2|,\ |1 - 2\omega|\}$, so that the optimal damping factor is $\omega = \frac{4}{5}$ and the optimal smoothing factor is $\mu^*_J = 0.6$.

In the case of the SOR method with lexicographical ordering, the eigenfunctions of the iteration matrix are not known, thus preventing the use of the Fourier mode analysis. However, using local mode analysis, Brandt (in [5]) shows that, for the generally used relaxation parameter $\omega = 1$, i.e. the Gauss-Seidel method, the optimal smoothing factor $\mu^*_{\text{SOR-L}}$ is equal to 0.5; the Gauss-Seidel method is therefore a better smoother than the damped Jacobi method.

Finally, in [23] it is shown that the smoothing factor of the SOR method with red-black ordering (SOR-RB) is optimal for $\omega = 1$ (i.e. the Gauss-Seidel method), and equal to $\mu^*_{\text{SOR-RB}} = 0.250$; it is therefore better than the two previous ones. Moreover, the use of the SOR-RB method as a smoother considerably reduces the computational cost of the MGM procedure, for the following two reasons:

(i) It is well known that at each iteration of the red-black relaxation the defect $f_l - L_l u_l$ vanishes at all the black points; thus in the multigrid method we compute it only at the red points. Therefore, the computation of $r(L_l u_l - f_l)$ requires less computational work than in the general case.

(ii) If we apply postsmoothing iterations, since the red-black relaxation requires only the values of u^k at the black points, we can compute $u_l + pv_{l-1}$ only at the black points, again reducing the computational work.

Additionally, we note that since the iteration MGM at levels $k < l$ is started with initial guess $u_k = 0$, and since the smoothing steps are linear, the first presmoothing step $S(u_k, f_k) = S(0, f_k)$ can be performed with a reduced number of iterations.

These properties and the fact that the red-black relaxation can be easily parallelized have led us to use the red-black relaxation as the smoother of our parallel multigrid method.

II.4. Convergence Speed and Operation Count

Before we turn to the numerical implementation, we briefly outline the convergence properties of the multigrid method. When the continuous problem is discretized by a finite-difference scheme, the convergence of multigrid methods using various smoothers, restrictions, and prolongations is proved by a local mode analysis. Stüben and Trottenberg [23], Brandt [5], and Hackbusch [16] have already proved a number of convergence results. We outline the principal ones.

(1) The convergence of a multigrid method depends on the smoothing property of the smoother used (the smaller μ^* is, the better the convergence is), and on the coarse-grid correction operator (involving the restriction r and the prolongation p).

(2) The energy norm of the multigrid iteration matrix can be bounded independently of l and thus of h, but depends only on $\nu = \nu_1 + \nu_2$. This result is important because multigrid methods are the only methods for which the convergence bound does not depend on h.

(3) The Euclidean norm of the multigrid iteration matrix is also bounded independently of h, but depends on the couple (ν_1, ν_2).

The convergence of multigrid methods in the case of discretization by a finite-element method has also been investigated. For more details about this case, we refer to [4, 18].

Let us now turn to the computational work that is required in the multigrid method. Typically, the number of arithmetic operations needed for one multigrid iteration is proportional to the number of grid points of the finest grid, N_l. In this sense, the multigrid method can be looked upon as the optimal method for the resolution of a linear system. The constant of proportionality depends on the type of cycle, i.e. γ, the type of coarsening, and the multigrid components: smoother, restriction, and prolongation.

Stüben and Trottenberg [23] and Hackbusch [16] showed that the computational work W may be expressed as

$$W \sim \frac{1}{1-\eta}(\nu C_s + C_d + C_p)N_l,$$

where

C_s, C_d, and C_p are respectively the computational work by grid points on the grid Ω_k of the pre- and postsmoothing steps, the computation of the

defect and its restriction, the prolongation of the correction, and its addition to the previous approximation;

$\eta = \gamma / c_h$, where c_h expresses the type of coarsening, i.e. satisfies $N_k \sim c_h N_{k-1}$; for a two-dimensional problem with standard coarsening, $c_h = 4$.

Let us now present the computational work needed in the particular case where red-black relaxation is used as the smoother, the restriction is the 5-point operator, and the prolongation is the 7-point operator. Using properties (i) and (ii) of Section II.3.2, we obtain

$$C_d = \begin{cases} \times: & 0.5, \\ +/-: & 1.25, \end{cases}$$

$$C_p = \begin{cases} \times: & 0.5, \\ +/-: & 1.0, \end{cases}$$

$$C_s = \begin{cases} \times: & 1.0, \\ +/-: & 3.0. \end{cases}$$

So we finally obtain, for the V-cycle, i.e. $\gamma = 1$, and the standard coarsening of the model problem (II.4),

$$W \sim \tfrac{4}{3} N_l \begin{cases} \times: & (\nu_1 + \nu_2) + 1, \\ +/-: & 3(\nu_1 + \nu_2) + 2.25. \end{cases}$$

III. NUMERICAL IMPLEMENTATION OF THE PARALLEL MULTIGRID METHOD

III.1. Sequential Tests: Choice of Operators and Parameters

We first implemented sequentially five different multigrid methods to solve the discrete model problem (II.3)–(II.5), which are all of the V-cycle type, i.e. $\gamma = 1$, in order to choose the optimal operators (smoother, restriction, and prolongation) as well as the optimal parameters ν_1, ν_2 in our parallel multigrid method; we recall that the most easily parallelizable smoothers are the damped Jacobi method and the red-black relaxation. We present the different methods by their smoother, their prolongation, and their restriction.

MG.JACOBI: The smoother is the damped Jacobi method with the optimal relaxation parameter $\omega = 0.8$; the restriction and the prolongation are the well-known 9-point schemes.

MG.GAUSS-SEIDEL: The smoother is the Gauss-Seidel method with lexicographical ordering; the restriction and the prolongation are the same as above.

MG.RB 9: The smoother is the red-black relaxation; the restriction and the prolongation are the same as above.

MG.RB 7: The smoother is the same as above; the restriction and the prolongation are the 7-point schemes.

MG.RB 5: The smoother is the same as above, the restriction is the 5-point one (i.e. the half-injection), and the prolongation is the 7-point one.

All these algorithms are tested with the right-hand side $f(x, y) = 2[x(1-x) + y(1-y)]$, so that the exact solution of Equations (II.3) and (II.4) is $u(x, y) = x(1-x)y(1-y)$. Thus, we can take as the convergence criterion

$$\|u_l^i - u\| \leqslant \varepsilon, \tag{III.1}$$

where u_l^i is the approximation and u is the exact solution. To compare the different algorithms we chose to compute the reduction factor

$$\tau_i = \frac{\|u_l^{i+1} - u\|}{\|u_l^i - u\|},$$

where $\|\cdot\|$ denotes the Euclidean norm. The average reduction factor is therefore

$$\bar{\tau} = \left(\prod_{i=1}^{\text{iter}} \tau_i \right)^{1/\text{iter}} \tag{III.2}$$

where ITER is the number of iterations needed to satisfy (III.1). We used subroutines from well-known packages [24, 11] for the direct solve and various linear algebra procedures. In the next sections, we shall see the influence of the smoother and the influence of the restriction and prolongation.

III.1.1. Influence of the Smoother. In previous sections we saw that the red-black relaxation has a very good smoothing factor, better than the Gauss-Seidel method and the optimal damped Jacobi method. In order to

TABLE 1
ITERATION NUMBER AND AVERAGE RATIO FOR $l = 7$, $N_l = 16{,}129$, $\varepsilon = 10^{-6}$, FOR THE SOLUTION OF THE MODEL PROBLEM (II.4)

$(\nu_1, \nu_2) =$	(1,1)		(2,1)		(5,1)	
Method	$\bar{\tau}$	ITER	$\bar{\tau}$	ITER	$\bar{\tau}$	ITER
MG.JACOBI	0.369	15	0.265	11	0.124	7
MG.GAUSS-SEIDEL	0.205	10	0.106	7	0.056	4
MG.RB 9	0.140	5	0.079	3	0.031	2

compare the influence of the smoother, we implemented the same restriction and prolongation in each program, and we chose the 9-point schemes. The results are shown in Table 1.

Let us now turn to the influence of the number of pre- and postsmoothing steps on the efficiency of the method when red-black relaxation is used. We always used at least one presmoothing and one postsmoothing step so as to be able to lower the cost of intergrid operations, as pointed out in remarks (i) and (ii) of Section II.3.2. For the model problem (Poisson equation), the best performance was obtained for MG.RB 5 with $(\nu_1, \nu_2) = (1,1)$, leading to 5 iterations and an average reduction factor of 0.063 for a fine grid of 128×128 nodes and a coarse grid of 16×16 nodes. The execution time in this case is 1.71 second. If ν_1 or ν_2 are increased, the iteration number and the average reduction factor decrease, but the additional smoothing steps lead to an increase in execution time. For instance, with $(\nu_1, \nu_2) = (2,1)$, the number of iterations to convergence is 4, with an average reduction factor of 0.031, but the execution time is now 2.16 seconds.

In the case of an equation with variable coefficients, however, this may no longer be true. We took as a model problem the following equation: $e^x \partial^2 u/\partial x^2 + e^y \partial^2 u/\partial y^2 = f$ on the unit square, with homogeneous Dirichlet boundary conditions. The linear system which arises after discretization of this equation has a much wider spectrum, and with $(\nu_1, \nu_2) = (1,1)$, the number of iterations to convergence is now 13, with an average reduction factor of 0.498; the execution time is in this case 5.49 seconds. If we now take $(\nu_1, \nu_2) = (2,2)$, the number of iterations to convergence goes down to 3, with an average reduction factor of 0.021; the execution time decreases to 1.97 second. If additional smoothing steps are performed, the number of iterations and the average reduction factor decrease slowly, but the execution time increases, so that the value $(\nu_1, \nu_2) = (2,2)$ seems to be optimal for this particular equation.

These two examples illustrate the fact that if ν becomes large ($\nu > 6$), the ratio $\bar{\tau}$ does not improve significantly. In [23, Theorem 8.1], Stüben and

Trottenberg show that typically $\rho^* \sim \text{constant}/\nu$ when $\nu \to \infty$, where ρ^* is the asymptotic spectral radius of the multigrid iteration matrix, when $h \to 0$. Thus, it is useless to use large values of ν, because it only increases the computational work.

III.1.2. Influence of Restrictions and Prolongations. We have seen that the convergence results depend on the restriction and the prolongation used. In [16], Hackbusch shows that if

$$L_{l-1} = rL_l p,$$

$$L_l = L_l^*,$$

$$r = p^*,$$

(where * denotes the Euclidean adjoint of a linear operator), the convergence is better. One can easily prove that these assumptions are satisfied only for the program MG.RB 7. Figure 2 illustrates this result.

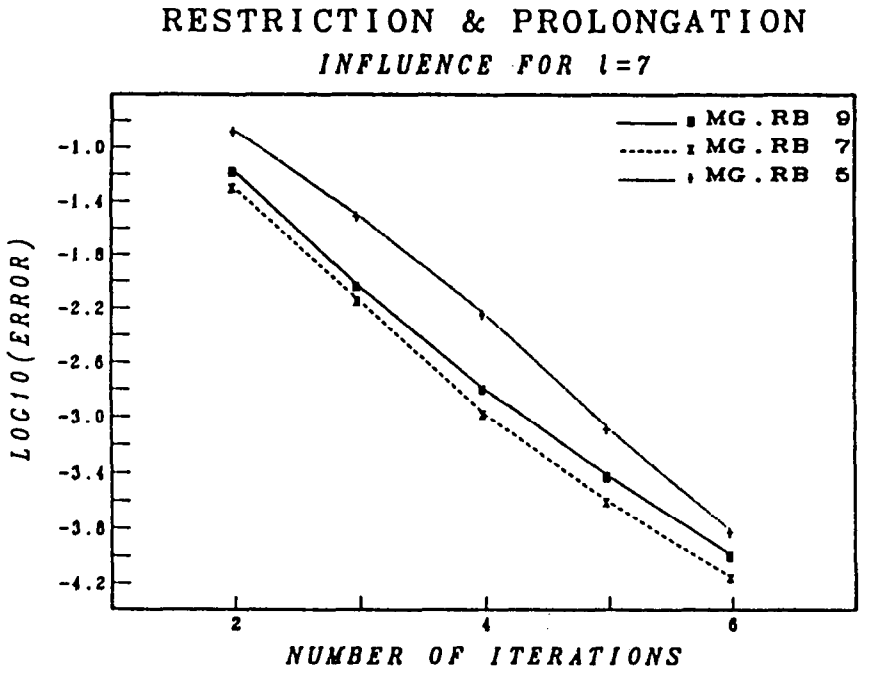

FIG. 2. Influence of the type of prolongation and restriction on the convergence speed for the solution of the model problem (II.4).

III.2. Description of the Parallel System lCAP1

The loosely coupled array of processors system (lCAP1) is an experimental multiprocessor architecture designed by Dr. E. Clementi et al. at IBM Kingston (New York) [10]. Built around IBM machines hosting 10 FPS-164 (Floating Point System) attached processors, this system is a very versatile tool for experimenting with parallel processing in a realistic environment.

III.2.1. Description of the Hardware. Each of the FPS-164 attached processors (APs) has 8 Mbyte of main memory and 0.5 Gbyte of disk storage. The peak performance of one FPS-164 processor is 55 MFLOPS with special hardware. The FPS-164 processors connect to the IBM mainframe via the standard IBM 3-Mbyte/sec channels. Presently, lCAP1 is front-ended by an IBM 3081, supplemented by IBM 4381 and IBM 4341 systems. All three systems run the VM/SP operating system. Because of the slow transfer rate from AP to host and back, shared bulk memories were added for faster communication between processors. The shared bulk memories were designed by Scientific Computing Associates, Inc. (SCA, New Haven, Conn.). Five of these memories are 32 Mbyte large and attach to four processors; the sixth one is 512 Mbyte and can attach to twelve processors. A second addition to the primitive lCAP configuration was a fast bus, the FPSBUS, connecting all ten of the FPS machines. The FPSBUS has the capability to transfer data at a rate of 32 Mbyte/sec along the bus, and 22 Mbyte/sec from the bus to the FPS node. The present configuration of lCAP1 is shown in Figure 3. We mention that a second system, lCAP2, has also been constructed at IBM Kingston. This system is very similar to lCAP1, except that the mainframe is now the dyadic IBM 3084, running the MVS operating system, and the attached processors are FPS-264.

III.2.2. Description of the Communication Software. Software provided and maintained by FPS is actually responsible for communication between the host and the APs; the same is true with SCA for communication between APs via the shared memories. A precompiler and a scheduler have been written by the researchers in the Department of Scientific and Engineering Computation at IBM Kingston, in order to provide the users with easy directives which are inserted in the FORTRAN codes (see [7] and [10] for a complete description).

A parallel code typically consists of a FORTRAN program, the master program, which runs on the IBM host, and FORTRAN AP routines, also called slave programs, which are called by the master program to run on one or more APs. The actual configuration information (number of slaves and APs used) is defined in a **COMMON** block which is automatically set up when calling the scheduler. The FORTRAN master program may execute the sequential portion of the code and calls the AP routine when it reaches a parallel

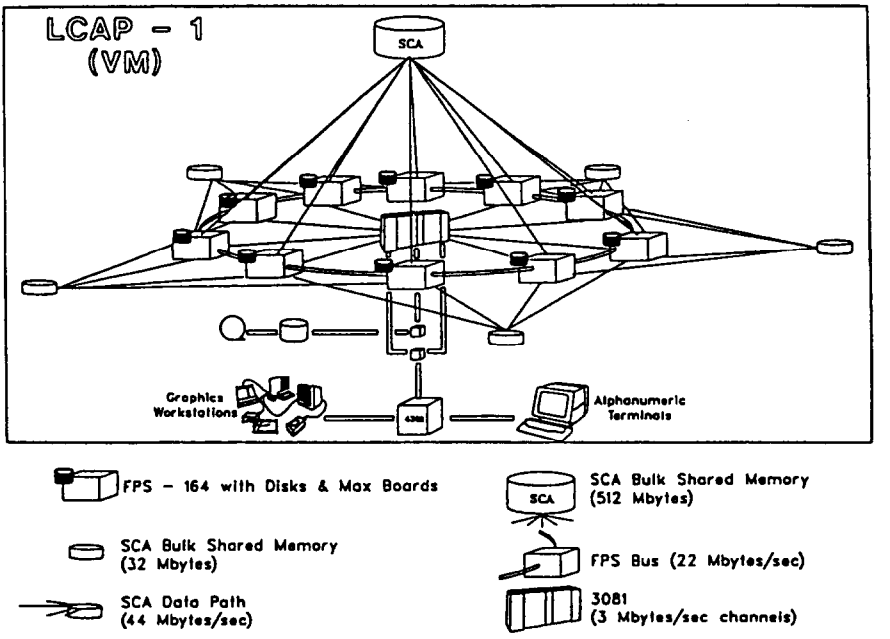

FIG. 3. Configuration of the ICAP system.

portion. In our case, however, the sequential portion is reduced to data input and output, and data transmission between the host and the attached processors.

There are two different ways of programming the AP routines for the transfer of data between APs, both using the shared memory; the chosen mode is defined in the master program right after the **START** directive. One mode is a purely shared-memory mode, i.e., the APs read and write to and from the shared memory, using the **MOVE** directive. We chose this mode in our program because of data-transfer considerations in the algorithm (see following sections). In the AP routine, the arrays of shared memory which are to be used are defined via the directive

```
C$AP SHARED /data structure name/ item1,...itemN
```

Any data transfer is done by instructions of the form

```
C$AP MOVE shared array (dimensions) = local array (dimen -
sions)
```

for writing to the shared memory, and

```
C$AP MOVE local array (dimensions) = shared array (dimen -
sions)
```

for reading from the shared memory. Finally, the

```
C$AP BARRIER
```

directive is used when synchronization between all APs is needed at any point of the parallel run.

The other mode, namely message passing, is used if one wants to send data from one processor to another one in a transparent way, i.e. without taking care of the actual addresses or the synchronization when reading or writing from and into the memory. We refer to [7] for further details.

III.2.3. Communication Cost on the ICAP System. Although the communication operations from processor to shared memory or from shared memory to processor are of $O(\sqrt{N})$, where N is the number of red nodes on the fine level on the whole domain, the communication time itself cannot be neglected in terms of parallel efficiency. On most parallel machines, the transmission time of a vector of length M between two directly connected processors can be expressed as $\alpha + \beta M$, where α is called the latency and β is the transfer rate (see e.g. [15]). On the experimental ICAP system, all processors are directly connected, if the global shared memory is used (see Figure 3). However, the transmission of any information from a processor A to a processor B occurs if (a) processor A writes the information into shared memory and (b) processor B reads the information from the shared memory, when using the shared-memory mode. Note that the message-passing mode differs only in that the synchronization step needed between steps (a) and (b) is handled by the communication software, in a user transparent way. On ICAP, the time taken to transfer information from a processor to the shared memory or the reverse is also of the form $\alpha + \beta M$, where the constants α and β are respectively 360 μs and 0.2 μs; therefore, it is obviously more efficient to transfer long vectors than short ones. However, it is not quite so straightforward to model the actual communication cost by the above formula, since communication instructions typically occur simultaneously on several processors, but the shared memory can only be accessed by four processors at a time. Thus, there is an additional latency time which is difficult to evaluate.

III.3. Parallelization by Decomposition of the Domain

Because of the very high communication costs from host to slave, the best way to implement parallel algorithm on ICAP is to use masterless programming: the host is only used for the input and output of the data, and does not

deal with any intermediate computed data. One way to do this is to have the exact same code on all processors, defining for each of the processors which part of the domain and unknowns are assigned to it. The only data communications are therefore from slave to slave, via one (or more) of the bulk memories.

We consider here, for simplicity, a square domain which is uniformly discretized with $N = 2^l$ intervals on a side, which we divide into P strips of size $N \times (N/P)$. The subregions thus defined are assigned to the P processors ($P = 1$, 2, 4, or 8) that we use by a one-to-one mapping, thus leading to very good load balancing. This type of decomposition was previously used for the implementation of a parallel domain-decomposition method [15, 17]. Three types of subregions (processors) are to be distinguished, as shown on Figure 5 (Section III.3.2):

(1) the bottom one, where the first row is subject to the physical boundary conditions, and information from the next processor will only be needed for the computations done on the nodes of the last row, which is the boundary between subregions 1 and 2;

(2) the intermediate ones, where information is needed from the previous processor for the first row, and from the next processor for the last row;

(3) the last one, where the last row is subject to the physical boundary conditions, and information is only needed from the previous processor for the first row. Note that in a more general case, load balancing would not be so easy to deal with, and that the data structure would have to be carefully studied.

In the following subsection, we describe how we make use of the shared memory for the communications which are required between processors.

III.3.1. Procedures for Local Communication. The communication which takes place between the processors can be classified into two categories: global communication and local communication.

By global communication, we refer to the transmission of data from each processor to every other processor, which occurs in the computation of the dot product (needed for the convergence criterion) and for the direct solve; the data-flow analyses of these steps are presented in the core of the parallel multigrid method (see following section).

By local communication, we mean either the transmission from the shared memory to a specific processor of the boundary data which were computed on neighboring processors, or the transmission by one processor to the shared memory of the data at the boundaries of the subregion it is assigned to. Two procedures were written to take care of local communication, namely **READSM** (read from shared memory) and **WRITSM** (write into shared memory), which

carry out the following instructions:

READSM:

MOVE from shared memory to local memory:

first row of processor 2 for processor 1,
first row of processor ip+1 and last row of processor ip−1 for processor ip, $1 < ip < P$,
last row of processor $P-1$ for processor P;

WRITSM:

MOVE from local memory to shared memory:

last row for processor 1,
first and last rows for processor ip, $1 < ip < P$,
first row for processor P.
BARRIER (synchronize)

Note that a synchronization step is required at the end of **WRITSM**, to prevent a processor from reading from shared memory (i.e. reaching a **CALL READSM**) before another has moved the appropriate data into shared memory.

In the FORTRAN code, the shared memory is represented by an array, where the updated data are being read and written starting at a certain address. For the purpose of local communication, this array is organized as shown in Figure 4, where N is the number of nodes on the horizontal side.

We pointed out in Section III.2.3 that on the ICAP system, one should try to have as few and as long data transfers as possible, which we did by packing vectors together before moving them. Consider a processor ip, with $1 < ip < P$; in **WRITSM**, this processor is going to move a local array of length $2N$ (first and last row, which have been jointed into the local array) into shared memory. Note that if we had used the message-passing mode, this would not have been possible, since one of the vectors needs to be sent to processor ip − 1 and the other one to processor ip + 1. In **READSM**, processor ip has to access two vectors which do not follow one another, i.e., a vector of length N starting at address $2N(ip-2)+1$, and another of the same length starting at address $2N(ip-1)+N+1$. This move cannot be done at once

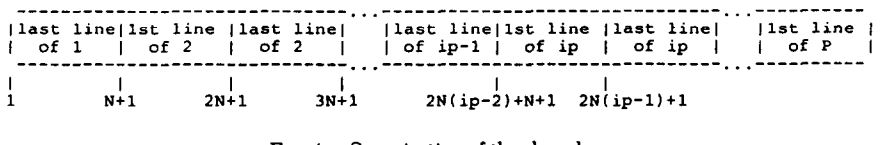

FIG. 4. Organization of the shared memory.

unless the data between the last address of the first vector and the first address of the second one are also transferred from shared memory to local memory, thus doubling the length of the data to be transferred. It turns out that this last operation is preferable to doing two transfers, at least for $N < 2000$, which is always the case in our algorithm. Note that once again, this reduction of communication time could not have been done using the message-passing mode.

In the multigrid method, two global communications are required throughout one iteration: communication of the defect on the coarse grid before the direct solve, and communication of the partial dot products in the convergence criterion.

III.3.2. The Parallel Multigrid Method. We now turn to the data flow for one iteration of the parallel multigrid procedure (MGMP): at the beginning of the iteration, we assume that u is the last-computed approximation to the solution, or the initialization, and that the array in the shared memory is organized as we depicted above, and contains u at the boundaries of each subregion (bottom and top, except for the first and the last subregion). We denote by L and L_0 the levels of the fine and coarse grid, respectively.

I. *From grid* K *to grid* $K-1$, $K = L$ *to* $L_0 + 1$, *step* -1. The descent step consists of:

   A. A smoothing step; the data flow for this step is the following:
      1. **READSM** (if initial guess is not 0)
      2. Computations on all red points of processor `ip`, `ip` $= 1, \ldots, P$
      3. **WRITSM**
      4. **READSM**
      5. Computations on all black points of processor `ip`, `ip` $= 1, \ldots, P$
      6. **WRITSM**

   B. The computation of the Laplacian of the result of the smoother. This operation is done on the red points only, because of the properties of the red-black relaxation method (see Section III). The data flow is the following:
      1. **READSM**
      2. Computations on all red points of processor `ip`, `ip` $= 1, \ldots, P$

      There is no need to write in shared memory after the computation, since the Laplacian is only needed for the computation of the defect, which is a totally local process (pointwise subtraction).

   C. The restriction of the defect from grid K to grid $K-1$. Since the 5-point restriction with red-black ordering is used, this operation boils down to the half-injection, which is obviously a completely local process. Note that the restriction of the defect becomes the

right-hand side for the smoother on the next grid; it is therefore not needed in shared memory.

II. *Direct resolution on grid* L_0. Since the number of points on the coarse grid is small, the direct resolution of the defect equation of the last grid is done sequentially. There are two alternatives for this operation: either pass all data to one processor which does the direct solve and passes the result back to the other processors as needed, or pass all data to each of the processors, solve directly on all processors, and then keep only the data which are needed. We chose to implement the second alternative, because it involves fewer data transfers to and from shared memory than the first one. The data flow for this step is the following:

1. **MOVE** to shared memory the array of the restricted defect, at the adequate address
2. **BARRIER** (synchronize)
3. Read the whole right-hand side in shared memory
4. Do the direct solve on the whole domain on each processor
5. Renumber the array of the solution so as to keep the array of components corresponding to the nodes of this processor only
6. **WRITSM**

III. *From grid* $K-1$ *to grid* K, $K = L_0 + 1$ *to* L. Once the direct solve has been done on the coarse grid, multigrid takes us back to the fine grid by doing, on each grid, a prolongation on the next grid, a correction, and a postsmoothing step. The data flow of the postsmoothing step has already been studied; after the correction step, we need to write the result in shared memory; again, this is performed by calling **WRITSM**.

The only step which we have not studied yet is the prolongation. We have implemented the 7-point prolongation, which was previously used in a vectorized multigrid method for the CDC Cyber 205 computer by Barkai and Brandt [3], and is also very suitable for a parallel implementation. The prolongation procedure is depicted in Figure 5 and is carried out by averaging coarse-grid vectors. First, let us recall that the use of red-black ordering, combined with the fact that a relaxation sweep always follows an interpolation, implies that only the black points among the fine-grid points need to be prolonged, since the red points will be computed from the black ones by the relaxation sweep. We first compute the fine black nodes located horizontally between two coarse red nodes, by doing the half-sum of two column vectors of the coarse grid. This is, of course, a vector operation; moreover, it does not require any information from the neighboring processor, so that no read from shared memory is required. Then, the fine black nodes located vertically between two coarse red nodes are computed by doing the half-sum of two

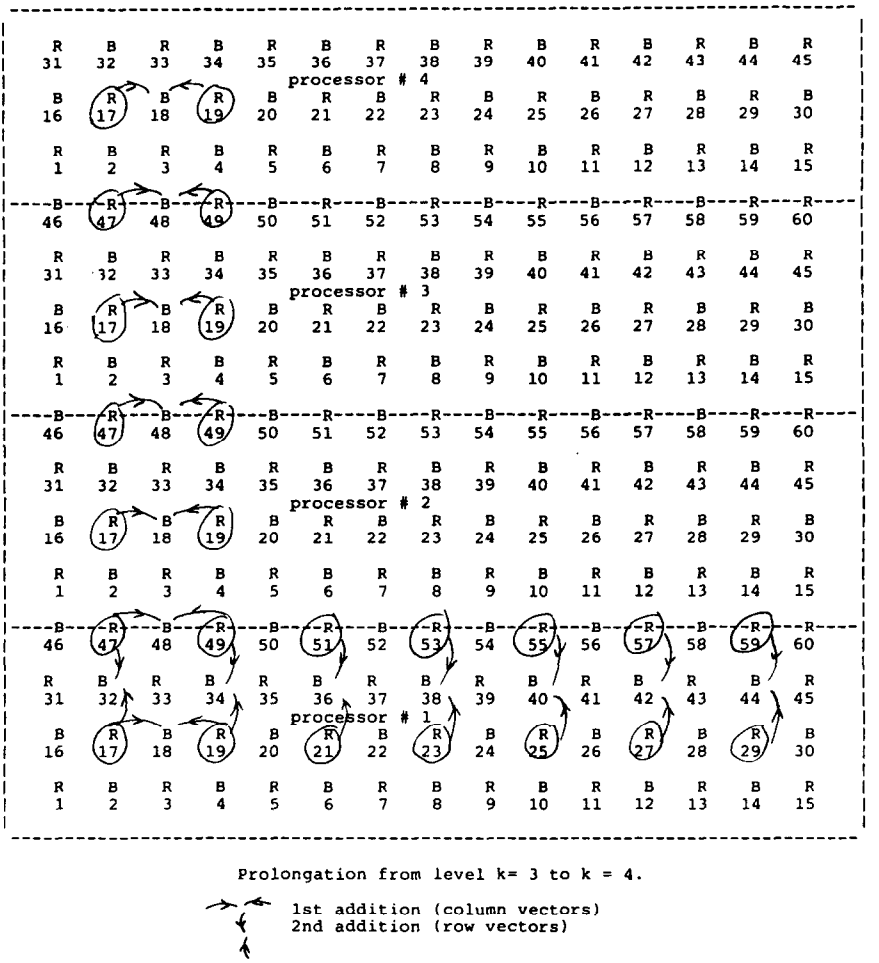

FIG. 5. Example of the grid-point assignment and prolongation for four processors for a fine grid, $l = 4$.

row vectors of the coarse grid. This is again a vector process. However, for the computation of the fine black nodes of the first line, the last row of the coarse red nodes of the previous processors are needed, thus leading to a data transfer from the shared memory (**READSM**). Also, the first row of the first processor and the last row of the last processor have to be computed, taking care of the boundary conditions (the same is true for the first and last columns of each processor). Note that no write into shared memory is needed

after the prolongation, since it is followed by a correction step, which is entirely sequential.

IV. *Data-flow analysis of the convergence criterion.* We now turn to the data flow for the convergence criterion. We recall that we have chosen

$$\|u_1^i - u\| \leqslant \varepsilon,$$

where u_1^i is the approximation and u is the exact solution. The data flow for this step is the following:

1. Do the partial dot product on each processor
2. **MOVE** from local memory to shared memory the partial dot product
3. **BARRIER** (synchronize)
4. Write from shared memory to local memory all the partial dot products
5. Each processor adds the partial dot products

IV. NUMERICAL RESULTS

We have implemented a parallel code for two model equations of the form

$$\begin{aligned} Lu &= g \qquad \text{in } \Omega =]0,1[\times]0,1[\ , \\ u &= 0 \qquad \text{on } \partial\Omega, \end{aligned} \tag{IV.1}$$

where $Lu = -\Delta u$ in the first case, and $Lu = a(x)\,\partial^2 u/\partial x^2 + b(y)\,\partial^2 u/\partial y^2 + [c(x) + d(y)]\,u$ with $a(x)b(y) > 0\ \forall (x,y) \in \Omega$ in the second case. For numerical experiments we took $a(x) = e^x$, $b(y) = e^y$, $c(x) = 0$, $d(y) = 0$. The right-hand side g is taken so that the exact solution is $u(x,y) = x(1-x)y(1-y)$ in both cases.

Note that:

(1) In both cases, we use as the smoother the red-black relaxation method with the 5-point restriction, which is in fact the half-injection and the 7-point prolongation.

(2) In the case where one direction is privileged, it would be more efficient to use an alternating line coloring (see [21]).

After one has obtained a parallel algorithm, it is natural to try to measure its performance in some way. The most commonly accepted measure is the

speedup, which is frequently defined as

$$S_p = \frac{\text{execution time using one processor}}{\text{execution time using } p \text{ processors}}.$$

The strength of this definition is that it uses execution time and thus incorporates any communication or synchronization overhead. A weakness is that it can be misleading to focus on algorithm speedup when in fact one is usually more interested in how much faster a problem can be solved with p processors. Thus, we wish to compare the "best" sequential algorithm with the parallel algorithm under consideration, and we define

$$S'_p = \frac{\text{execution time using the fastest known algorithm on one processor}}{\text{execution time using the parallel algorithm on } p \text{ processors}}.$$

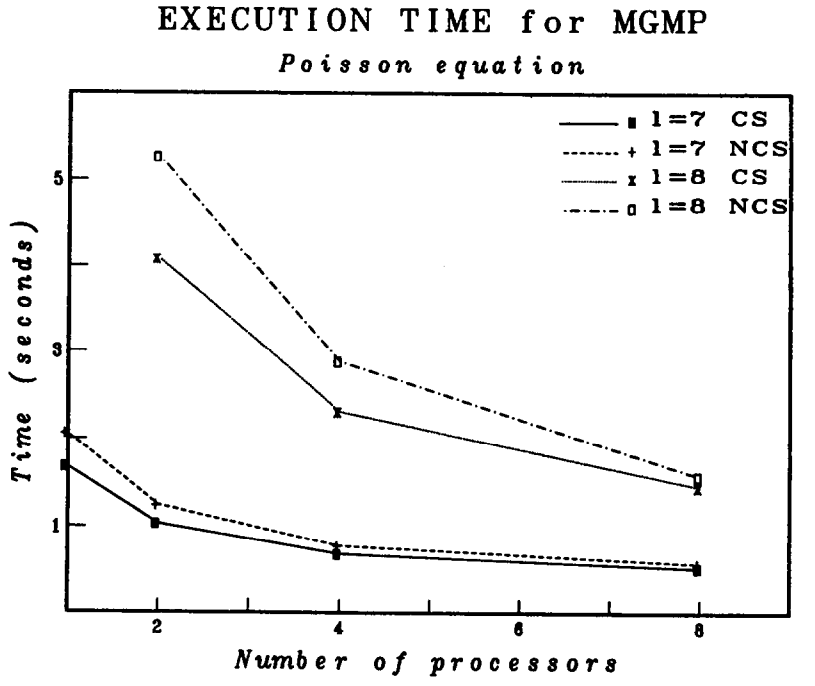

FIG. 6. Execution time of the parallel multigrid code, for the Poisson equation (II.4) with $(\nu_1, \nu_2) = (1,1)$, fine-grid levels $l = 6, 7, 8$ (64×64, 128×128, and 256×256 nodes respectively), coarse-grid level $l_0 = 4$ (16×16 nodes), and error tolerance $\varepsilon = 10^{-6}$. Number of iterations performed: 4 with $l = 6$ or 7, and 5 with $l = 8$. CS: with a fast Poisson solver for the exact coarse solution; NCS: with 10 iterations of the SOR method ($\omega = 1.73$) instead of the exact coarse solution.

This ratio is very difficult to obtain, because one does not know which algorithm is actually the fastest. In our case, however, multigrid methods are known to be among the fastest methods implemented on a serial machine, so that S'_p can be expected to be close to S_p. Finally, we define the efficiency by

$$E_p = \frac{S_p}{p}.$$

Because of memory requirements, when the finest grid is of level 8 or higher, the code has to be run on more than one processor. In such cases, we have measured the speedup and the efficiency in base 2, defined now by

$$S_{p,2} = \frac{\text{execution time using two processors}}{\text{execution time using } p \text{ processors}}, \qquad E_{p,2} = \frac{S_{p,2}}{p/2}.$$

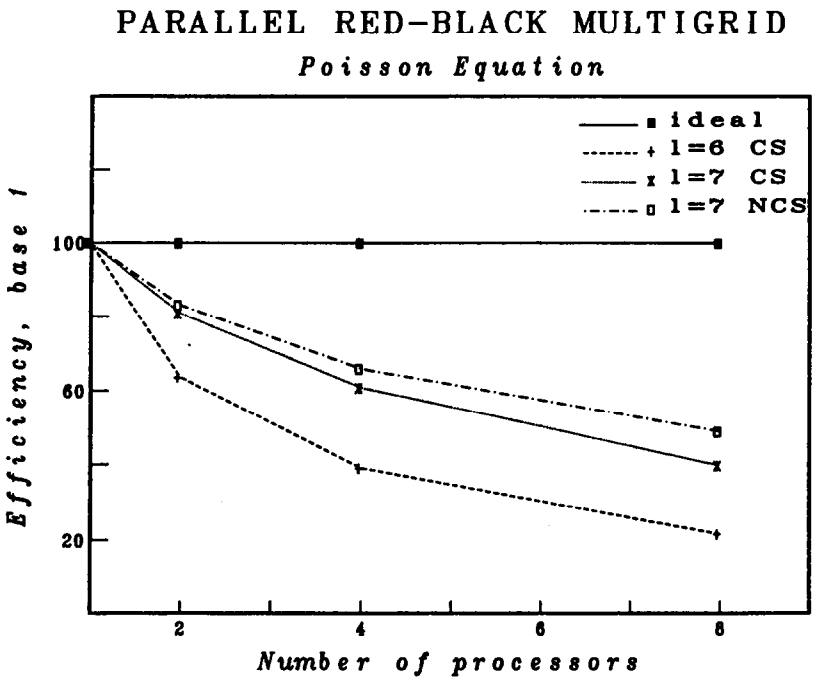

FIG. 7. Efficiency base 1 (i.e., w.r.t. the execution time on one processor) of the parallel multigrid code for the Poisson equation (II.4), with $(\nu_1, \nu_2) = (1,1)$, fine-grid levels $l = 6$, 7, coarse-grid level $l_0 = 4$, and error tolerance $\varepsilon = 10^{-6}$. CS: with a fast Poisson solver for the exact coarse solution; NCS: with 10 iterations of the SOR method ($\omega = 1.73$) instead of the exact coarse solution.

In order to measure the efficiencies, the sequential code was run on one processor, when possible, and the parallel code on 2, 4, and 8 processors. The execution time includes all computation, communication, and synchronization times needed in the multigrid iterations themselves and in the convergence test, but does not include the computation of the matrix coefficients, since we are interested specifically in the parallel efficiency of the multigrid method. Note, however, that including the matrix computation time would lead to even better speedups, since it is an entirely parallel process.

Figure 6 shows the execution times for the Poisson equation. The optimal times were obtained for one presmoothing and one postsmoothing steps, leading to 4 iterations to convergence for fine levels 6 and 7 (64×64 and 128×128 grid nodes respectively), and 5 iterations for fine level 8 (256×256 grid nodes). Although the number of iterations tends to decrease with

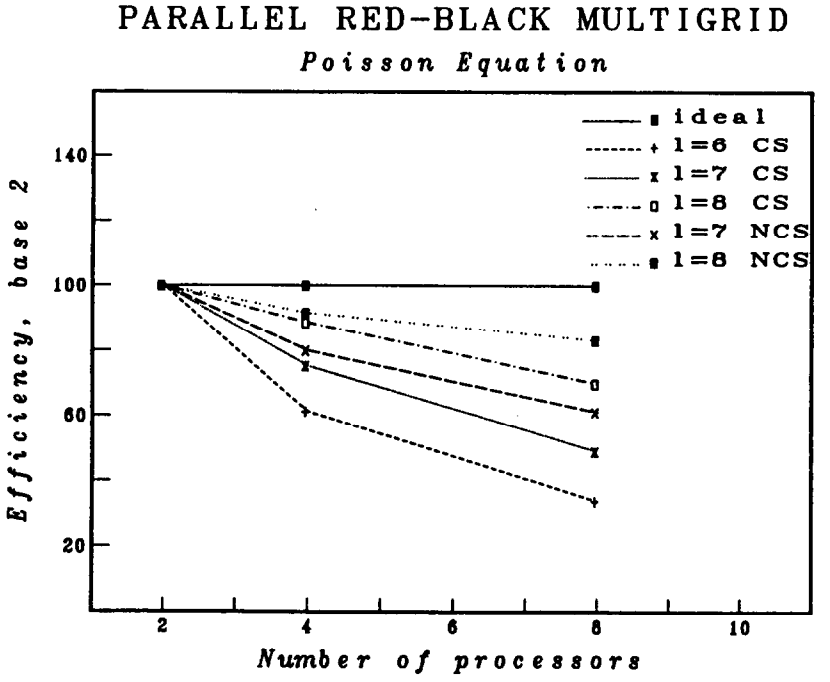

FIG. 8. Efficiency base 2 (i.e., w.r.t. the execution time on two processors) of the parallel multigrid code for the Poisson equation (II.4) with $(\nu_1, \nu_2) = (1, 1)$, fine-grid levels $l = 6, 7, 8$, coarse-grid level $l_0 = 4$, and error tolerance $\varepsilon = 10^{-6}$. CS: with a fast Poisson solver for the exact coarse solution; NCS: with 10 iterations of the SOR method ($\omega = 1.73$) instead of the exact coarse solution.

increasing coarse grid level, the best results were obtained for a coarse-grid level of 4. Higher coarse levels lead to a high coarse-grid resolution cost; on the other hand, lower coarse levels reduce the parallel efficiency because of the small number of nodes (see Section III.2.3).

Although the execution time of the sequential code is excellent, we can see in Figures 7 and 8 that we get good parallel efficiency only for large problems. For smaller problems, the ratio of communication cost to computational cost increases, thus leading to a lower parallel efficiency. Note that the efficiency levels off with an increasing number of processors: the communication routines **READSM** and **WRITSM** take up more time because of the delay in getting access to the shared memory. This raises the difficult issue of programming the interprocessor communications so that the processors do not try to access the shared memory simultaneously.

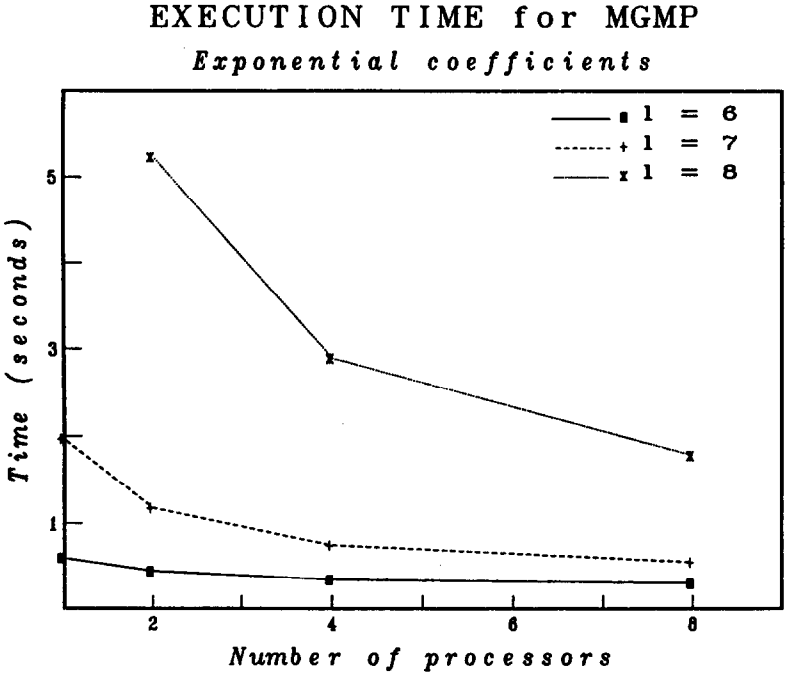

FIG. 9. Execution time of the parallel multigrid code with an exact coarse solution for the exponential coefficient equation (IV.1), with $(\nu_1, \nu_2) = (2,2)$, fine-grid levels $l = 6, 7, 8$, coarse-grid level $l_0 = 4$, and error tolerance $\epsilon = 10^{-6}$. Number of iterations performed: 3 with $l = 6$ or 7, and 4 with $l = 8$.

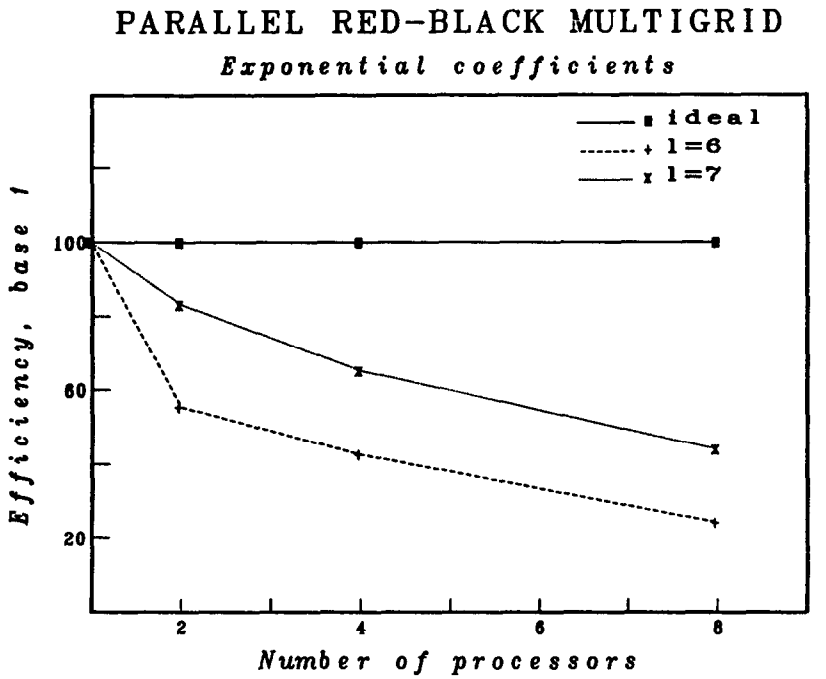

FIG. 10. Efficiency base 1 (i.e., w.r.t. the execution time on one processor) of the parallel multigrid code with an exact coarse solution, for the exponential coefficient equation (IV.1), with $(\nu_1, \nu_2) = (2,2)$, fine-grid levels $l = 6, 7$, coarse-grid level $l_0 = 4$, and error tolerance $\varepsilon = 10^{-6}$.

Figure 9 shows the execution time for the exponential coefficients equation; in these experiments the number of pre- and postsmoothing steps was taken to be $(\nu_1, \nu_2) = (2,2)$, from the sequential tests which were performed previously (see Section III.1.1). The execution times are a little larger than for the Poisson equation. The parallel efficiency, however, is somewhat better (see Figures 10 and 11). This is due to the fact that the ratio of communication time to computation is lower. We also give, in Figure 12, the speedups corresponding to the efficiencies of Figure 11, i.e. for fine levels l and in base 2. Notice that the speedups increase significantly with the size of the problem.

In order to analyze the parallel efficiency of the various steps of the multigrid method, we ran one iteration, with one presmoothing step and one postsmoothing step. We then inserted clocks for each procedure, counting computation, communication, and synchronization time, and summed the timings thus obtained for all grid levels. The results are shown in Table 2.

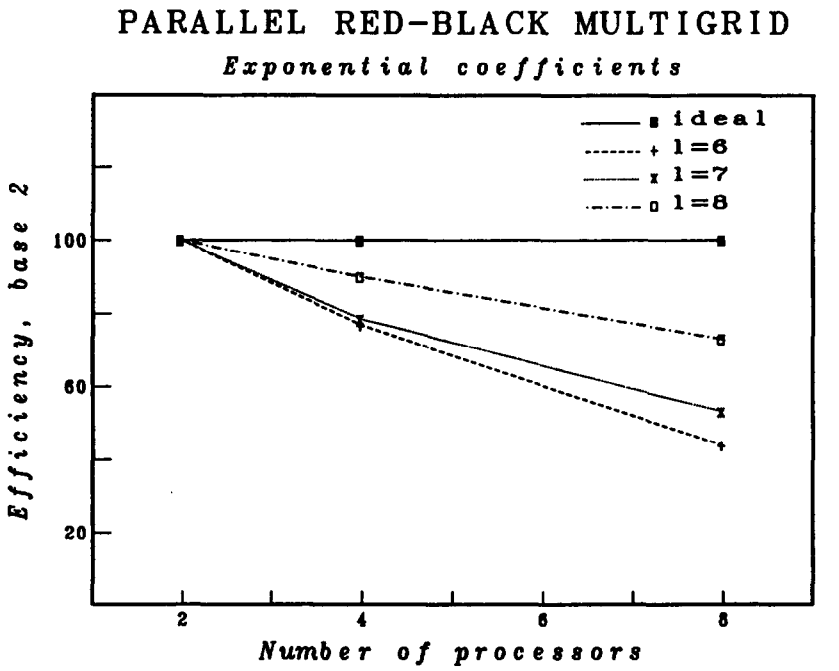

FIG. 11. Efficiency base 2 (i.e., w.r.t. the execution time on two processors) of the parallel multigrid code with an exact coarse solution for the exponential coefficients equation, with $(\nu_1, \nu_2) = (2,2)$, fine-grid levels $l = 6, 7, 8$, coarse-grid level $l_0 = 4$, and error tolerance $\varepsilon = 10^{-6}$.

Note that these results are averaged on a number of runs, since the synchronization time in one of the procedures may vary from one run to the next, depending on the order in which the processors get access to the shared memory. In order to get these results, we took a large problem; therefore we have very good overall parallel efficiencies (88.5% for 4 processors and 70% for 8 processors). From Table 2, we see that:

(1) The two most time-consuming steps, namely pre- and postsmoothing, show excellent parallel efficiency, although they require two calls to **READSM** and **WRITSM** (which includes a **BARRIER**). The presmoothing step is a little more efficient than the postsmoothing one; this is due to the fact that on all grids coarser than the fine one, the computation of the red nodes is simpler because the starting guess is 0, and it does not require a previous **READSM** (see Section III.3). These results confirm the choice of the multicolor SOR as a parallel smoother.

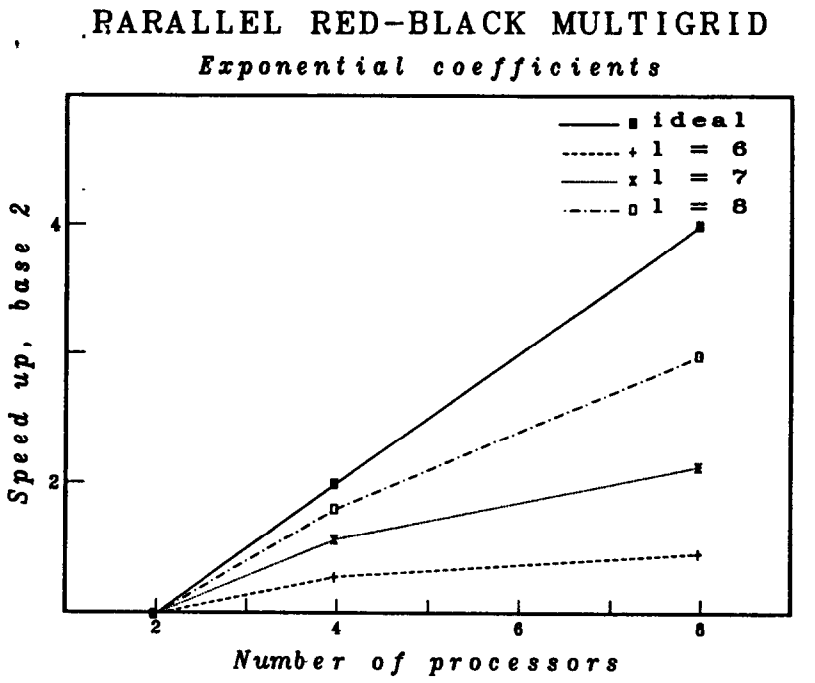

FIG. 12. Speedup, base 2 (i.e., w.r.t. the execution time on two processors) of the parallel multigrid code with an exact coarse solution for the exponential coefficient equation, with $(\nu_1, \nu_2) = (2,2)$, fine-grid levels $l = 6, 7, 8$, coarse-grid level $l_0 = 4$, and error tolerance $\varepsilon = 10^{-6}$.

(2) The second most time-consuming step is the prolongation. This step requires a **READSM**; it can be optimized by calling the vectorized APAL (FPS assembler language) versions of **SAXPY** and **SSCAL**, which add vectors and multiply them by a scalar. Note that the parallel efficiency of this step is also excellent.

(3) One step is actually lowering the parallel performance: the coarse-grid resolution. The time for this resolution increases with the number of processors. This is absolutely normal, since the resolution is inherently sequential, and is performed on all processors. Furthermore, it requires a call to **READSM** and **WRITSM**. This raises the question of implementing the multigrid method with no coarse-grid resolution, or only a few steps of the smoother on the coarse grid. However, when no coarse resolution is performed, the convergence ratio of the multigrid method is known to increase; therefore, although the parallel efficiency can be expected to be better, the increasing number of iterations should also increase the execution time, at

TABLE 2
ANALYSIS OF THE TIME AND PERCENTAGE TIME SPENT IN EACH ROUTINE[a]

	2 APs		4 APs			8 APs		
Procedure	Time[b]	%	Time[b]	%	Eff.[c]	Time[b]	%	Eff.[c]
Presmoothing	.2301	28.1	.1213	26.1	94.8	.0662	22.6	87.1
Matrix multiply	.0691	8.6	.0359	7.6	96.3	.0185	6.3	93.8
New residual	.0298	3.6	.0150	3.1	98.5	.0077	2.5	98.0
Restriction	.0255	3.2	.0128	2.7	99.5	.0066	2.3	98.0
Coarse-grid solve	.0544	6.7	.0548	11.7	49.6	.0614	21.0	22.2
Prolongation	.0828	10.1	.0434	9.3	95.3	.0229	8.1	90.8
Correction	.0189	2.3	.0095	1.9	99.5	.0050	1.7	98.4
Postsmoothing	.2479	30.1	.1334	28.8	92.9	.0722	24.7	85.9
Euclidean norm	.0597	7.3	.0411	8.8	72.5	.0316	10.8	47.3
Total	.8177	100.0	.4624	100.0	88.5	.2921	100.0	70.0

[a] For one iteration of the parallel multigrid code, using a fast Poisson solver on the coarse grid (level 4), for the solution of the model problem (II.4), with fine level = 8 (65,025 nodes), one presmoothing and one postsmoothing step.
[b] In seconds.
[c] Average parallel efficiency of each routine.

least for the Poisson problem, for which very fast solvers are available. For a more general problem for which no fast solver is available, this approach might be a lot more interesting. We implemented the code with 10 steps of the SOR method with a relaxation parameter $\omega = 1.73$ instead of the fast Poisson solver on the coarse grid. The results are shown on Figures 6, 7, and 8 by the curves labeled NCS. It is readily seen that, although the execution times are somewhat larger than with a fast Poisson solver, the overall parallel efficiency is much higher; these results could be further improved by the use of a more efficient system solver such as the conjugate-gradient method.

We would like to thank J. F. Maitre and M. Schultz for their valuable suggestions and comments. We are also grateful to W. Gropp and D. Keyes for interesting discussions about the parallelization of numerical algorithms. To E. Clementi of IBM Kingston, we express our gratitude for support of this work.

REFERENCES

1 L. M. Adams, Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers, Ph.D. Thesis, Univ. of Virginia, Charlotteville; also published as NASA CR-166027, NASA Langley Research Center, Hampton, Va., 1982.

2 L. M. Adams and J. Ortega, A multicolor SOR method for parallel computation, *Proc. 1982 Int. Conf. Par. Proc.*, 1982, pp. 53–56.
3 D. Barkai and A. Brandt, Vectorized multigrid Poisson solver for the CDC Cyber 205, *Appl. Math. Comput.* 13:215–228 (1983).
4 R. E. Bank and T. Dupont, An optimal process for solving finite element equations, *Math. Comp.* 36:35–51 (1981).
5 A. Brandt, Multi-level adaptive solutions to boundary value problems, *Math. Comp.* 31:333–390 (1977).
6 A. Brandt, Multigrid solvers on parallel computers, in *Elliptic Problem Solvers* (M. H. Schultz, Ed.), Academic, 1981, pp. 39–83.
7 R. Caltabiano, A. Carnevali, and J. Detrich, Directives for the Use of Shared Bulk Memories, an Extension to the Precompiler, IBM Technical Report KGN 110, 1987.
8 T. F. Chan and Y. Saad, Multigrid algorithms on the hypercube multiprocessor, *IEEE Trans. Comput.* C35:11 (1986).
9 T. F. Chan and R. Schreiber, Parallel networks for multigrid algorithms: Architecture and complexity, *SIAM J. Sci. Statist. Comput.* 6(3):698–711 (1985).
10 E. Clementi and D. Logan, Parallel Processing with a Loosely Coupled Array Processor System, IBM Technical Report KGN 43, 1986.
11 J. J. Dongarra, J. R. Bunch, C. B. Moler, and J. W. Stewart, LINPACK *Users' Guide*, SIAM Publications, Philadelphia, 1979.
12 H. Foerster, K. Stüben, and U. Trottenberg, Non-standard multigrid techniques using checkered relaxation and intermediate grids, in *Elliptic Problem Solvers* (M. H. Schultz, Ed.), Academic, 1981, pp. 285–300.
13 D. Gannon and J. Van Rosendale, On the structure of parallelism in a highly concurrent PDE solver, *J. Parallel and Distributed Comput.* 3:106–135 (1986).
14 A. Greenbaum, A multigrid method for multiprocessors, *Appl. Math. Comput.* 19:75–88 (1986).
15 W. D. Gropp and D. E. Keyes, Complexity of parallel implementation of domain decomposition techniques for elliptic partial differential equations, *SIAM J. Sci. Statist. Comput.*, to appear.
16 W. Hackbusch, *Multigrid Methods and Applications*, Springer Series in Computational Mathematics 4, Springer, 1985.
17 R. Herbin, W. D. Gropp, and D. E. Keyes, A Domain Decomposition Technique on a Loosely Coupled Array of Processors, IBM Technical Report KGN 124, 1987.
18 J. F. Maitre and F. Musy, Multigrid methods: Convergence theory in a variational framework, *SIAM J. Numer. Anal.* 21:657–671 (1984).
19 O. McBryan and E. Van de Velde, Elliptic equations algorithms on parallel computers, *Comm. Appl. Numer. Math.* 2:311–318 (1986).
20 O. McBryan and E. Van de Velde, Hypercube algorithms and implementations, *SIAM J. Sci. Statist. Comput.* 8:227–287 (1987).
21 D. P. O'Leary, Ordering schemes for parallel processing of certain mesh problems, *SIAM J. Sci. Statist. Comput.* 5:620–632 (1984).
22 J. M. Ortega and R. G. Voigt, Solution of partial differential equations on vector and parallel computers, *SIAM Rev.* 27(2):149–240 (1985).

23 H. Stüben and U. Trottenberg, Multigrid methods: Fundamental algorithms, model problem analysis and applications, in *Multigrid Methods*, Proceedings Köln-Porz, Nov. 1981 (W. Hackbush and U. Trottenberg, Eds.), Lect. Notes in Math., 960, Springer, Berlin, 1982.

24 P. Swarztrauber and R. Sweet, Efficient FORTRAN Subprograms for the Solution of Partial Differential Equations, NCAR-TN/IA-109, 1975.

25 R. Varga, *Matrix Iterative Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1962.

26 P. Wesseling, Theoretical and practical aspects of a multigrid method, *SIAM J. Sci. Statist. Comput.* 3:387–407 (1982).

27 P. O. Frederickson and O. A. McBryan, *Parallel Superconvergent Multigrid*, Applied Math. Ser., Pitman, Boston, to appear.