Parallel Computing 19 (1993) 209–218
North-Holland

PARCO 754

A multi-level diffusion method for dynamic load balancing

G. Horton

Lehrstuhl für Rechnerstrukturen, Universität Erlangen-Nürnberg, Martensstr. 3, 8520 Erlangen, Federal Republic of Germany

Received 11 March 1992

Abstract

Horton, G., A multi-level diffusion method for dynamic load balancing, Parallel Computing 19 (1993) 209–218.

We consider the problem of dynamic load balancing for multiprocessors, for which a typical application is a parallel finite element solution method using non-structured grids and adaptive grid refinement. This type of application requires communication between the subproblems which arises from the interdependencies in the data. A load balancing algorithm should ideally not make any assumptions about the physical topology of the parallel machine. Further requirements are that the procedure should be both fast and accurate. An new multi-level algorithm is presented for solving the dynamic load balancing problem which has these properties and whose parallel complexity is logarithmic in the number of processors used in the computation.

Keywords. Dynamic load balancing; parallel computing; distributed-memory multiprocessor; multi-level algorithm.

1. Introduction

Load balancing is one of the central problems which have to be solved in parallel computation. Since load imbalance leads directly to processor idle times, high efficiency can only be achieved if the computational load is evenly balanced among the processors.

Two kinds of load balancing can be distinguished, static and dynamic. *Static* load balancing is used when the computational requirements of a problem are known a priori and do not change during the course of the calculation. In this case it is sufficient to decompose the problem once before the parallel application is run. For simple problems this may often be done manually; more complex cases are solved with heuristic methods. Typical approaches are based on recursive bisection of the data according to geometric or other criteria, or simulated-annealing like techniques. A comparison of typical load balancing techniques has been performed by Williams in [5]. The results show that performing the load balance may take a significant amount of computation time.

Problems whose load changes during the computation will necessitate the redistribution of the data in order to retain efficiency. Such a strategy is known as *dynamic* load balancing. One typical example of a multiprocessor application requiring dynamic load balancing is the parallel solution of a partial differential equation (pde) by finite elements on an unstructured

Correspondence to: G. Horton, Lehrstuhl für Rechnerstrukturen, Universität Erlangen-Nürnberg, Martensstr. 3, 8520 Erlangen, Federal Republic of Germany, email: graham@informatik.uni-erlangen.de

grid with adaptive refinement. Such a computation starts with an initial, often relatively coarse grid, and computes a solution to the problem. Using an estimate for the discretization error the grid is refined locally in areas where the error exceeds some predefined cirterion. A new solution is then computed on the refined grid. Since the grid refinement is local, arising for example near corners or singularities in the solution, some processors will experience an increase in load, which, if not redistributed, will lead to a (possibly serious) loss of parallel efficiency.

If the changes in the load are small relative to the total load, the dynamic load balancer may work with the given problem topology, restricting itself to shifting data between the processors, the assumption being that the previous distribution of work is a good approximation to the optimal new load distribution. If, on the other hand, the load changes are large, then it may be necessary to completely re-solve the load balancing problem from scratch. In this case, a static load balancing scheme must be used.

One simple, parallel method for dynamic load balancing is for each processor to transfer an amount of work to each of its neighbours which is proportional to the load difference between them. Since this approach will not, in general, provide a balanced solution immediately, the process is iterated until the load difference between any two processors is smaller than a specified value. Such methods correspond closely to simple iterative methods for the solution of diffusion problems; indeed, the surplus load can be interpreted as diffusing through the processors towards a steady balanced state. Diffusion methods have, however, two disadvantages which result from the local nature of the transfer of information and of load.

Firstly, the number of iterations required by the load balancer may be high, making the algorithm too expensive to use. Boillat has analysed diffusive methods in [1], obtaining rates of convergence for various machine topologies. He shows the number of iterations in several cases to be of the form $O(n^2)$ where n is the number of processors. In any particular situation, the number of iterations needed to achieve a balanced load depends on the initial load imbalance, and is not known a priori. In the ideal case, however, the algorithm should provide a balanced load after a small, fixed number of steps.

The second problem of diffusion methods (and in contrast to genuine diffusion problems), is that since the work is packaged into discrete units, the algorithm may produce solutions which, although they are locally balanced, prove to be globally unbalanced. Table 1 shows the work loads of 6 processors p_i, $i = 0,\ldots,5$, connected in a linear array, which are obtained as balanced solutions with two diffusion-based algorithms. In the case of the original diffusion scheme (*Diff.*), although the load of each processor differs by only at most one unit from that of each of its neighbours, the global load balance is very poor. In the Help-thy-Neighbour method (*HtN*) each processor tries to equalize the load between two neighbouring processors, rather than between itself and one neighbour. Although the situation is slightly improved, processor p_0 still has twice the load of p_5. One optimal solution is given by *Opt.*, in which the maximum load difference *any* two processors is one unit. Any load balancing method based on the pairwise comparison of loads on neighbouring processors will not be

Table 1
Solutions to the load balancing problem

	p_0	p_1	p_2	p_3	p_4	p_5
Diff.	7	6	5	4	3	2
HtN	6	5	5	4	4	3
Opt.	5	5	5	4	4	4

able to recognize solutions such as *Diff.* and *HtN* as unbalanced. This is evidently unsatisfactory.

Cybenko has presented a dimension-oriented dynamic load balancer for hypercube multiprocessors in [2], which is shown to require ld(n) steps, where n is the number of processors and ld denotes the logarithm to base 2. The method utilizes the topology of the hypercube machine for its efficiency, but ignores any dependencies between the individual items of data moved. The algorithm is thus well suited to the *embarassingly parallel* type of problem, where little or no interdependency between subproblems is present (see Fox in [3]). For these cases, a permutation of the processors has no significant effect on the efficiency of the parallel algorithm. Were the method to be used on geometrically oriented applications, where data dependencies between subproblems necessitate communications, it may, by moving data to neighbouring processors not involved in the current communication pattern, increase both the fineness of the granularity and the total communication overhead. An example of this problem is given in the next section. The multi-level load balancing algorithm to be presented achieves the same logarithmic parallel complexity as Cybenko's scheme but makes no use of the physical topology of the parallel computer.

The paper is organized as follows: in the next section, the design requirements for the dynamic load balancing algorithm are discussed, followed by the standard diffusion algorithm. Then the multi-level algorithm is introduced and its complexity discussed. In the fifth section, the model problems used to test the new algorithm and to compare it with the standard procedure are described, together with the results obtained by implementations of both methods.

2. Constraints and objectives

The load balancing algorithm to be described in section four is designed for use in a parallel solver for partial differential equations. Algorithms for this type of problem have the following characteristics:

- One subproblem is assigned to each processor.
- The portion of the problem assigned to each processor (a section of the computational grid) is divisible into smaller units (the grid points), which may be reassigned to other processors and which have equal computational load.
- The computation proceeds by alternating calculation and synchronization phases. A calculation phase consists typically of one or more applications of an iterative pde solver; the summation and broadcast of the residual forms the synchronization phase. This is followed by the error estimation and grid adaptation. It is at this point that a dynamic load balancer would be applied.

The prime objective in the design of a dynamic load balancing algorithm is that it must be fast. Since it will be incorporated into a parallel application program and may be called frequently, without however contributing directly to the solution of the user's problem, the time spent in performing the load balancing will lead directly to a loss in parallel efficiency. If the algorithm is too costly, it may well be cheaper to continue the computation with a load imbalance than to use the balancing procedure.

Since it is to be incorporated into the parallel application program, the dynamic load balancing algorithm should also contain as much parallelism as is possible. This should both accelerate the balancing process and reduce the need for global communication.

Many applications have a degree of data dependency between the subproblems. This is particularly true of problems with an underlying geometry and which are parallelized via some kind of domain decomposition or domain partitioning approach. Each dependency will, for a

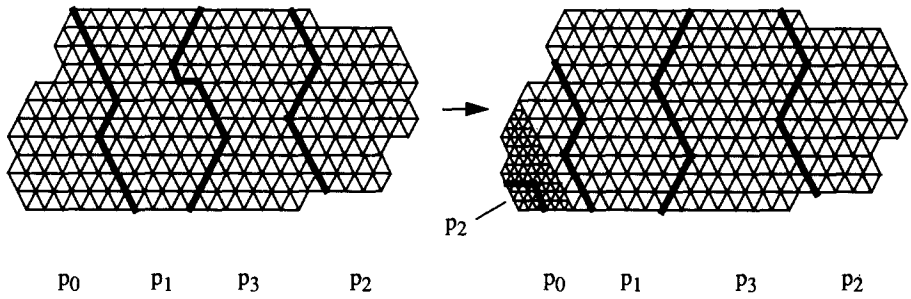

Fig. 1. Non-optimal load balancing.

message-passing based multiprocessor, result in a communication requirement. Thus in the case of numeric solution methods for pdes, the size and pattern of the discretization molecule determine the neighbourhood relationships between grid points, and thus between the grid partitions. For parallel computers with local neighbourhoods such as Transputer systems and Hypercubes, the mapping of the subproblems onto the processors will try to take these data dependencies into account. Since the cost of transferring a message between neighbouring processors is generally lower than between non-neighbouring ones, subproblems with an interdependency should be mapped onto neighbouring processors. The dynamic load balancer should therefore respect the dependencies between subproblems, i.e. it should not produce solutions which introduce additional communication requirements between processors. Consider *Fig. 1*, where a grid has been initially balanced among the four processors p_0–p_3 such that neighbouring subgrids are allocated to neighbouring processors in a two-dimensional hypercube configuration. Note that in addition, processors p_0 and p_2 are also neighbours. This is a typical situation, in which the topology of the problem is not equal to that of the processors. A load balancer should not be tempted to shift work (grid points) between physically neighbouring processors whose assigned sub-problems are not related. This would lead to the second situation, in which a portion of the grid assigned to p_0 has been refined. A load balancing method which takes advantage of the physical hypercube topology may wish to move a portion of the refined grid to p_2. This is obviously to be avoided on grounds of unnecessary complication, too fine a granularity and the additional communication requirement. In the above example, therefore, processor p_0 should only be allowed to move additional computational load to processor p_1.

3. Basic Diffusion Method

The diffusion load balancing method is defined as follows:

Algorithm 1 (diffusion method)

```
procedure diffusion_balance
begin
  while (not converged) do
    for all processors p_i do
      for all neighbours p_j of p_i do
        compare l_i and l_j
        transfer ⌊(l_i - l_j)/2⌋ work units from p_i to p_j
      end for
    end for
  end while
end diffusion_balance
```

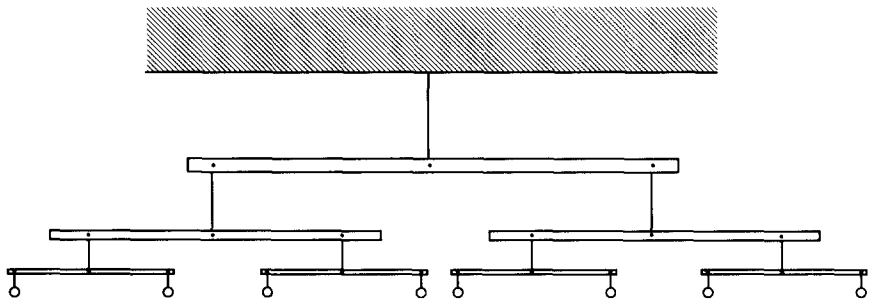

Fig. 2. Balanced mobile.

Essentially, each processor p_i compares its current load l_i with that of each of its neighbours in turn and transfers enough work units to achieve a local load balance. The process is repeated until all processors detect the load to be locally balanced.

The analogy of the movement of load through the processor network to the physical process of diffusion gives this and similar methods their name. In fact, the analysis of diffusive load balancing by Boillat in [1] makes use of the matrices occuring in the numerical solution of such a physical problem.

Methods of this kind, whose decision criteria are of a local nature are likely to produce solutions such as *Diff.* and *HtN* given in *Table 1*. They are not able to detect a global imbalance, nor indeed to remedy it. Furthermore, the number of iterations needed increases with the number of processors.

4. A multi-level load balancing algorithm

The load balancing algorithm presented in this paper is motivated by a divide-and-conquer, or multi-level type approach to the problem. Consider the mobile of *Fig. 2*, which consists of *beams, threads* and *weights*. The point of suspension of each beam on the bottom end of the thread may be slid backwards and forwards. We call the mobile *balanced* when all beams are horizontal. This is the case when the load on each side of each beam is equal. Thus we may go about balancing the mobile by first balancing the top beam by sliding the uppermost thread into the correct position and then proceeding recursively by balancing each of the beams one level beneath. The algorithm presented below follows exactly this strategy. Thus, the decision space of the algorithm in the first phase is global, since it involves all of the weights, and then becomes successively more local. In this way, it is hoped that the globally imbalanced solutions of diffusion methods can be avoided.

We consider the set P of subproblems and denote by $\| P \|$ the number of subproblems in the set P. Since we are assuming one subproblem per processor, denoting one particular subproblem will, in effect, also identify one particular processor and vice versa. The change in computational load of subproblem p_i is denoted by l_i. A positive (negative) value of l_i indicates that an increase (decrease) in the work load of subproblem p_i has taken place compared to the previous, balanced state. The sum of the load increments l_i of all subproblems p_i in the subset P_j of P is denoted by L_j.

The multi-level load balancer is then given by the following algorithm:

Algorithm 2 (multi-level load balancing)

```
procedure balance (P: set of subproblems)
begin
  if || P || =1 then return
```

```
  bisect P into P1 and P2
  calculate L1, L2
  transfer ⌊(L2 || P1 || -L1 || P2 || )/( || P1 || + || P2 || )⌋ work units
  from
    P2 to P1
  balance (P1)
  balance (P2)
end balance
```

The algorithm, which is called with `balance` (P), where P is the set of all subproblems, proceeds by first finding two subsets P_1 and P_2 of P which are connected by one or more edges in the communication graph and satisfy the following conditions:

$$P_1 \cap P_2 = \emptyset$$
$$P_1 \cup P_2 = P$$
$$| \, \| P_1 \| - \| P_2 \| \, | \leq 1.$$

The total load increment for each subset L_1, L_2 is then calculated in order to determine the number of work units to be transferred from P_1 to P_2. As for diffusion methods, for which this is the basic operation, the number of work units transferred is equal to the fraction of the difference in workload of the two sets of processors that corresponds to the number of processors in each set. The procedure is then called recursively for each of the subsets P_1 and P_2. The recursion terminates when the cardinality of the set of subproblems has been reduced to one. Note that no assumptions on the processor topology are made by the algorithm. This gives the user the freedom to orient the bisection of the processor sets towards his or her problem topology. In this manner, problems such as are depicted in *Fig. 1* can be avoided.

The basic operation of the method is a diffusion-like movement of work from one set of processors to another. We will consider the complexity of the method in terms of the number of calls to procedure `transfer` that is necessary.

The complexity of the algorithm is given by the following theorem:

Theorem 1. *Algorithm 2 solves the load balancing problem with* $\lceil ld(\| P \|) \rceil$ *(parallel) transfer operations. Induction over the level of recursion: Let* $lb(P)$ *assert that the set P is load balanced. We then observe*

$$lb(P) \Leftarrow lb(P_1) \wedge lb(P_2) \wedge \left(| \, L_1 - L_2 \, | \leq 1 \right).$$

(1) *In order to satisfy* $| \, L_1 - L_2 \, | \leq 1$, *the algorithm must equalize the loads in the two subsets* P_1 *and* P_2. *This is achieved by transferring* $\lfloor (L_2 \| P_1 \| - L_1 \| P_2 \|)/(\| P_1 \| + \| P_2 \|) \rfloor$ *units of work from* P_2 *to* P_1, *where a negative value indicates a transfer in the opposite direction. This leads to the new load in the subsets*

$$L_1^{new} = \left\lfloor \frac{L_2 \| P_1 \| + L_1 \| P_1 \|}{\| P_1 \| + \| P_2 \|} \right\rfloor$$

$$L_2^{new} = \left\lfloor \frac{L_1 \| P_2 \| + L_2 \| P_2 \|}{\| P_1 \| + \| P_2 \|} \right\rfloor,$$

which represents a balanced load w.r.t. the subsets P_1 *and* P_2.

(2) *Conditions* $lb(P_1)$ *and* $lb(P_2)$ *are achieved by the recursive call to procedure* `balance`.

(3) *Since the size of each subset is halved with each recursive call, and the recursion terminates when the size of the subset reaches one, the algorithm requires* $\lceil ld(\| P \|) \rceil$ *recursion levels.*

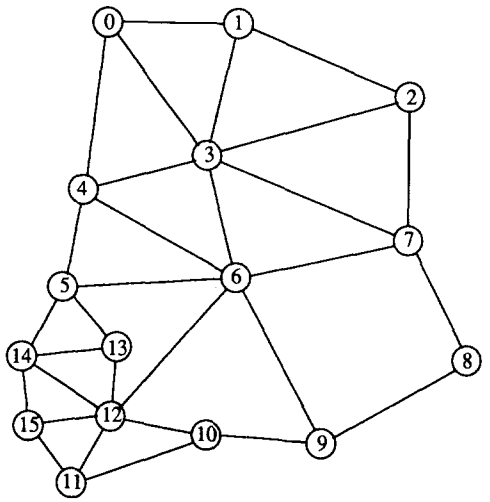

Fig. 3. Communication graph of test case $T1$.

Furthermore we observe that since the work transfers initiated by the various instantiations of procedure `balance` *at the same recursion level apply to disjoint sets of subproblems, they may be performed in parallel. The number of parallel transfer operations is therefore equal to the number of recursion levels.*

5. Results

Both the multi-level algorithm and the diffusion algorithm have been implemented in order to demonstrate and compare their behaviour. Two test problems are considered. The communication graph of the first test case ($T1$) consisting of 16 subproblems is depicted in *Fig. 3*. The load imbalance is created by processor p_0 generating 16 extra work units. The division into subproblems (processor sets) for the multi-level algorithm was obtained by grouping those processors with the lowest and with the highest indices together. A second test case ($T2$) for comparing the multi-level method with the standard diffusion scheme merely has a linear array of 16 processors.

The rebalancing instructions issued by the multi-level algorithm for each test case are given by *Table 2*. The transfer strategy that was implemented searches for as many pairs of neighbouring processors as possible in the two subgroups between which to transfer units of work. The load transferred per processor pair is a corresponding fraction of the total amount calculated by Algorithm 2. In this manner, the number of work units transferred between any two processors can be reduced, which may be advantageous in retaining the structure of the problem. Each transfer phase corresponds to one level of the recursive algorithm.

Table 2 shows that exactly $4 = \mathrm{ld}(16)$ parallel redistribution phases are required by the algorithm to solve each of the load balancing problems. Furthermore no data movement is required between non-neighbouring sub-problems. The communication graph remains invariant during the load balancing procedure. Where it is possible, the algorithm allows several work transfers between different processors within each phase, enabling a corresponding degree of parallelism.

In order to demonstrate the progress of the diffusion load balancing algorithm we define the vector $L = (l_0, \ldots, l_{15})^T$, where each element l_i is equal to the the current load increment

Table 2
Balancing instructions for test problems

Phase	*T1*			*T2*		
	Transfer units	From proc.	To proc.	Transfer units	From proc.	To proc.
1	3	5	13	8	7	8
	3	6	9	–	–	–
	2	7	8	–	–	–
2	4	0	4	12	3	4
	4	2	7	4	11	12
	4	3	6	–	–	–
	1	10	12	–	–	–
3	5	0	3	10	1	2
	5	1	2	14	5	6
	1	6	4	6	9	10
	3	9	10	2	13	14
	2	12	14	–	–	–
4	6	0	1	11	0	1
	4	4	5	9	2	3
	1	7	6	15	4	5
	1	8	9	13	6	7
	1	10	11	7	8	9
	2	13	12	5	10	11
	1	14	15	3	12	13
	–	–	–	1	14	15

of processor p_i. The results obtained by the standard parallel diffusion algorithm (Algorithm 1) are given in *Table 3*. After nine data exchange phases local balance is achieved. However, processor p_0 still has a load of 5 units, whilst the majority of processors has not received any extra load at all. This particular problem has, of course, the solution $l_i = 1$, $i = 0 \ldots 15$.

Tables 3 and *4* show the successive phases of the vector L produced by the diffusion method and the multi-level method respectively, when applied to the second test case. A comparison reveals that the diffusion algorithm transports information slowly across the array of processors, terminating when the load imbalance between all pairs of neighbouring processors becomes ≤ 1. The multi-level method, on the other hand, besides giving an exact solution, spreads the load change globally.

Table 3
Successive values of L with diffusion method applied to $T2$

Phase	L															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	8	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	8	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0
3	6	6	2	2	0	0	0	0	0	0	0	0	0	0	0	0
4	6	4	4	1	1	0	0	0	0	0	0	0	0	0	0	0
5	5	5	3	2	1	0	0	0	0	0	0	0	0	0	0	0
6	5	4	4	2	1	0	0	0	0	0	0	0	0	0	0	0
7	5	4	3	3	1	0	0	0	0	0	0	0	0	0	0	0
8	5	4	3	2	2	0	0	0	0	0	0	0	0	0	0	0
9	5	4	3	2	1	1	0	0	0	0	0	0	0	0	0	0

Table 4
Values of L with multi-level method applied to $T2$

Phase	L															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	16	0	0	0	0	0	0	−8	8	0	0	0	0	0	0	0
2	16	0	0	−12	12	0	0	−8	8	0	0	−4	4	0	0	0
3	16	−14	14	−12	12	−10	10	−8	8	−6	6	−4	4	−2	2	0
4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 5
Comparison of diffusion and multi-level

#Processors	Diffusion		Multi-level	
	#Iterations	Imbalance	#Iterations	Imbalance
8	9	4.9	3	0
16	18	9.6	4	0
32	36	18.4	5	0
64	76	32.4	6	0
128	148	57.4	7	0

Denoting the total load change by C and the number of processors by n, we define the imbalance I as the $\|\cdot\|_2$ norm of the difference between the final load vector L computed by the load balancing algorithm and the vector $(l, l, \ldots, l)^T$, $l = C/n$, which represents a perfectly balanced solution. In order to enable a perfectly balanced solution, we choose test cases in which the total load change is divisible by the number of processors. For the comparison of *Table 5*, we choose a linear array of processors, where processor p_0 creates a load imbalance equal to twice the number of processors. The number of iterations for the diffusion method is seen to increase approximately linearly with the number of processors, whereas that of the multi-level method increases, as was shown above, only logarithmically. Observe that the linear increase is less than one might expect, based on the analysis of Boillat [1]. This is due to the fact that the smallest amount of load transferable is one discrete unit. In addition, the multi-level method is exact, whereas the diffusion method has an increasing imbalance which is similar in structure to that of *Table 1*.

An analogy with iterative methods for the solution of diffusion (i.e. elliptic) equations is useful in comparing the two load balancing schemes. The diffusion method corresponds to a single-grid iteration such as Jacobi or Red-Black Gauss-Seidel, which may be parallelized efficiently. However, the spread of information of such methods is very slow, as only local exchanges take place. Multigrid methods, on the other hand, combine single-grid iterations such as Gauss-Seidel on grids of varying degrees of fineness in order to provide a global exchange of information at a low computational cost (see e.g. Hackbusch [4]). The same is true of the multi-level load balancer presented here – transfer operations provide a diffusion-like movement of load on a hierarchy of scales corresponding to the levels of recursion.

6. Conclusions

The efficient solution of problems with changing computational load on multiprocessors requires the use of dynamic load balancing methods. These must be fast and preferably

deterministic in the sense that they provide a redistributed load within a known number of steps to some guaranteed degree of accuracy. Standard diffusion techniques do not fulfil these requirements. In this paper it has been shown how a hierarchic, or multi-level approach to the problem which is based on an intuitive model can motivate a balancing algorithm which is both fast and accurate. The algorithm has logarithmic parallel complexity, which for typical MIMD machines (up to $\approx 10^3$ processors) should prove to be sufficiently fast. Furthermore, the algorithm makes no use of the physical topology of the machine. Results obtained from serial implementations of the new algorithm and of the standard diffusion algorithm demonstrate the characteristics of the two methods.

Further work will include the implementation of the load balancer in an important application program: a parallel finite element code for unstructured grids with adaptive refinement.

References

[1] J.E. Boillat, Load balancing and Poisson equation in a Graph, *Concurrency Practice Exper.* 2(4) (1990) 289–313.
[2] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, *J. Parallel Distributed Comput.* 7 (1989) 279–301.
[3] G.C. Fox, Parallel computing comes of age: supercomputer level parallel computations at Caltech, *Concurrency Practice Exper.* 1 (1) (1989) 63–103.
[4] W. Hackbusch, *Multi-grid Methods and Applications* (Springer-Verlag, Heidelberg, 1985).
[5] R.D. Williams, Performance of dynamic load balancing algorithms for unstructured mesh calculations, *Concurrency Practice Exper.* 3 (5) (1991) 457–481.