Abstract

We describe and analyse a family of highly parallel special purpose computing networks that implement multi-grid algorithms for solving elliptic difference equations. The networks have many of the features and advantages of systolic arrays. We consider the speedup achieved by these designs and how this is affected by the choice of algorithm parameters and the level of parallelism employed. We find, for example, that when there is one processor per grid-point, the designs cannot avoid suffering a loss of efficiency as the grid-size tends to zero.

1. Introduction

We shall describe and analyse a family of highly parallel special-purpose computing networks that implement multi-grid algorithms for solving elliptic difference equations. These networks have the same characteristics - regularity, local communciation, and repetitive use of a single, simple processing element - that make systolic architectures attractive [9]. These architectural advantages make it possible to build large computing networks of VLSI cells that would be relatively cheap, reliable, and very powerful.

Parallel Networks for Multi-Grid Algorithms: Architecture and Complexity

Report # 262

September 19, 1983

Tony F. Chan[1]
and Robert Schreiber[2]

---

[1]Box 2158, Yale Station, Computer Science Dept., Yale Univ., New Haven, CT 06520.

This work was supported in part by Department of Energy Grant DE-AC02-81ER10996 and completed while this author was a visitor at Institute National Recherche en Informatique et Automatique, Le Chesnay, France in 1983.

[2]Computer Science Dept., Stanford Univ., Ca 94305.

This work was supported in part by the Office of Naval Research under contract number N00014-82-K-0703.

Both a basic and a full multi-grid algorithm are considered. The basic method reduces the error in a given initial approximation by a constant factor in one iteration. The full method requires no initial guess and produces a solution with error proportional to the truncation error of the discretization. These algorithms are representative of many variants of linear and nonlinear multi-grid algorithms.

The analysis assumes that we are solving a linear system originating in a discretization of an elliptic partial differential equation on a rectangle in $\mathbf{R}^d$, using a regular n^d point grid. The network is a system of grids of processing elements. For each $1 \leq k \leq K$, processor grid P_k has $(n_k)^\gamma$ elements, where γ is an integer less than or equal to d, and $n_K = n$. The machine implements a class of multi-grid algorithms using a corresponding system of nested point grids. For each $1 \leq k \leq K$, point grid G_k has $(n_k)^d$ points. The key assumption, which is quite realistic, is that it takes this machine $O((n_k)^{d-\gamma})$ time to carry out the computation required by one step of the multi-grid algorithm on point grid G_k using processor grid P_k.

We shall consider the efficiency of these parallel implementations, defining efficiency to be the ratio of the speedup achieved to the number of processors employed [8]. We shall consider a design to be efficient if this ratio remains bounded by a positive constant from below as $n \rightarrow \infty$. The analysis will show that when $\gamma < d$ some algorithms can be efficiently implemented. But when $\gamma = d$ (this is the most parallelism one can reasonably attempt to use) no algorithm can be efficiently implemented. There does exist, in this case, one group of algorithms for which the efficiency falls off only as $(\log n)^{-1}$.

The analysis assumes that we implement the same algorithms used by uniprocessor systems. Convergence results for these algorithms have been rather well-developed recently [1, 5, 7]. We make no attempt to develop algorithms that exhibit concurrent operation on several grids. Note, however, that some encouraging experimental results with such an algorithm have been obtained recently by Gannon and Van Rosendale [6].

In any discussion of the practical use of a specialized computing device, it must be acknowledged that overspecialization can easily make a design useless. At least, the designed device should be able to solve a range of size of problems of a particular structure, perhaps solving large problems by making several passes over the data, solving a sequence of smaller subproblems, or with some other techniques. We shall consider how a large grid, with $(mn)^d$ points, can be handled by a system of processor grids with n^γ elements each having $O(m^d n^{d-\gamma})$

memory cells. Problems on nonrectangular domains can be handled by techniques requiring repeated solutions on either rectangular subdomains or containing domains.

Brandt [3] has also considered parallel implementations of multi-grid methods. He discusses the use of various interconnection networks and appropriate smoothing iterations. One of our results, a $(\log n)^2$ time bound for fully parallel, full multi-grid algorithms, is also stated in his paper.

2. Multi-Grid Algorithms

We shall consider multi-grid algorithms in a general setting. The continuous problem is defined by the triple {H, a(u,v), f(v)}, where H is a Hilbert space with a norm ||.||, a(u,v) is a continuous symmetric bilinear form on H x H, and $f(v) : H \rightarrow R$ is a continous linear functional. The problem is:

$$\text{Find } u \in H \text{ such that } \quad a(u,v) = f(v) \text{ for all } v \in H. \tag{1}$$

It can be shown that if a(*,*) satisfies certain regularity conditions (for example, that $a(v,v) \geq c_0 ||v||^2$ for all $v \in H$), then Problem (1) has a unique solution [4].

We consider finite dimensional approximations of Problem (1). Let M_j, $j \geq 1$, be a sequence of N_j-dimensional spaces, on which one can define a corresponding bilinear form $a_j(u,v)$ and a corresponding continuous linear functional $f_j(v)$, which are constructed to be approximations to a(u,v) and f(v) respectively. Also, since the multi-grid algorithms involve transferring functions between these spaces, we have to construct extension (interpolatory) operators $E_j : M_{j-1} \rightarrow M_j$.

We shall give two multi-grid algorithms, namely BASICMG and FULLMG, with FULLMG calling BASICMG in its inner loop. The two algorithms differ in that BASICMG starts its computation on the finest grid and works its way down to the coarser grids, whereas FULLMG starts with the coarsest grid and works its way up to the finest grid. In the conventional single processor case, BASICMG reduces the error on a certain grid by a *constant factor* in optimal time, whereas FULLMG reduces the error to *truncation error level* in optimal time.

We give the basic multi-grid algorithm BASICMG in Table (2-1). This is a recursive algorithm, although in practice it is usually implemented in an iterative fashion. The iterations are controlled by the predetermined parameters (c,j,m). In this sense it is a direct method, unlike related adaptive algorithms which control the iterations by examining relative changes in the residuals [2]. Figure (2-1) illustrates the iteration sequence in the case c = 2. The major

computational work is in the smoothing sweeps (subroutine SMOOTH), which usually consists of some implementation of the successive-over-relaxation or Jacobi iteration or the conjugate gradient method. The smoothing sweeps are used to annihilate the highly oscillatory (compared to the grid spacing) components of the error in z efficiently. We require that a suitably "parallel" method, Jacobi or odd-even SOR for example, be used as the smoother. In the next section, we shall discuss new architectures for implementing these smoothing operations in more efficient ways.

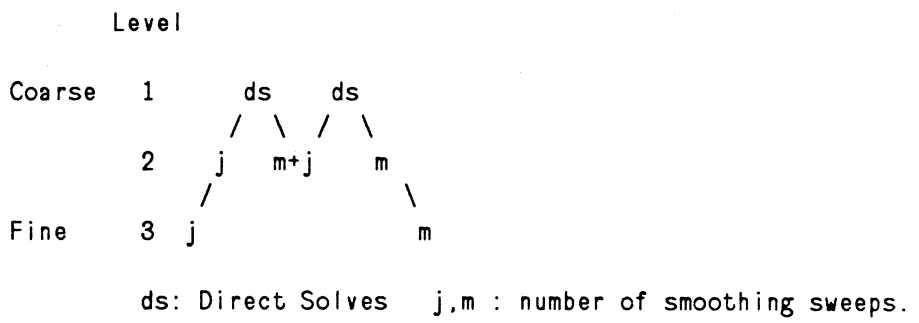
Table 2-1: BASICMG Algorithm

Algorithm BASICMG(k,z,c,j,m,a_k,f_k)

```
<Computes an approximation to u_k ∈ M_k,
 where a_k(u_k,v) = f_k(v) for all v ∈ M_k,
 given an initial guess z ∈ M_k.
 Returns the approximate solution in z.
 Reduces initial error in z by a constant factor.>

   If k = 1 then
     Solve the problem using a direct method. Return solution z.
   else
     <Smoothing step (j sweeps):>
        z ⇐ SMOOTH(j,z,a_k,f_k).

     <Form coarse grid correction equation:>
        a_{k-1}(q,v) = b_{k-1}  for all v ∈ M_{k-1}.

     <Solve coarse grid problem approximately by c cycles of BASICMG:>
        q ⇐ 0.
        Repeat c times:
           BASICMG(k-1,q,c,j,m,a_{k-1},b_{k-1})

     <Correction step:>
        z ⇐ z + E_k q.

     <Smoothing step (m sweeps):>
        z ⇐ SMOOTH(m,z,a_k,f_k).
   End If
End BASICMG
```

We give the full multi-grid algorithm FULLMG in Table (2-2). In the BASICMG algorithm, the choice of initial guess for u_k is not specified. In practice, good initial guesses are sometimes available essentially free (for example, from solutions of a nearby problem, from solutions at a previous time step, etc.). The FULLMG algorithm interpolates approximate solutions on coarser

Figure 2-1: Iteration of BASICMG for k = 3 and c = 2



grids as intial guesses for the BASICMG algorithm. It is also recursive and non-adaptive. Figure (2-2) illustrates the iteration sequence in the case k = 3, c = 2 and r = 1.

Table 2-2: FULLMG Algorithm

Algorithm FULLMG(k,z_k,r,c,j,m,a_k,f_k)

<Computes an approximation z_k to $u_k \in M_k$
where $a_k(u_k,v) = f_k$ for all $v \in M_k$,
using r iterations of BASICMG,
using initial guess from interpolating the approximate
solution obtained on the next coaser grid.
Solution obtained can be proven to have truncation error accuracy.>

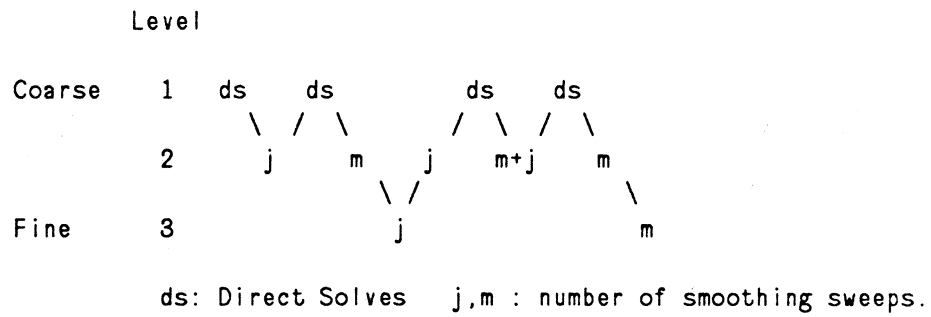
If k = 1 **then**
Solve the problem using a direct method to get z_1.
else
<Obtain solution on next coarser grid:>
FULLMG($k-1,z_{k-1},r,c,j,m,a_{k-1},f_{k-1}$).

<Interpolate z_{k-1}:>
$z_k \Leftarrow E_k\, z_{k-1}$.

<Reduce the error by iterating BASICMG r times:>
Repeat r times:
BASICMG(k,z_k,c,j,m,a_k,f_k).
End if
End FULLMG

We would like to summarize briefly the accuracy and convergence behaviour of the above two multi-grid algorithms. Since the main emphasis of this paper is on the algorithmic aspects of these multi-grid algorithms, we shall refer the reader to the literature for more details. The framework presented here is based on the work of and Bank and Dupont [1] and Douglas [5].

Figure 2-2: Iteration of FULLMG for k = 3, c = 2 and r = 1



The accuracy and the convergence of the BASICMG algorithm obviously depend on the three crucial steps of the algorithm: smoothing, coarse grid transfer, and fine grid correction. The basic requirements are that the smoothing sweeps annihilate the high frequency components of the error efficiently, the coarse grid correction q be a good approximation to the fine grid error in the low frequency components, and the interpolation operators (E_j's) be accurate enough. These conditions can be formalized into mathematically precise hypotheses which can then be verified for specific applications [5]. Assuming these hypotheses, one can show that Algorithm BASICMG reduces the error on level k by a constant factor provided that enough smoothing sweeps are performed. Moreover, it can be shown (see Section 4) that Algorithm BASICMG (for small values of c) can achieve this in optimal time, i.e. $O(N_k)$ arithmetic operations. Obviously, the work needed depends on the accuracy of the initial guess and increases with the level of accuracy desired. Often, one is satisfied with *truncation error accuracy*, i.e. $||z - u|| = O(||u_k - u||) \leq C\ N_k^{-\theta}$ for some fixed θ and C which are independent of k. For a general initial guess, the straightforward application of Algorithm BASICMG to reduce the initial error to this level takes $O(N_k \log(N_k))$ time, which is not optimal. The FULLMG algorithm overcomes this problem by using accurate intial guesses obtained by interpolating solutions from coarser grids. The convergence result for Algorithm BASICMG can be combined with the basic approximation properties of the various finite dimensional approximations $\{M_j, a_j, f_j\}$ to show that Algorithm FULLMG computes a solution z_k that has truncation error accuracy in $O(N_k)$ time.

3. The Computing Network

In this section we describe a simple parallel machine design for multi-grid iteration. We restrict attention to linear elliptic problems in d dimensions over rectangular domains, to discretizations based on grids of n^d points, and to multi-grid methods based on a system of point-grids $\{G_k\}_{k=1}^K$ where G_k has $(n_k)^d$ gridpoints, with mesh lengths $h_{k,j}$, $1 \leq j \leq d$, the finest grid has $n_K = n$, and

$$n_{k+1} = a\,(n_k + 1) - 1, \quad k = 1, 2, \ldots, K\text{-}1$$

for some integer $a \geq 2$.

The machine consists of a system of processor-grids $\{P_k\}_{k=1}^K$ corresponding to the point-grids. Each processor-grid is an $(n_k)^\gamma$ lattice in which a processor is connected to its 2γ nearest neighbors.

We shall employ a standard multi-index notation for gridpoints and processors. Let

$$\mathbf{z}_n^+ \equiv \{0, 1, \ldots, n\text{-}1\}.$$

Let $\mathbf{z}_{n,s}^+ \equiv (\mathbf{z}_n^+)^s$, the set of s-tuples of nonnegative integers less than n. We shall make use of a projection operator $\pi_r^s : \mathbf{z}_{n,r}^+ \rightarrow \mathbf{z}_{n,s}^+$ defined for $r \geq s$ by

$$\pi_r^s((i_1, \ldots, i_r)) = (i_1, \ldots, i_s).$$

By convention, if $\underline{i} \in \mathbf{z}_{n,s}^+$, then $\underline{i} = (i_1, \ldots, i_s)$. Also let $\underline{1} = (1, \ldots, 1)$. We shall also use the norm $|\underline{i}| \equiv |i_1| + \ldots + |i_s|$ on $\mathbf{z}_{n,s}^+$.

We shall label the gridpoints in G_k with elements of $\mathbf{z}_{n_k,d}^+$ in such a way that the point with label $\underline{i}$ has spacial coordinates $(i_1 h_{k,1}, i_2 h_{k,2}, \ldots, i_d h_{k,d})$. Similarly, we label processors in P_k with indices in $\mathbf{z}_{n_k,\gamma}^+$.

Thus, processors $\underline{i}$ and $\underline{k}$ are connected if $|\underline{i} - \underline{k}| = 1$. In order to make the machine useful for problems with periodic boundary conditions, we might also add "wrap-around" connections, so that $\underline{i}$ and $\underline{k}$ are connected if $|(\underline{i} - \underline{k}) \bmod n| = 1$. In Section 3.1, it is shown that periodic problems can also be handled without these connections.

Evidently, if each processor has $O(n^{d-\gamma})$ memory cells, we can store the solution, forcing function, and O(1) temporary values belonging to the whole of grid G_k in the processors of P_k; we store gridpoint $\underline{i}$ in processor $\pi_d^\gamma(\underline{i})$ for $\underline{i} \in \mathbf{z}_{n,d}^+$.

With the given connectivity, smoothing sweeps of some types can be accomplished in $O(n^{d-\gamma})$

time. It is not necessary for the stencil of the difference scheme to correspond to the connectivity of the processor grid. Jacobi or odd/even SOR smoothing can be so implemented, for example. Let t be the time taken by a single processor to perform the operations at a single gridpoint that, done over the whole grid, constitute a smoothing sweep. If S is the time to implement a smoothing sweep over the whole of grid G_k on processor-grid P_k, then

$$S = t\, n_k^{d-\gamma}. \tag{2}$$

Grid P_k is connected to grid P_{k+1}. Processor $\underline{i} \in P_k$ is connected to processor $a(\underline{i}+\underline{1}) - \underline{1} \in P_{k+1}$. These connections allow the inter-grid operations (forming coarse grid forcing terms b_k and interpolation E_k) to also be computed in O(S) time. We refer to the system of processor-grids $\{P_1, ..., P_J\}$ as the machine L_J for J = 1, 2, ..., K.

The execution of the BASICMG iteration by L_k proceeds as follows.

1. First, j smoothing steps on grid G_k are done by P_k. All other processor-grids are idle.
2. The coarse grid equation is formed by P_k and transferred to P_{k+1}.
3. The c cycles of BASICMG on grid k-1 are performed by L_{k-1}.
4. The solution q is transferred to P_k and interpolation $E_k q$ is performed by P_k.
5. The remaining m smoothing steps are done by P_k.

Let W(n) represent the time needed for steps 1, 2, 4 and 5. Then

$$W(n) = (j + m + s)\, t\, n^{d-\gamma} \tag{3}$$

where s is the ratio of the time required to perform steps 2 and 4 to the time needed for one smoothing sweep. Note that s is independent of n, d and γ.

The natural way to build such a machine is to embed the $\gamma = 1$ machine in two dimensions as a system of communicating rows of processors, the $\gamma = 2$ machine in three dimensions as a system of communicating planes, etc. Of course, realizations in three-space are possible for any value of γ. Gannon and Van Rosendale [6] consider the implementation of the fully parallel machine ($\gamma = d$) on proposed VLSI and Multi-Microprocessor system architectures.

This design differs from systolic array designs in that there is no layout with all wire lengths equal. But for reasonably large machines the differences in wire length should not be so great as to cause real difficulties. Moreover, one need not continue to use ever coarser grids until a 1x1

grid is reached. In practice, 3 or 4 levels of grids could be used and most of the multi-grid efficiency retained; this would make the construction much simpler.

3.1. Solving Larger Problems

Suppose there are $(mn)^d$ gridpoints and only n^γ processors. Assume that each processor can store all information associated with $m^d n^{d-\gamma}$ gridpoints. Now we map gridpoints to processors in such a way that neighboring gridpoints reside in neighboring processors. To do this we define a mapping f_m: $\mathbf{z}^+_{mn} \rightarrow \mathbf{z}^+_n$, such that, for all $i, j \in \mathbf{z}^+_{mn}$, $|f_m(i) - f_m(j)| \leq |i - j|$, as follows. Let $j = qn + r$ where q and r are integers, $0 \leq r \leq n-1$. Now let

$$f_m(j) \in \begin{cases} r & \text{if q is even} \\ n\text{-}1\text{-}r & \text{if q is odd.} \end{cases}$$

Now if m is even, then $f_m(0) = f_m(mn-1) = 0$, so that periodic boundary conditions can be handled without any "wrap-around" connections. This operation corresponds to folding a piece of paper in a fan-like manner; for $m = 10$, for example, like this:

/\/\/\/\/\

To map a multidimensional structure we fold it as above in each coordinate. Let the processor-grid have n^d elements and the point-grid have $m_1 n \times m_2 n \times \ldots \times m_d n$ points. Point $\underline{i}$ can be stored in processor $F_d(m_1, \ldots, m_d, n; \underline{i})$ where $F_d(\underline{i}) \equiv (f_{m_1}(i_1), \ldots, f_{m_d}(i_d))$. If we have only n^γ processors then we map $\underline{i}$ into $F_d^\gamma(\underline{i})$ where $F_d^\gamma(\underline{i}) \equiv F_\gamma(\pi_d^\gamma(\underline{i}))$.

4. Complexity

In this section, we are going to analyse the time complexity of the two multi-grid algorithms, BASICMG and FULLMG, as implemented by the different architectures just discussed. It turns out that the complexity of the two algorithms is very similar. Since the BASICMG algorithm is simpler and is called by FULLMG, we shall discuss and analyse it first. After that, we shall indicate how to derive the results for FULLMG.

4.1. Complexity of BASICMG

To simplify the analysis, we shall assume that the computational domain is a rectangular parallelopiped and is discretized by a hierarchy of cartesian grids (corresponding to the M_j's) each with n_j mesh points on each side (denoted the n_j-grid). Further, we assume that the n_j's satisfy

$n_j = a\,(n_{j-1}+1) - 1$ where a is an integer bigger than one. Generally , we denote by T(n) the time complexity of the BASICMG Algorithm on an n-grid. By inspecting the description of Algorithm BASICMG, it is not difficult to see that T(n) satisfies the following recurrence:

$$T(an) = c\ T(n) + W(an), \tag{4}$$

where W(an) denotes the work needed to preprocess and postprocess the (an)-grid iterate before and after transfer to the coarser n-grid. We have the following general result concerning the solution of (4), the proof of which is elementary.

Lemma 1: Let T_p be a particular solution of (4), i.e.

$$T_p(an) = c\ T_p(n) + W(an), \tag{5}$$

then the general solution of (4) is:

$$T(n) = \alpha\ n^{\log_a c} + T_p(n), \text{ where } \alpha \text{ is an arbitrary constant.} \tag{6}$$

The term W(an) includes the smoothing sweeps, the computation of the coarse grid correction equation (i.e. the right-hand-side b_{k-1}) and the interpolation back to the fine grid ($E_k q$). The actual time needed depends on the architecture used to implement these operations (specifically the dimensionality of the domain and the number of processors available on an n-grid). In general, as derived in Section 3, W(n) is given by

$$W(n) = (j+m+s)\ t\ g(n) \equiv \beta\ g(n)\ , \tag{7}$$

where $g(n) = n^p$ with $p \equiv d\text{-}\gamma$.

In Table (4-1), we give the form of the function g(n) as a function of the architecture and the dimensionality of the domain. We also give a bound on the total number of processors (P) needed to implement the architecture and note that it is always the same order as the number of processors on the finest grid. For a d-dimensional problem with n^γ processors on the n-grid, we have

$$P(\gamma) = \begin{cases} 1 & \text{if } \gamma = 0, \\ (a^\gamma/(a^\gamma-1))\ n^\gamma & \text{if } \gamma > 0. \end{cases}$$

We have the following general result for this class of functions g(n).

Lemma 2: If $W(n) = \beta\ n^p$, then we can take the following as particular solution of (4):

$$T_p(n) = \begin{cases} \beta\ (a^p/(a^p-c))\ n^p & \text{if } p \neq \log_a c, \\ \beta\ n^p \log_a n & \text{if } p = \log_a c. \end{cases} \tag{8}$$

Table 4-1: Table of g(n)

Architecture	1-D	2-D	3-D	Total # of processors on all grids, P
1 processor	n	n^2	n^3	1
n processors	1	n	n^2	$(a/(a-1))\ n$
n^2 processors	-	1	n	$(a^2/(a^2-1))\ n^2$
n^3 processors	-	-	1	$(a^3/(a^3-1))\ n^3$

Note: Architecture column gives number of processors on the n-grid.

Combining the results of the last two lemmas, we arrive at our main result:

Theorem 3: The solution of (4) for $W(n) = \beta\, n^p$ satisfies:

$$T(n) = \begin{cases} \beta\,(a^p/(a^p-c))\,n^p + O(n^{\log_a c}) & \text{if } c < a^p, \\ \beta\, n^p \log_a n + O(n^p) & \text{if } c = a^p, \\ O(n^{\log_a c}) & \text{if } c > a^p. \end{cases} \tag{9}$$

Note that in the last case, $\alpha \neq 0$ (in Equation (6)) because $T_p(n)$ does not satisfy the boundary conditions.

For the first two cases ($c \leq a^p$), we can determine the highest order term of T(n) explicitly. However, for the case $c > a^p$, the constant in the highest order term depends on the initial condition of the recurrence (4) (i.e. the time taken by the direct solve on the coarsest grid), which is more difficult to measure in the same units as that of the smoothing and interpolation operations. Fortunately, the complexity for this case is non-optimal and thus not recommended for use in practice and therefore, for our purpose, it is not necessary to determine this constant.

Based on the results in Theorem 3 and the specific forms of the function g(n) in Table 4-1, we can compute the time complexity of Algorithm BASICMG for various combinations of c, a and p, some of which are summarized in Table 4-2, where we tabulated the highest order terms of $T(n)/\beta$.

The classical one processor ($\gamma = 0$) optimal time complexity results [1, 5] are contained in these tables. For example, in two dimensions ($d = 2$) $g(n) = n^2$ and Table 4-2 shows that, for the refinement parameter $a = 2$, $c < 4$ gives an optimal algorithm ($O(n^2)$) whereas $c \geq 4$ is non-

Table 4-2: Time Complexity T(n)/β of Algorithm BASICMG

$a = 2$

c	p = d − γ = 3	p = 2	p = 1	p = 0
1	$8/7\ n^3$	$4/3\ n^2$	$2\ n$ *	$\log_2 n$
2	$8/6\ n^3$	$4/2\ n^2$ *	$n\log_2 n$	$O(n)$
3	$8/5\ n^3$	$4\ n^2$ *	$O(n^{\log_2 3})$	$O(n^{\log_2 3})$
4	$8/4\ n^3$ *	$n^2\log_2 n$	$O(n^2)$	$O(n^2)$

$a = 3$

c	p = d − γ = 3	p = 2	p = 1	p = 0
1	$27/26\ n^3$	$9/8\ n^2$	$3/2\ n$ *	$\log_3 n$
2	$27/25\ n^3$	$9/7\ n^2$	$3\ n$ *	$O(n^{\log_3 2})$
3	$27/24\ n^3$	$9/6\ n^2$ *	$n\log_3 n$	$O(n)$
4	$27/23\ n^3$	$9/5\ n^2$ *	$O(n^{\log_3 4})$	$O(n^{\log_3 4})$

$a = 4$

c	p = d − γ = 3	p = 2	p = 1	p = 0
1	$64/63\ n^3$	$16/15\ n^2$	$4/3\ n$ *	$\log_4 n$
2	$64/62\ n^3$	$16/14\ n^2$	$4/2\ n$ *	$O(n^{\log_4 2})$
3	$64/61\ n^3$	$16/13\ n^2$	$4\ n$ *	$O(n^{\log_4 3})$
4	$64/60\ n^3$	$16/12\ n^2$ *	$n\log_4 n$	$O(n)$

a : mesh refinement ratio
c : number of correction cycles in BASICMG
γ : n^γ processors on the n-grid
d : dimension of the domain
***** Asymptotic Efficiency Boundary (See Theorem 5)

optimal. *More generally, we have an optimal scheme if and only if* $c < a^d$. For example, the larger a is or the larger d is, the larger the value c can take for the algorithm to remain optimal. However, with a larger value of a, more relaxation sweeps are needed and a larger value of c has to be taken in order to achieve the same accuracy. We note that the constant in the highest order term of T(n) does not vary a great deal with either c or a (they are all about one, especially for the larger values of a). This suggests that the best balance between speed and accuracy can be achieved by choosing the largest value of c (or close to it) such that the algorithm remains optimal. When $d\text{-}\gamma = 2$ and $a = 2$, this means taking c to be 2 or 3.

4.2. Efficiency, Speedup, Accuracy, and Optimal Design

Next, we are going to look at the effects of the new architectures on the performance of the BASICMG algorithm. There are four parameters in this study: c, a, γ and d. We shall call a particular combination of these four parameter a *design*. We shall use the notation $T(c,a,\gamma,d)$ to denote the corresponding complexity of the design. One of the main issues that we would like to address is the *efficiency* E and *speedup* S of a particular design, which are defined as [8]:

Definition 4:

$$S(c,a,\gamma,d) = T(c,a,0,d) \,/\, T(c,a,\gamma,d) ,$$

$$E(c,a,\gamma,d) = T(c,a,0,d) \,/\, (P(\gamma)\, T(c,a,\gamma,d)) .$$

The speedup S measures the gain in speed over the one processor architecture while the efficiency E reflects the tradeoff between processors and time and measures the efficiency with which the architecture exploits the extra processors to achieve the speedup. The optimal efficiency is unity, in which case a P-fold increase in the number of processors reduces the time complexity P-fold. In general, the efficiency E and the speedup S are functions of n. We shall call a design *asymptotically efficient* if E tends to a constant as n tends to infinity and *asymptotically inefficient* if it tends to zero. We shall primarily be concerned with analysing the efficiency E of a design in this section. The speedup S can be easily read from Table 4-2.

For determining the asymptotic efficiency of a design, it suffices to determine the highest order term of E. The efficiency E can be derived from the explicit expressions for T and P in a straightforward manner. Since the efficiency for $\gamma = 0$ is unity by definition, we shall only be interested in $\gamma > 0$. We summarize the results in the following theorem.

Theorem 5: Assume $\gamma > 0$.

(1) If $c < a^{d-\gamma}$ then $E(c,a,\gamma,d) = (a^{\gamma}-1)(a^{d-\gamma}-c)/(a^{d}-c)$.

(2) If $c = a^{d-\gamma}$ then $E(c,a,\gamma,d) = (a^{\gamma}-1)a^{d-\gamma} / ((a^{d}-c)\log_a n)$.

(3) If $c > a^{d-\gamma}$ then

$$E(c,a,\gamma,d) = \begin{cases} O(1/(n^{\log_a c - d + \gamma})) & \text{if } c < a^{d}, \\ O(\log_a n / n^{\gamma}) & \text{if } c = a^{d}, \\ O(1/n^{\gamma}) & \text{if } c > a^{d}. \end{cases}$$

Based on the above theorem, we can immediately make the following observations:

1. *A design is asymptotically efficient if and only if* $c < a^{d-\gamma}$. This inequality defines an *efficiency boundary* in the four parameter space of $\{c,a,\gamma,d\}$, the projections of which are shown by *'s in Table 4-2.
2. *The fully parallel design ($\gamma = d$) is always asymptotically inefficient.* This follows because to have an efficient design in this case requires $c < 1$ which is meaningless for the multi-grid algorithm.
3. Define a logarithmically asympototically efficient design to be one with $E = O(1/\log_a n)$ as n tends to infinity. *A fully parallel design is logarithmically asymptotically efficient if and only if $c = 1$.* This is case (2) in Theorem 5.
4. *If we start with a non-optimal design in the one processor case, then adding more processors will not make the design asymptotically efficient.* This corresponds to the last two cases in Case (3) of Theorem 5. The reason is that too many coarse grid correction cycles are performed so that even if more processors are added to speed up the setup time for transferring to the coaser grids, too much time is spent on the coaser grids.

Asymptotically efficient designs are theoretically appealing. They indicate that the extra processors are utilized efficiently to achieve the speedup. For this reason, it is interesting to consider the following problem:

Optimal Design Problem:
For a given problem (i.e. given d), find the design that
minimizes T(n) and/or maximizes the accuracy of the computed solution
subject to the constraint that it is asymptotically efficient.

Table 4-3: Influence of Design Parameters on Optimality Conditions

		Design Parameters		
		c	a	γ
Optimality Conditions	Max Accuracy	large	small	indep
	Min T(n)	small	large	large
Constraint	Efficiency E	small	large	small

In Table 4-3, we indicate the influence of each of the three design parameters $\{c,a,\gamma\}$ on the optimality conditions and the constraint. For example, the accuracy of the solution is independent of γ and to achieve maximum accuracy, one should take c large and a small. The appropriate choice of optimality condition depends on the requirements of the given problem. Moreover, the general optimal design problem may not have a unique or bounded solution in the three parameter space $\{c,a,\gamma\}$. In practice, however, we usually do not have the freedom to choose all three parameters. If the number of free parameters are restricted, then the optimal design problem may have a unique solution.

We shall illustrate this by fixing two of the three parameters in turn and study E as a function of the free parameter. First, let us fix c and a and consider the effect of varying γ. In other words, we consider the case where the multi-grid algorithm and the refinement of the domain are fixed and we are free to choose the architecture. Varying γ corresponds to moving across a particular row of Table 4-2. It is easy to see that one achieves a speedup as we use more processors (i.e. as one moves from left to right in one of these rows). However, the efficiency E generally goes down as one uses more processors, and after a certain entry the design starts to be asymptotically inefficient. For example, take the three dimensional case (d = 3), with a = 2 and c = 2. With n processors on the n-grid, the efficiency is E(2,2,1,3) = 1/3. With n^2 processors on n-grid, we have E(2,2,2,3) = $1/\log_2 n$, and with n^3 processors E(2,2,3,3) = O(1/n). Both the last two designs are asymptotically inefficient and thus the $\gamma = 1$ design is the fastest efficient design. *In general, for fixed c and a, the design just to the left of the efficiency boundary is the fastest efficient design.*

Next we shall fix a and γ and vary c (columns in Table 4-2). That is, we fix the architecture and vary the multi-grid algorithm. This time the speedup factor S decreases slightly as we increase c which is not surprising as we are doing more work on coarse grids, and this keeps many processors idle. Again, the efficiency E decreases as we increase c, and after a certain entry, the algorithm becomes asymptotically inefficient. For example, take the two dimensional case with n processors on the n-grid (d = 2, γ = 1) and a = 3. Going down the appropriate column, we have E(1,3,1,2) = = 1/2, E(2,3,1,2) = 2/7 and E(3,3,1,2) = $1/\log_3 n$. Recall that the larger c is, the more accurate is the computed solution and the more robust is the overall algorithm. Thus, the c = 2 design is the most accurate efficient design. *In general, for fixed a and γ, the design just above the efficiency boundary is the most accurate efficient design. If accuracy is no problem, then c can be chosen smaller to speed up the algorithm.*

Finally, we fix c and γ and vary a. Generally, a larger value of a means fewer processors are needed to implement the architecture. It also means that less work has to be done on the coarse grids because they have fewer points. To see the effect of varying a, note that the efficiency boundary moves towards the lower right hand corner of the tables in Table 4-2 as a is increased. This implies that, for a fix architecture (γ) and algorithm (c), using a larger value of a will generally exploit the available processors more efficiently. However, one cannot indiscrimatorily use large values for a because this leads to larger interpolation errors and less accurate solutions. For example, take the two dimensional case with n-processors on the n-grid and c = 2. With a = 2, the algorithm is asymptotically inefficient (E(2,2,1,2) = $1/\log_2 n$) whereas with a = 3 and a = 4, it is asymptotically efficient (E(2,3,1,2) = 2/7, E(2,4,1,2) = 3/7). Thus, the a = 3 design is the most accurate efficient design. *In general, for fixed c and γ, the smallest value of a that yields an efficient design is the most accurate efficient design. If accuracy is no problem, then a can be chosen larger to speed up the algorithm.*

One can carry out similar parametric studies, for example, by fixing one parameter and varying the other two. For instance, if we are free to choose both the architecture (γ) and the algorithm (c), then by inspecting the form of the efficiency constraint $c < a^{d-\gamma}$, it can be seen that smaller values of c allow more processors to be used (larger γ) to produce a faster efficient design. For example, in the a = 2 case, if c = 2 then the p = 2 entry gives the fastest efficient design whereas if c = 1, it becomes the p = 1 entry, with the latter being faster. Similarly, for a fixed c, a larger value of a allows more processors to be used to achieve a faster efficient design.

4.3. Complexity of FULLMG

In this section, we shall derive the complexity of the FULLMG Algorithm. Since FULLMG calls BASICMG, the results here depend crucially on the complexity of Algorithm BASICMG.

Let F(n) denote the time taken by one call to FULLMG. By inspecting the algorithm in Table 2-2, it can easily be verified that F(n) satisfies the following recurrence:

$$F(an) = F(n) + r\,T(an)\,, \tag{10}$$

where we have absorbed the cost of the interpolation step in FULLMG into the interpolation costs of BASICMG (i.e. the term s in Equation (7)). Note that this is just a special case of the recurrence (4), with c = 1 and W(an) = r T(an). By inspecting the entries in Table 4-2, we see that the forcing function W(n) in this case takes the form of either n^p or $n^p\log_a n$. We have the following result for particular solutions of (10) for this class of forcing functions, which can be verified by direct substitution.

Lemma 6:

(1) If $T(n) = \alpha n^p$ then a particular solution of (10) is:

$$F_p(n) = (a^p/(a^p-1))\, r\alpha n^p\,.$$

(2) If $T(n) = \alpha n^p\log_a n$, with $p > 0$, then a particular solution of (10) is:

$$F_p(n) = (a^p/(a^p-1))\, r\alpha n^p\log_a n - (a^p/(a^p-1)^2)\, r\alpha n^p\,.$$

(3) If $T(n) = \alpha\log_a n$ then a particular solution of (10) is:

$$F_p(n) = (r\alpha/2)\,(\,\log_a^2 n + \log_a n\,)\,.$$

Since the homogeneous solution of (10) is a constant, the general solution is dominated by the particular solutions. We give the highest order terms of F(n) in terms of T(n) in the following theorem.

Theorem 7:

(1) If $T(n) = \alpha n^p$ then $F(n) = (a^p/(a^p-1))\, rT(n) + O(1)$.

(2) If $T(n) = \alpha n^p\log_a n$, with $p > 0$, then $F(n) = (a^p/(a^p-1))\, rT(n) + O(n^p)$.

(3) If $T(n) = \alpha\log_a n$ then $F(n) = (r/2)\log_a n\, T(n) + O(\log_a n)$.

In the first two cases, the complexity of F(n) is the same as that of T(n), except for the *constant* multiplicative factor $(a^p/(a^p-1))r$. The $(a^p/(a^p-1))$ part of this constant is very close to

unity for the values of a and p that occur. The r part of the constant applies equally to all entries of Table 4-2 and reflects the number of times BASICMG is called by FULLMG. We point out that the choice of r that results in truncation error level accuracy depends on how efficiently BASICMG reduces its initial error but can be chosen *independent of n* [5]. Thus, the extra multiplicative factor does not affect the asymptotic efficiency of a particular entry in Table 4-2. The complexity of F(n) in the last case is actually increased by a factor of $\log_a n$ over that of T(n). However, the corresponding entries in Table 4-2 are already asymptotically inefficient and thus this extra factor again does not affect the asymptotic efficiency of the design. It follows that the discussions concerning the efficiency, speedup and optimal design for the BASICMG algorithm in Section 4.1 are also valid for the FULLMG algorithm, with the exception that a fully parallel logarithmically asymtotically efficient design is slower by the factor $\log_a n$.

5. Conclusion

We have proposed an architecture based on a system of processor-grids for parallel execution of multi-grid methods based on a system of point-grids. We have analyzed its efficiency and shown that a combination of algorithm and machine is asymptotically efficient if and only if $c < a^{d-\gamma}$, where

- c is the number of coarse grid iterations per fine grid iterations,
- a is the mesh refinement factor,
- d is the dimension of the point-grids,
- γ is the dimension of the processor-grids.

We find therefore that fully parallel designs ---- with $\gamma = d$ ---- cannot be asymptotically efficient. There is, however, only a logarithmic fall-off in efficiency when $c = a^{d-\gamma}$, and for fully parallel designs this occurs for $c = 1$.

References

[1] R. E. Bank and T. Dupont.
An optimal order process for solving elliptic finite element equations.
Math. Comp. 36:35-51, 1981.

[2] A. Brandt.
Multi-level adaptive solutions to boundary-value problems.
Math. Comp. 31:333-390, 1977.

[3] A. Brandt.
Multi-Grid Solvers on Parallel Computers.
Technical Report 80-23, ICASE, NASA Langley Research Center, Hampton, VA, 1980.

[4] P. G. Ciarlet.
The Finite Element Method for Elliptic Problems.
North-Holland, Amsterdam, 1978.

[5] C. Douglas.
Multi-Grid Algorithms for Elliptic Boundary-Value Problems.
PhD thesis, Yale University, 1982.
Computer Science Department Technical Report 223.

[6] D. Gannon and J. van Rosendale.
Highly Parallel Multi-Grid Solvers for Elliptic PDEs: An Experimental Analysis.
Technical Report 82-36, ICASE, NASA Langley Research Center, Hampton, VA, 1982.

[7] W. Hackbusch.
A Multi-Grid Method Applied to a Boundary Value Problem with Variable Coefficients in a Rectangle.
Technical Report 77-17, Mathematisches Institut, Universitat zu Koln, Cologne, 1977.

[8] David J. Kuck.
The Structure of Computers and Computations.
Johm Wiley & Sons, New York, 1978.

[9] H.T. Kung.
Why Systolic Architectures?
IEEE Computer 15(1):37-46, January, 1982.