CWI Quarterly Volume 9 (1 & 2) 1996, pp. 9 – 17

Some Basic Facts for Efficient Massively Parallel Computation

Owe Axelsson and M.G. Neytcheva[1]
Department of Mathematics, University of Nijmegen
Toernooiveld, 6525 ED Nijmegen
The Netherlands

Four fundamental aspects of efficient massively parallel computation 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 computing times must eventually increase with problemsize, no matter how many processors are used. The remedy to this situation in scientific computing is to limit the problem size by use of adaptive refinement methods. Some aspects of dynamic load balancing for adaptive refinement methods will be mentioned.

[1] Supported in part by NCF/Cray Research under grant no CRG 96.28.

1. Introduction

The computational complexity of the solution of a problem can grow very fast with increasing problem size. To illustrate this, consider a time dependent PDE in 3D. Assuming that one uses a uniform partitioning in time and space, the number of (unknown) parameters to be computed increases as $\sim cn^4$, $n \to \infty$, for some constant c, where both the timestep and the meshwidth $= O(n^{-1})$. Assume, for simplicity, that for a given stepsize $h_0 = n^{-1}$, the computer time is 1 second and assume now that we must resolve some boundary or interior layers of the solution and for this purpose we divide the meshsizes with a factor 10 in each time-space direction then the complexity becomes $\sim c10^4n^4$.
In the most ideal case the solver requires only the same amount of 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. 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. Therefore, 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. Hence, one must use algorithms for which the performance increases linearly with the number of processors. Some reasons why massively parallel scientific computation hasn't been a great success yet, are identified and a solution is suggested. The present paper is a short version of [3].

2. Megaflop rates contra computing times; or nearest neighbor contra global communication

The above, already indicated as a serious 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* avoid 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 exist (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: $x^{k+1} = x^k - \tau(Ax^k - b)$. Here τ is a damping parameter chosen to get convergence of the iterations (possibly $\tau = \tau_k$ is variable). The classical conjugate gradient method has a similar form.

In such cases the above iterations can be executed extremely fast on parallel computers because the need for exchange of data is relatively 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 '*embarrassingly parallelizable*'.

However, what obviously matters for a user is not the megaflop rate but the computer time. 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 $O(n^2 N)$ or, at least as $O(nN)$, $n \to \infty$, where N is the total number of meshpoints.

Nowadays 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 and 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. This is illustrated in Tables 1 and 2, where first a comparison is made of an 'embarrassingly parallelizable' method run on a massively parallel computer and an optimal method run on a single processor.

Example 2.1.
Consider the equation $a\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f$ in the square $0 \le x, y \le 1$ where $a = a_0$ in the square $-\frac{1}{4} \le x, y \le \frac{3}{4}$ and $a = 1$ elsewhere, and where $u = 0$ on the boundary of the first square. Then on a 128×128 grid the computer times as shown in Table 1 were found.

Computer	Algorithm	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$^{1)}$

TABLE 1. Computer times on a $p = 256$ parallel machine and on a single processor machine.$^{1)}$ Peak performance = 66 Mflops.

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, (cf [6]).

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. In Table 2 we consider now the use of the multilevel iteration method on a massively parallel computer.

Coeff.		No Preconditioning		Diag. Preconditioning		AMLI Preconditioning	
jump	Grid	Iter.	CPU time	Iter.	CPU time	Iter.	CPU time
1\0.75	128^2	494	2.96	480	2.93	21	2.98
1\0.1	128^2	—	—	475	2.91	22	3.01
1\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 2.1.

The large computer time for the (close to) optimal order algorithm is due to the dominating 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 *scalability*. Here we use scalable in the following sense.

DEFINITION 2.1.
(a) An algorithm A is said to have an *optimal order parallel complexity* or shortly to be *scalable*, if its efficiency $E(A,p) = T(A^*,1)/pT(A,p)$ (A^* denotes the optimal algorithm on a single processor) approaches its *optimal value of one* for increasing N and for 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.

For further discussions on scalability issues , see [?]1 & 21996 and the references quoted therein. As it turns out one can not expect to see the effect of scalability if one considers an increasing number of processors but a fixed problem size. In fact, frequently it makes sense 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 complexity (per V-cycle iteration) on the coarsest level, in order to minimize the total computer time.

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 [5], [2], [4] 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 for A. 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 preconditioner is implicitly defined. An action of its inverse on a vector requires only matrix vector operations and acts on the different levels depending on some matrix polynomial. The so-called V-cycle corresponds to a polynomial of degree one.

A short version of the AMLI method, which is described in detail in [10] turns out to be an improvement of the classical method and is specially suitable for parallel implementations. Some numerical tests using this method are reported in Table 3. 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 using one of the methods as described in [1].

The short level version of the AMLI method has been applied in various more general contexts such as for nonsymmetric, indefinite, and eigenvalue problems, see [10]. 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 results in Table 3 were found.

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
	8	50	8	6	-	-	-	-
$\left(2^5\right)^3 = 32768$	9	39	13	5	-	-	-	-
13 levels	**10**	29	18	4	2	1	-	-
	11	29	29	6	-	-	-	
	9	57	7	54	-	-	-	-
$\left(2^6\right)^3 = 262144$	**10**	$\boxed{49}$	10	49	25	13	7	-
16 levels	11	-	-	>90	-	-	-	-
	12	64	13	-	-	-	76	-
$\left(2^7\right)^3 = 2097152$	**13**	$\boxed{50}$	19	-	-	-	67	35
19 levels	14	54	30	-	-	-	91	-

TABLE 3. 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 consecutive levels $N_{k+1}/N_k = 2$.

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 small;
- 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 has been shown in [1] and [3];
- the performance of the method is generally more robust with respect to various problem parameters.

The stabilization property of the short AMLI method is shown in [1] and it turns out that the corresponding minimal computational cost (per node point) is

$$\frac{W_l}{n_l} \leq \text{const}2^{\frac{1}{2}\cdot\frac{l}{d+1}\log_2\sigma} = O\left(N^{\frac{1}{2(d+1)}\log_2\sigma}\right), d = 2 \text{ or } 3,$$

when a short level version with coarsest level $k_0 = \frac{d}{d+1}l - |O(1)|, \quad l \to \infty$, is used. Here σ is the factor in the condition numbers of the preconditioner matrix relating two adjacent levels and $N = n_l$.
If a full length V-cycle method had been used the result would have been $W_l/n_l \leq O(N^{\frac{1}{2}\log_2\sigma})$. 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 asymtotic values of $\frac{k_0}{l}$ 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 runs presented in Table 3 nicely illustrate the above result. 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 processors.

On a 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 $p \leq N_{k_0}$ and that $it_{in} = O(N_{k_0}^{\frac{1}{d}})$. The latter holds for the conjugate gradient method, for instance. We let N_{k_0} be determined for smallest computational cost and let p be chosen to minimize T_p. We find then asymptotically

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

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, $\sigma \leq 2$, in which case the total computer time grows as $\widehat{T}_p \leq O(N^{\frac{9}{16}})$.

4. Physical limits on parallelism

For a long time the increase of the performance of computers was done by increasing clock speed. However, due to physical constraints such as speed of light and wave reflections one has now come close to the ultimate bound when metal interconnections are used. Also, any packing of processor elements in

real physical (i.e. 2 or 3) space means that there is a growth of the total volume occupied by wires and eventually processors become to wires as needles in a haystack. This translates into a corresponding barrier for the clock cycle. For more details see [13] and [3], and the references quoted therein.
Optical interconnections might be a solution in the future machines both for interchip and interboard connections, see [7] for a discussion. However, even they will have limits due to fundamental physical laws. Therefore, the construction of computer systems with an 'infinitely' large number of processor is impossible.

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, cf. [12].

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) and one must settle for a less ambitious task, such as finding in some heuristic way a mapping which nearly achieves this goal.

One must in general use some discrete optimization method. General algorithms, such as simulated annealing [12] or genetic algorithms [9] exist, but require normally more time than the solver itself and seem therefore to be out of question to use in the present context.

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 [8] to methods using the eigenvector corresponding to the second smallest eigenvalue of the Boolean or Laplacian matrix derived from the connectivities of the graph [11].

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.

References

1. O. Axelsson (1996). The stabilized V-cycle method, *TICAM Conference Proceedings*, editor J.T. Oden, Wiley, to appear.
2. O. Axelsson, V. Eijkhout (1991). The nested recursive two-level factorization method for nine-point difference matrices, *SIAM J. Scientific and Statistical Computations*, **12**, 1373–1400.
3. O. Axelsson, M. Neytcheva (1996). Some basic facts for efficient massively parallel computation, Report 9607, University of Nijmegen.
4. O. Axelsson, M. Neytcheva (1994). Algebraic multilevel iteration method for Stieltjes matrices, *Numer. Linear Algebra with Applications*, **1**, 213–236.
5. O. Axelsson, P.S. Vassilevski (1990). Algebraic multilevel preconditioning methods II, *SIAM J. Numer. Anal.*, **27**, 1569–1590.
6. D.H. Bailey (1992). Misleading Performance Reporting in the Supercomputing Field, *Scientific Progr.* **1**, 141–151.
7. A. Louri, H. Sung (1994). 3D Optimal Interconnectors for High-Speed Interchip and Interboard Communications, *Computer*, 27–37.
8. Gary L. Miller, Stephen A. Vavasis (1991). A unified geometric approach to graph separators. In *32th Annual Symposiium on Foundations of Computer Science*, pp. 538–547, Puerto Rico, IEEE.
9. H. Mühlenbein, M. Gorges-Schleuter, O. Krämer (1987). New Solutions to the Mapping Problem of Parallel Systems: The Evolution Approach, *Parallel Computing*, **4**, 269–279.
10. M. Neytcheva (1995). *Arithmetic and communication complexity of preconditioning methods*, Ph.D. Thesis, University of Nijmegen.
11. A. Pothen, H. Simon, K. Liou (1990). Partitioning Sparse Matrices with Eigenvectors of Graphs, *SIAM J. Matrix Anal. Appl.*, **11**, 430–452.
12. R.D. Williams. *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Report C3P913, Caltec, Pasadena, CA.
13. P.H. Worley (1991). Limits on Parallelism in the Numerical Solution of Linear Partial Differential Equations, *SIAM J. Sci. Stat. Comput.*, **12**, 1–35.