Some basic facts for efficient massively parallel computation

Owe Axelsson
Department of Mathematics, University of Nijmegen
Toernooiveld, 6525 ED Nijmegen
The Netherlands

Abstract

Four fundamental aspects of efficient massively computation with application for composite materials are discussed:
(i) the need for massively parallel computations and, consequently, the need for distributed memory machines
(ii) the need for scalable algorithms
(iii) some physical limits to massively parallel computations
(iv) the need for dynamic load balancing algorithms.
Massively parallel computation in large scale numerical modelling requires scalable algorithms, i.e., the performance of algorithms must increase (asymptotically) linearly with the number of processors (p). This requires on one hand algorithms with a communication complexity which increases at most linearly with p (and, ideally, decreases in a relative sense to zero) and on the other hand, an algorithm with a (close to) optimal computational complexity when run on a single processor. An example of such an algorithm will be presented. The consequence of the fast growth of the computational complexity with problem size will be illustrated on a 4D partial differential equation. Finally, it will be shown that due to physical limits, in reality there does not exist any asymptotic ($p \to \infty$) state.
Hence computer time must eventually increase with problemsize, no matter how many processors are used. The remedy to this situation in mathematical modelling is to limit the problem size by use of adaptive refinement methods. Some aspects of dynamic adaptive refinement methods will be mentioned.
An application for some composite material research problem is described.

1 Introduction

The computational complexity of the solution of a problem can grow very fast with increasing problem size. To illustrate this, take the classical example of a time dependent PDE in 3D. Assuming that one uses a (quasi)uniform partitioning in time and space, the number of (unknown) parameters to be computed increases as

$$\sim cn^4, \quad n \to \infty$$

for some constant c, where the both the timestep and meshwidth $= O(n^{-1})$.

Assume, for simplicity, that for a given stepsize $h_0 = h^{-1}$, the computer time is 1 second.

Assume now that we must resolve some boundary and/or interior layers of the solution and for this purpose we divide the meshsizes with a factor 10 in each time-space direction. The complexity becomes then $\sim c10^4 n^4$.
In the most ideal case the solver requires only $\sim c10^4 n^4$ arithmetic computer operations. Then we need at least a factor of 10^4 more processors for the same computer time as for the unrefined mesh problem.
Put in another way: An increase of the number of processors with a factor 4000 (say) allows only a mesh refinement with a factor 1/8 for the same computer time (in an ideal case).
To cope with the fast growth of the computational complexity one must use *local* mesh refinements, preferably in the 4D time-space domain. In certain CFD computations one may need to refine the mesh (locally) with perhaps a factor 1/100 or more. The computing time would then grow with a factor 10^8 if one uses a uniform refinement, which is clearly out of reach (10^8 seconds $\simeq$ 3 years!). Use of an adaptive mesh and local refinement can reduce the average meshsize to $h_0/10$, perhaps, but then we still must account for a factor 10^4. Clearly, massively parallel computation is required with a number of processors $p \geq 10^4$, say.

Due to physical constraints, massively parallel computers can not be built with a shared memory, but only with distributed local memories. Therefore, in addition one must use scalable algorithms, i.e., algorithms for which the performance increases linearly with the number of processors. In the talk we shall try to identify the major reasons why massively parallel scientific computation hasn't been a great success yet, and to suggest a solution.

2 Megaflop rates contra computertimes; or nearest neighbor contra global communication

The above already as serious indicated problem is in fact even worse because we have neglected the issue of communication complexity.
On a massively parallel distributed memory machine one can *not* avaid communication overhead: In any reasonable algorithm the computer processors need to exchange data. With an increasing number of processors the time spend an data communication tends to increase relative to the time for computation.
One must therefore use solution algorithms where the communication overhead is not dominating. The most desirable situation from the latter view-point is that only nearest neighbor communication would arise, because communication between processors far apart must take place in general using message-passing via intermediate processors, which latter can increase the communication overhead significantly.
Indeed, there exists (elementary) algorithms which only need nearest neighbor communication. For instance, to solve a (large scale sparse) linear system $Ax = b$ one can use iteration methods in the basic form:
Given x^0, for $k = 0, 1, \ldots$ until convergence compute:

$$x^{k+1} = x^k - \tau(Ax^k - b),$$

or, in preconditioned form:
Given x^0, for $k = 0, 1, \ldots$ until convergence compute:

$$\begin{aligned} r^k &= Ax^k - b, \\ \text{solve } C\delta^k &= r^k, \\ x^{k+1} &= x^k - \tau\delta^k. \end{aligned}$$

Here τ is a damping parameter chosen to get convergence of the iterations (possibly $\tau = \tau_k$ is variable) and C is a matrix chosen to improve the distribution of eigenvalues, i.e., $C^{-1}A$ should have a (much) more favorable eigenvalue distribution than A. However, when coping with the requirement of nearest neighbor communication, C must be very simple, such as a (block) diagonal matrix. The classical (preconditioned) conjugate gradient method has a similar form.
The matrix A need to be defined only locally $A = \sum_e A^{(e)}$ (in the context of f.e.m. there is no need to "assemble" A), and frequently $A^{(e)}$ involves only nearest points in a difference or finite element mesh.
In such cases the above iterations can be executed extremely fast on parallel computers because the need for exchange of data is small and occurs only via nearest-neighbor processors (assuming one has used a proper mapping of mesh points onto processors). Therefore when implementing the above methods on parallel computer architectures one normally gets a close to peak performance measured in megafloprates. Such methods are sometimes referred to as "*embarrasingly parallelizable*'.
However, what obviously matters for a user is not the megaflop rate but the computer times.
The above basic iteration method requires (for elliptic second order pde problems) $O(n^2)$ iterations (or possibly $O(n)$, if the method parameters τ_k can be chosen properly). Hence the computer time grows as $O(n^2N)$ or, at least as $O(nN)$, $n \to \infty$, where N is the total number of meshpoints.
Now there exist very efficient algorithms to solve large scale elliptic type pde, and other similar types of problems, on a single processor. They are generally based on some substructuring of the original problem which can be a multigrid or (algebraic) multilevel iteration method, or some variants of DD methods and, under quite general conditions, they require only a bounded, $O(1)$ or $O(\log n)$ number of iterations for convergence.
However, as the name multilevel of the methods indicates, they are based on a sequence of (nested or nonnested) meshes or subgraphs (subgraphs of the matrix graph belonging to the given sparse matrix). Therefore, as one progresses down in the method to coarser and coarser meshes, the communication must take place to increasingly more distanced processors.
Except possibly in special cases (such as when a hypercube computer architecture is used and one can apply a so-called inverse Gray code mapping of mesh-points to processors), the method involves therefore a rapidly increasing and, in fact, dominating amount of data communication overhead. Hence such methods must be implemented with special care to enable an efficient use of the parallel machine.

What has been said so far can be illustrated in a figure:

Table: Computer time versus problemsize for a fixed number (p) of processors

Remark 2.1 As it turns out, the break-even point N_0 can be taken even for not excessively large problem sizes. If $T_p = \frac{1}{p} C_p N^\beta$, one finds $N_0 = \left(\frac{C_0 p}{C_p}\right)^{1/\beta - 1}$.

Example 2.1 Assume that $\beta = 1 + \frac{1}{d}$ (d = dimension), and $\frac{C_0}{C_p} = \varepsilon^{\frac{1}{2}}$ then

$$N_0^{1/d} = \frac{C_0 p}{C_p} \sim \varepsilon^{\frac{1}{2}} p \quad \text{or} \quad N_0 \simeq (\varepsilon^{\frac{1}{2}} p)^d.$$

Here ε is a small problem (singular perturbation) parameter.

Example 2.2 Consider the equation $a\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f$ in the square $-1 \leq x, y \leq 1$ where $a = a_0$ in the square $-\frac{1}{2} \leq x, y \leq \frac{1}{2}$ and $a = 1$ elsewhere, and where $u = 0$ on 128×128 grid the following computer times were found

Computer	Algorightm	Computer time(sec)	Clock cycle
CM-200 8 K bit = 256 32 bit processors	Diagonally pre- conditioned CG	3.6	10 MHz
IBM RS/6000 ($p = 1$)	Multilevel V-cycle	2.7	33 MHz (corresp, to 66 MHz because of vector unit)

Table 1. Computer times on a $p = 256$ parallel machine and on a single processor machine

Clearly, it is pointless to employ numerically inefficient algorithms merely to exhibit artificially high performance rates on a particular parallel machine using algorithms which are inefficient on a serial computer.
It can be seen that any algorithm with an optimal order of computational complexity $O(N)$ requires some form of global communication such as occurs in the multilevel iteration methods.

Consider now the use of the multilevel iteration method on the massively parallel computer.

Coeff.		No Preconditioning		Diag. Preconditioning		AMLI Preconditioning	
jump	Grid	Iter.	CPU time	Iter.	CPU time	Iter.	CPU time
$1\backslash 0.75$	128^2	494	2.96	480	2.93	21	2.98
$1\backslash 0.1$	128^2	—	—	475	2.91	22	3.01
$1\backslash 0.001$	128^2	5538	31.26	505	3.07	25	3.60
	256^2	—	—	1027	14.05	31	16.53
	512^2	—	—	2090	81.15	40	85.10

Table 2. Number of iterations and computer times on CM-200 for two "embarrassingly parallelizable" algorithms and for a multilevel iteration method for example 2.

The large computer time for the (close to) optimal order algorithm is due to the cominating communication time.
Therefore, the dilemma:

- fast convergence $\Longrightarrow$ much communication overhead
- little communication overhead $\Longrightarrow$ slow convergence

may seem impossible to overcome.
However, as shall be seen in the next section, it is curable and using a proper algorithm one can achieve *scalable algorithms*. Here we use scalable in the following sense.

Definition 2.1 (a) An algorithm A is said tohave an *optimal order parallel complexity* of shortly to be scalable, if its efficiency

$$E(A,p) = \frac{T(A^*,1)}{P^T(A,p)},$$

approaches its *optimal value of one* for increasing N and p increasing not faster than $p \leq p^*(N)$ as some properly increasing function of N.
(b) The algorithm A is said to be scalable on a particular parallel architecture if it has an optimal order of parallel complexity.

As it turns out one can not expect scalable algorithms if one considers increasing number of processors but a fixed problem size. In fact, frequently it makes sence to consider the situation where p increases slower than N.

3 A scalable implementation of a multilevel iteration method, with some numerical testresults

Fortunately there exists a cure for the "rate of convergence $\longleftrightarrow$ dominating communication overhead" conflict.
It is based on the following simple idea: Use as few levels as possible and solve the final coarse mesh problem with an "embarrassingly parallelizable" method. The number of levels used should be a balance between the computational complexities on the finest level and the *total* complexity (per V-cycle iteration) on the coarsest level.

The algebraic multilevel iteration method
The AMLI methods belong to the class of multilevel solution methods and fits into the general framework of block-incomplete factorization methods for constructing a preconditioner to a given matrix. Two main forms of such a preconditioner are the *block-diagonal* or *additive* form and *full block factorized* or *multiplicative* form.
The hierarchical basis function preconditioner is an example of the first type and the AMLI methods by [Ax, Vass, Eijkhout, Neytcheva, and others] are example of the second type.
To give a flavour of the construction of the AMLI type preconditioner, we give here a brief sketch of the method, when applied to linear selfadjoint second order elliptic partial differential equations.
Let $Ax = b$ be an algebraic system derived by differences or finite elements from the differential equation. We want to find a preconditioner for A to be used in a conjugate gradient method.
AMLI is a recursive procedure for constructing a preconditioner M. It replaces the solution of a system with the preconditioning matrix by a sequence of subproblems to be solved on some hierarchy of levels. The levels are based on a certain decomposition of the matrix graph $-\Omega(N,S)$ of $A = [a_{ij}]_{i,j=1}^n$ where N is the corresponding set of nodes (vertices) and S is the set of edges (i,j), such that $(i,j) \in S$ if and only if $a_{ij} \neq 0$. In particular, going upwards the levels may be related to levels of consecutive grid *refinements* of the discretized

problem. A matrix $A^{(k)}$ is associated with each level k.
The desired spectral equivalence of the preconditioner with the original matrix is achieved by using matrix polynomials of special type which recursively connect pairs of adjacent levels. Upper and lower bounds for the degree of the polynomials are derived as conditions for optimal-order computational complexity and optimal rate of convergence, respectively.
A multilevel iteration method which is based on the above idea is the short version of the algebraic multilevel iteration method [Axelsson, Neytcheva] which is described in detail in [ ].

Some numerical tests using this method are reported below.
Being a V-cycle algorithm, the above method is not of optimal order of computational complexity, since the number of iterations grows, albeit slowly, with increasing problem size.
However, this growth can be cured to any order arbitrarily close to the optimal order method using one of the following methods.
a) Polynomial acceleration as used in [Ax,Vass] can be applied at regular intervals of levels in the V-cycle. However, this increases the recursive overhead and seems not be competitive, for problem sizes one encounters in practice, with the pure V-cycle method.
b) Stabilization of the V-cycle method can be done using the same type of method as a coarse solver. This has two effects. First, it allows one to chose the finest and coarsest levels even closer. Secondly, the condition number decreases and therefore also the number of iterations and the total computational complexity. This method can be repeated recursively more times and in this way the condition number becomes arbitrarily close to $O(1)$, for further details, see [Ax95].

The above short level version of the AMLI method has been applied in several more general contexts such as for nonsymmetric, indefinite, eigenvalue, and nonlinear problems, see [Neytchev]. It has been implemented and tested already on various supercomputers or massively parallel computers, such as the CM-2, CM-200, CM-5, Cray C98, and Cray T3D. For the latter machine the following results were found.

Remark 3.1
• The time in the table includes also the construction of the AMLI preconditioner which for the biggest test problem has been done within 3 *sec* on 128 PEs
• The performance of the present implementation can be improved at least with 40% by reducing data caches.
It is seen that the method scales very nicely.

There are several advantages in using a short level version of the AMLI and similar multilevel iteration methods as compared to a full length version.
• the recursive loop and communication overhead is less
• the condition number is smaller, in particular for the V-cycle version (the work per V-cycle may be somewhat larger, but this can be controlled as we shall show below)
• the performance of the method is generally more robust with respect to various problem parameters.

No. of unknowns Total *no.* of levels	Coarse level *no.*	*No.* of outer iter.	Average *no.* of inner iter.	Total execution time on *no.* of PEs (in *sec*)				
				16	32	64	128	256
$(2^5)^3 = 32768$ 13 levels	8	50	8	6	-	-	-	-
	9	39	13	5	-	-	-	-
	10	29	18	4	2	1	-	-
	11	29	29	6	-	-	-	
$(2^6)^3 = 262144$ 16 levels	9	57	7	54	-	-	-	-
	10	$\boxed{49}$	10	49	25	13	7	-
	11	-	-	>90	-	-	-	-
$(2^7)^3 = 2097152$ 19 levels	12	64	13	-	-	-	76	-
	13	$\boxed{50}$	19	-	-	-	67	35
	14	54	30	-	-	-	91	-

Table 4. Number of inner and outer iterations and computer execution times for a 3D code of the short AMLI method, using a trilinear (27-point) stiffness matrix, preconditioned by finite difference (7-point) matrix, ratio between degrees of freedom on sequitive levels $N_{n+1}/N_k = 2$.

The stabilization property of the short AMLI method will now shortly be discussed, for more details, see [ ].
The condition number of the V-cycle of the AMLI method increases with the distance between the finest level number (l) and the coarsest (k_0) as

$$\mathrm{cond}(C^{l,k_0)^{-1}}, A^{(l)}) \leq C_0 \sigma^{l-k_0}$$

for some $\sigma > 1$ and constant C_0. However, the cost per iteration step increases with k_0 and we shall seek the value of k_0/l for which the total cost during all iteratiopns is minimized. To this end we must first find the computational cost for one iteration step.
This is readily found to be

$$w_l \leq C(n_l + n_{l-1} + \cdots + n_{k_0+1}) + C_{k_0} n_{k_0},$$

where n_s denotes the degree of freedom at level s and the constant C depends only on sparsity of the matrices involved. C_{k_0} depends on the solution method used when solving systems on the coarsest mesh level. For the diagonally preconditioned conjugate gradient method it is $C_{k_0} = K n_{k_0}^{1/d}$ where d is the space dimension of the differential equation problem.
For recursive red-block orderings we have

$$n_s = c 2^s, \quad s = l,\ l-1, \ldots, k_0$$

for some constant c, which we can take $= 1$.
We find then

$$\begin{aligned} \frac{W_l}{n_l} &\leq 2C(1 - 2^{k_0 - l}) + K n_{k_0}^{1+1/d}/n_l \\ &= 2C(1 - 2^{k_0 - l}) + K 2^{k_0 - l} \cdot 2^{k_0/d}. \end{aligned}$$

In general, the conjugate gradient method needs $\widetilde{C}\sigma^{(l-k_0)/2}$ iterations for convergence, so the total cost becomes

$$\frac{W_l}{n_l} \le \widetilde{C}2^{(l-k_0)\xi}[2C + K2^{k_0-l+\frac{k_0}{d}}]$$

where $\xi = \frac{1}{2}\log_2\sigma$. An elementary computation shows that this is minimized for

$$k_0 = \frac{d}{d+1}l - |0(1)|, \quad l \to \infty,$$

which is essentially independent on the condition number factor σ.
The corresponding minimal computational cost (per node point) is

$$\frac{W_l}{n_l} \le \text{const}2^{\frac{1}{2}\cdot\frac{l}{d+1}\log_2\sigma} = O\left(N^{\frac{1}{2(d+1)}\log_2\sigma}\right).$$

If a full length V-cycle method had been used the result would have been $W_l/n_l \le O(N^{\frac{1}{2}\log_2\sigma})$.
The stabilizing factor of the short level AMLI method is hence $\frac{1}{d+1}$ and takes the values $\frac{1}{3}$ and $\frac{1}{4}$, for 2D and 3D problems, respectively. The corresponding values of $\frac{k_0}{e}$ are $\frac{2}{3}$ and $\frac{3}{4}$, respectively. Hence we see that in practice, for the optimal value of the coarse mesh level the coarse mesh is very close to the fine mesh.
The rest runs presented in Table 4 nicely illustrate the above rsult. They show also the close to optimal scalability of the method when implemented on a parallel machine with up to 256 processors.
So far, only the computational complexity of the AMLI method has been analysed. What remains is to analyse the communication complexity, which, as we have seen, depends on the number of processors used, and to find a proper balance of the problem size and number of prcessors.

As an introduction to this topic, consider the diagonally preconditioned conjugate gradient method to solve an elliptic difference equation problem, implemented on a cube $(\sqrt[3]{p}\times\sqrt[3]{p}\times\sqrt[3]{p})$ mesh array computer. Here the computer time per iteration on p processors is

$$T_p = \frac{T_1}{p} + C\sqrt[3]{p}$$

where the first term contains the arithmetic computations and the second term is the communication time necessary for the computation of the inner products in the conjugate gradient method (the time for nearest neighbor communication can be included in the first term).
Obviously, there exists a number of processor p^* which minimizes T_p and one finds

$$p^* = \left(\frac{3T_1}{c}\right)^{\frac{3}{4}}, \quad T_{p^*} = \frac{4}{3^{\frac{3}{4}}}\left(c^3T_1\right)^{\frac{1}{4}}, \quad E_{p^*} = \frac{1}{4}.$$

The efficiency is hence low. If one lets $p = \mu p^*$, for some $\mu \le 1$, one finds

$$T_p = \frac{1}{\mu}\left(1 + 3\mu^{\frac{4}{3}}\right)T_{p^*}; \quad E_{p^*} = \frac{1}{1+3\mu^{\frac{4}{3}}}.$$

Note that if μ is fixed the efficiency does not depend on problemsize nor on the number of processors. If we let μ decrease slowly with T_1, such as $\mu = 1/logT_1$, then

$$E_p \to 1 \quad \text{as} \quad N \to \infty,$$

i.e., for all $p \leq p^*/\log T_1$, we have asymptotically an optimal efficiency (cf. Definition 2.1). For the above method, the number iterations grows also with $T_1 = O(N)$, namely as $O(N^{\frac{1}{3}})$, so the total computational complexity grows at least as $O(N^{\frac{1}{3}} + \frac{1}{4}) = O(N^{\frac{7}{12}})$, irrespective of the number of processors used.
Hence, even if as many as $O(N^{\frac{3}{4}})$ processors are used, the computer time will grow rapidly, and not very much has been gainced compared to using an optimal order algorithm on a single processor.
Consider now the AMLI algorithm. Here on the same 3D mesh array computer one finds the computer time per iteration step

$$T_p = \frac{T_1}{p} + w_1 it_{in} \sqrt[3]{p} + w_2 it_{in} \sqrt[3]{\frac{N_{k_0}}{p}} + w_3 \sqrt[3]{p} + w_4 \sqrt[3]{\frac{N}{p}},$$

where w_i, $i = 1, 2, 3, 4$ are constants which depend on the communication rate. Here it_{in} is the average number of inner iterations on the coarsest mesh performed per outer iteration, and the additional terms arise from nearest neighbor and global communications. We assume that $p \leq N_{k_0}$ (i.e., on each processor, where $m = \sqrt[3]{\frac{N_{k_0}}{p}} \geq 1$) and that $it_{in} = O(N_{k_0}^{\frac{1}{d}})$. Then

$$T_p = \frac{T_1}{p} + w_1' N_{k_0}^{\frac{1}{d}} \sqrt[3]{p} + w_2' N_{k_0}^{\frac{1}{3}+\frac{1}{d}} / \sqrt[3]{p}.$$

We let N_{k_0} be determined for smallest computational cost and let p be chosen to minimize T_p. We find then asymptotically

$$p^* \simeq \left(\frac{3}{w_1'} \frac{T_1}{N_{k_0}^{\frac{1}{d}}} \right)^{\frac{3}{4}} \simeq O(N^{\frac{9}{16}}), \quad N \to \infty \tag{3.1}$$

and

$$T_{p^*} = O(N^{\frac{7}{16}}).$$

As we have seen previously, the number of iterations grow like $O\left(N^{\frac{1}{2}\frac{1}{d+1}\log_2 \sigma}\right)$, so the total computer time is

$$\widehat{T}_p \equiv O\left(N^{\frac{7}{16}+\frac{1}{8}\log_2 \sigma}\right).$$

Typically, $\sigma \leq 2$, so the total computer time grows as

$$\widetilde{T}_p \leq O(N^{\frac{9}{16}}).$$

This is almost as much as for the diagonally preconditioned method but for the short AMLI method we need much fewer processors $O(N^{\frac{9}{16}})$ instead of $O(N^{\frac{3}{4}})$.
As for the diagonally preconditioned method the efficiency is not optimal but we can used a slightly slower growing number of processor than p^* in (3.1) for which the efficiency approaches its optimal value 1 asymptotically.

4

5 On dynamic load balancing for adaptive refinement methods

When solving unstructured mesh problems a proper mapping of node-points to processors becomes an essential part of an efficient algorithm for parallel processors. When adaptive refinement methods are used this mapping must be applied dynamically during the solution process, i.e., one must use remappings.
The mapping should distribute the computational load fairly evenly among the processors and in such a way that *similar communication* patterns, or data localities, are seen from all processors.

Ultimately, the task of a load balancing algorithm is to find a mapping $q : \mathcal{N} \to \mathcal{P}$, where $\mathcal{N}, \mathcal{P}$ are the set of nodpoints and set of processors, respectively, in such a way that the computing time is minimized. This is a very complex problem which involves many parameters (hardware, software, method -for instance, the mapping may influence even the rate of convergence of the iteration method) and one must settle for a less ambitious task, such as finding in some heuristic way a mapping which nearly achieves this goal.
Even if one restricts the problem to minimizing the computing time for a single V-cycle one must in general use some discrete optimization method. General algorithms, such as simulated annealing [ ] or genetic algorithms [ ] exist, but require normally more time than the solver itself and seem therefore to be out of question to use.

Much effort has recently been devoted to algorithms for use on *a single unstructured grid*. The goal has here been to partition the vertices of the matrix graph (or, equivalently, of the mesh) into p equal subsets, while minimizing the numbers of inter-subset edges (which latter determines the amount of interprocessor communication required).
A popular method has been to use a *divide and conquer* approach which reduces the graph mapping problem to a graph bisection problem, which is applied recursively to obtain 2^s partitionings after s steps.
The bisection strategies range from methods based on geometric information of the grid [Miller et al] to methods using the eigenvector corresponding to the second smallest eigenvalue of the Boolean or Laplacian matrix derived from the connectivities of the graph [Pothen et al].
However, in the context of multilevel and hierarchically determined meshes these methods may be less efficient for the following reasons:

(i) The recursions between the matrices on different levels when a multiplicative version of multilevel method is used require some further inter-grid dependencies, which are not considered in the bisection strategies.

(ii) The computational complexity of the multilevel methods is normally relatively small per node-point so the overhead caused by the mesh-partitioning method can easily dominate the total computing times.

(iii) In the context of dynamic load balancing the mapping method does not take the current mapping into account in order to restrict the number of elements that must be moved from one processor to another.

Therefore, it seems that an efficient dynamic load balancing algorithm must be closer connected to the evolution of the solution algorithm itself and not used separately from this. Another related issue is the overhead caused by the more complicated code necessary for unstructured meshes. This code must be duplicated on each processor and the fact that each subset of the mesh can vary makes it necessary to use MIMD type machines.
It may therefore turn out that for many problems a uniformly fine regular mesh coupled with a fictitious domain method and implemented with a much simpler structured mesh code possibly on a (much cheaper) SIMD type machine can be a competitive approach.
In some problems, there is already much information available about the appearence of layers in the solution, and, hence where there is a need to use finer meshes. For instance, for convection dominated convection diffusion flow problems one can partition the mesh along the flow directions. In this way, the domains get only weakly coupled and even a simple Jacobi block preconditioned method may converge fast.

Regarding the choice between *additive* (or block-diagonal) preconditioners and *multiplicative* the following general remakrs can be made.

The additive methods, coupled with stabilization by solving coarse mesh problems at some levels (typically in total, one one or two) may turn out to be competitive on parallel computers because
- it simplifies the way the grid hierarchy ar mapped onto the processors
- it allows partitionings with much lower inter- and intra-subgrid communication.
Therefore, *parallel* efficiences are usually higher and scalability is better.

However, the gain may not always compensate for the worse *numerical* efficiency as compared to the multiplicative methods as have been used in this report.
As it may turn out, some other versions of additive methods with better spectral equivalence properties must be used and analysed before a final recommendation can be made.
Acknowledgement. The help provided by at Cray Research in was very valuable when improving the performance of our first implementations of the AMLI method on the Cray T3D.

References

[1] O. Axelsson, M. Neytcheva, Parallel implementations of the Algebraic Multilevel Iteration Method. In J.G. Lewis (ed.), *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra*, SIAM Philadelphis, 1994, 372-376.

[2] O. Axelsson, M. Neytcheva, Scalable Parallel Algorithms in CFD Computations. In M. Hafez and K. Oshima (eds.), *Computational Fluid Dynamics Review* 1995, 837-857.

[3] D.H. Bailey, Misleading Performance Reporting in the Supercomputing Field, *Scientific Progr.* **1** (1992), 141-151.

[4] J.H. Bramble, J.E. Pasciak, J. Xu, Parallel Multilevel Preconditioners, *Math. Comput.*, **55** (1990), 1-22.

[5] T.F. Chan, Hierarchical Algorithms and Architectures for Parallel Scientific Computing. In *Proceedings of Supercomputing '90, ACM*, 1990, 318-329.

[6] P. Deuflhard, P. Leinen, H. Yserentant, Concepts of an Adaptive Hierarchical Finite Element Code, *IMPACT of Computing in Science and Engineering*, **1** (1989), 3-35.

[7] H.C. Elman and X.-Z. Guo, Performance Enhancements and Parallel Algorithms for Two Multilevel Preconditioners, *SIAM J. Sci. Stat. Comput.*, **14** (1993), 890-913.

[8] H.P. Flatt and K. Kennedy,Performance of Parallel Processors, *Parallel Comput*, **12** (1989), 1-20.

[9] A. Grama, A. Gupta and V. Kumar, Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures. Technical report TR 93-24, April 1993, Computer Science Department, University of Minnesota.

[10] B. Hendrickson, R. Leland, An Improved Spectral Load Balancing Method, *Proc. of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, March, 1993.

[11] D. Hodgson, P. Jimack, Efficient Mesh Partitioning for Parallel PDE Solvers on Distributed Memory Machines, *Proc. of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, March, 1993.

[12] V. Kumar and A. Gupta, Analyzing Scalability of Parallel Algorithms and Architectures, *J. Parallel Distrib. Comp.*, **22** (1994), 379-391.

[13] L. Matheson and R. Tarjan, A Critical Analysis of Multigrid Methods on Massively Parallel Computers, Report Princetoncs: TR-448-94, 1994. Analysis of multigrid methods on miassively parallel computers: architectural implications. In N.D. Melson, T. Manteuffel and S.F. McCormick (eds.), *Proceedings of the Sixth Copper Mountain Conference on Multigrid Methods*, NASA, 1993, 405-421.

[14] H. Mühlenbein, M. Gorges-Schleuter, O. Krämer, New Solutions to the Mapping Problem of Parallel Systems: The Evolution Approach, *Parallel Computing*, **4** (1987), 269-279.

[15] M. Neytcheva, Arithmetic and communication complexity of preconditioning methods, Ph.D. Thesis, University of Nijmegen, 1993.

[16] A. Pothen, H. Simon, K. Liou, Partitioning Sparse Matrices with Eigenvectors of Graphs, *SIAM J. Matrix Anal. Appl.*, **11** (1990), 430-452.

[17] H. Simon, Partitioning of Unstructured Problems for Parallel Processing, *Computing Systems in Engineering*, **2** (2/3) (1991), 135-148.

[18] X.-H. Sun and D.T. Rover, Scalability of Parallel Algorithm-Machine Combinations, *IEEE Trans. Parallel Distrib. Systems.* To appear. [Technical report IS-5057, 1991, Ames Laboratory, Iowa State University.]

[19] O. Tatebe and Y. Oyanagi, Efficient Implementation of the Multigrid Preconditioned Conjugate Gradient Method on Distributed Memory Machines. In *Proceedings of PCG'94, Matrix Analysis and Parallel Computing*, Keio University, March 14-16, 1994, 113-125.

[20] R.D. Williams, Performance of dynamic road balancing algorithms for unstructured mesh calculations, Report C3P913, Caltec, Pasadena, CA.

[21] P.H. Worley, Limits on Parallelism in the Numerical Solution of Linear Partial Differential Equations, *SIAM J. Sci. Stat. Comput.*, **12** (1991), 1-35.