LA-UR 97-333

CONF-970342--9

Approved for public release; distribution is unlimited

Title: MLB: Multilevel Load Balancing for Structured Grid Applications

Author(s): Daniel J. Quinlan
Markus Berndt

Submitted to: Proceedings of Eighth SIAM Conference on Parallel *Processing for Scientific Computing*
March 14-17, 1997
Minneapolis, Minnesota

MASTER

Los Alamos
National Laboratory

DISCLAIMER

MLB: Multilevel Load Balancing for Structured Grid Applications *

Dan Quinlan † Markus Berndt ‡

Abstract

The Multilevel Load Balancing algorithm (MLB) is a parallel algorithm that determines the communication schedule that is necessary to balance a distributed discrete load function. The MLB algorithm focuses on structured grid computations and their load balancing requirements, which we feel are largely unsupported within the load balancing community. The interface to MLB is inherently simple; a distributed discrete load function is provided by the user and a communication schedule is returned. The load function can, for example, map to one or more distributed arrays. So far the implementation includes a parallel version of only the one dimensional MLB algorithm and produces a communication schedule that requires at most log(p) communication steps, where p is the number of processors (log() stands for the logarithm of base two). MLB was first described by Dan Quinlan and Steve McCormick [1, 2, 7]. This work forms just one of the object-oriented class libraries within the OVERTURE Framework [5], an object-oriented environment for the numerical solution of partial differential equations in serial and parallel environments.

1 Introduction

This paper focuses upon load balancing appropriate for structured grid computations. We present two things in this paper, first a simple interface for structured grid load balancing, and second the MLB algorithm itself. The interface represents a simple application interface we have found useful for load balancing our own parallel adaptive mesh refinement (AMR) [4, 7, 2] applications. We expect that the interface is sufficiently simple to permit a wide number of application areas to use MLB for load balancing of structured computations. We present the MLB algorithm using this interface because we think it is useful for structured grid applications and because we hope that others might adopt it to more readily permit interchangeability of load balancing algorithms within structured grid computations. We feel that load balancing has most commonly been too intimately linked to the structured grid applications where it has been used. This has not been true of unstructured grid computations because the load balancing and distribution of data are more orthogonal issues to the unstructured grid computations.

Within load balancing there are two principle methods, graph partitioning and geometric decomposition. Each have advantages and disadvantages. The graph partitioning work is well suited to unstructured grid computations, but the requirements of structured grid computations are not well addressed. Within structured grid computations the

*This work supported by the U.S. Department of Energy through Contract W-7405-ENG-36.

†Computing, Information and Communications Division, Los Alamos National Laboratory, Los Alamos, NM. http://www.c3.lanl.gov/dquinlan/home.html, http://www.c3.lanl.gov/cic19/teams/napc/

‡University of Colorado at Boulder, http://amath-www.colorado.edu/appm/student/berndt/Home.html

resulting decomposition is most readily handled if the partitions of the domain represent a set of rectangular subdomains. Geometric decomposition of the domain can accomplish this requirement readily. MLB is a parallel load balancing algorithm which internally works by means of a geometric decomposition of the domain. MLB produces simple rectangular subregions. Since the cost of load balancing is often less that that of the shuffling of data as required to balance the loads, the particular focus in MLB is upon the generation of efficient communication schedules for the transfer of data within the parallel environment. This has been an important concern because of the high latency of the parallel machines used at Los Alamos. Communication costs with structured grid computations are less of an issue since they are most generally uniform across the processors owning the load balanced grids.

2 Interface Description

The interface to a load balancer based on graph partitioning methods is most often a graph of the connections between elements of the domain to be load balanced. This edge connection graph is inappropriate for the load balancing of structured grid domains because it both does not take advantage of the regular nature of the structured grid and it does not embed the constraints that the subdomains found through load balancing be rectangular. The later constraint is most important within structured grid computations.

The interface for the MLB code is a simple distributed multidimensional integer array. The elements of the load array define relative weights of computational load, typically. No attempt is made thus far to represent or balance the communication loads, but the communication loads are more uniform within the structured grid methods and so not considered as significant an issue within our own driving applications (Adaptive Mesh Refinement). The load array may be chosen to have any discretization of actual problem domain that is considered suitable, this permits tradeoffs in memory and computation for quality in the resulting load balanced distribution.

Though a simple interface, we would propose general geometric based load balancers could readily use this and thus form the basis for a comparison and sharing of geometric load balancers which is presently not possible. This would substantially simplify the use of load balancing for structured grid computations. Further, we feel this would encourage the abstraction of load balancing from applications where for structured grid applications the two are often mixed.

3 Description of the 1D Algorithm

The input for this algorithm is a distributed array of nonnegative integers, the discrete load function. The user provides a meaningful mapping of this array to some distributed structure to be load balanced. The numbers in the array can be thought of as representing a computational load associated with a certain region in the distributed structure that is to be load balanced. The output of the algorithm is a communication schedule, that leads to a balanced computational load.

For simplicity consider a number of processors that is a power of two. We can number these processors successively by their rank from 0 to p-1, where $p = 2^d$. Now we map these numbers to a graycode of of order d, which means that we need d bits to represent the graycode. The mapping for d = 2 is

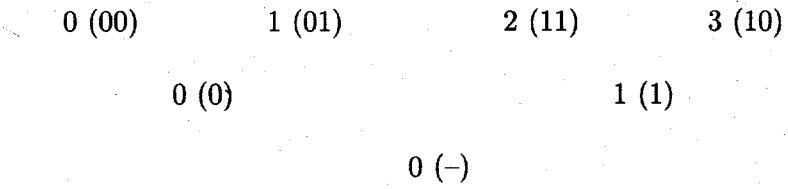
rank	graycode
0	00
1	01
2	11
3	10

Note that graycodes of processors with successive ranks differ only by one bit. We chose a unique mapping by mapping rank 0 to the zero graycode and changing bits starting with the least significant one. Every processor in a hypercube of dimension 2^d has d edges. We number the edges in the following way: The 0-edge is the connection to the processor with a graycode that differs from the current one in the least significant bit, the (d-1)-edge is the connection to the processor with a graycode that differs from the current one in the most significant bit. All the other edges are defined accordingly.

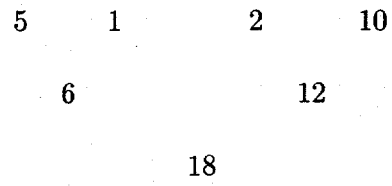
The algorithm can be divided up into four phases:

3.1 Phase 1: Accumulation of the Load on Sub-Hypercubes

Given the piece of the discrete load function that is local to the processor we calculate the total load associated with this processor (by summing up all the numbers in the local piece of the load function). Now we assume to have a coarser level of 2^{d-1} processors. Each of these processors calculates the sum of the loads of two processors on the next finer level. For example in the case where d=5 the processor with graycode 10010 on level d-1 calculates the sum of the loads on the processors with graycodes 100101 and 100100. Recursively this defines a hierarchy of levels where on the coarsest level (where d=0) we only have one processor. This processor's load is the total load that is associated with the whole distributed discrete load function. This hierarchy of processors can be easily represented in a binary tree, where the finest level corresponds to the leaves and the coarsest level corresponds to the root, like in the following example with d=2.



Here the graycodes are displayed in parentheses. A typical example for the tree of processors after accumulating the loads is the following. The numbers represent the loads on the corresponding processors.



This accumulation can be done in $\log(p)$ steps of communication, since we only need to do interlevel communication. There are $\log(p) + 1$ levels, hence $\log(p)$ steps of communication are needed.

3.2 Phase 2: Calculation of the Desired Loads

In order to determine how much load has to be moved from one sub-hypercube to another we have to calculate the desired load for each processor in the tree. On the coarsest processor

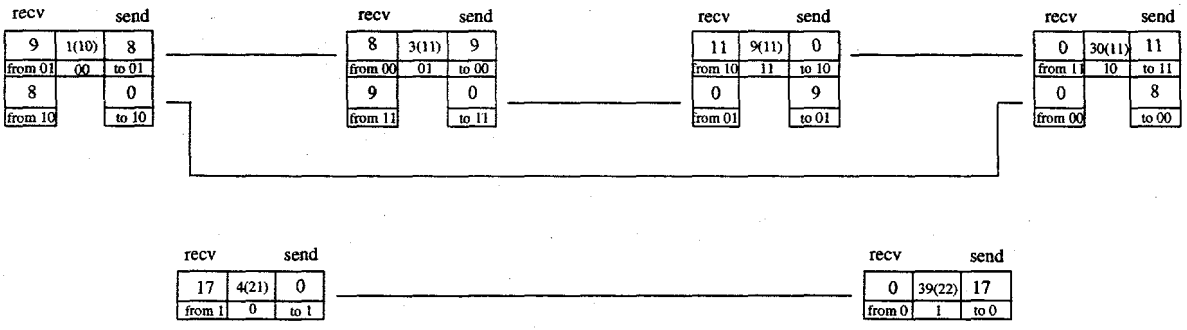
the desired load is equal to the total load that has been determined in the previous phase. On the next finer level there are only two processors (we will call them child-processors), each of which should represent one half of the discrete load function of their parent. Thus the desired load for both is half the desired load on the coarser processor. Since we are dealing with integer valued loads, the remainder of the division has to go on one of the child-processors. This introduces an discretization error that is bounded by $\log(p)$ on the finest level. More precisely, the desired loads on the finest level differ by at most $\log(p)$. On a parallel computer one can not always come up with an additional p-1 processors in order to build the binary tree. So each processor keeps the data that is on all its parent nodes and siblings along edge number 0 (flip least significant bit in the graycode) in the tree. This strategy does not affect the complexity in phase 1 and does not require any communication during phase 2 at all. The computational complexity on each node is of order $\log(p)$.

3.3 Phase 3: Optimal Balancing of the Accumulated Loads

In this phase we find the communication schedule that is associated with a constant discrete load function (where each array element in the load array is equal to one). In order to do that we need not deal with the load function itself. Considering the accumulated loads and desired loads on each processor is enough.

We traverse the tree starting at the root. The first step is easy: looking at the desired load and the accumulated load on each child-node we can easily determine, how much load each child has to send to or receive from its sibling. On the next finer level (four processors) the situation is already more involved. Each node has two siblings, one along the 0-edge and one along the 1-edge in the 2^2 hypercube. The communication schedule determined on the next coarser level tells us how much load the two leftmost nodes have to send to, or have to receive from, the two rightmost nodes (same for the two rightmost nodes). Given the information about the accumulated loads on the four nodes and the communication schedule for the next coarser level we can determine the communication schedule along the 1-edge. The communication along the 0-edge, which corresponds to communication with the direct sibling in the tree, is now easily computed, taking in account the communication along the 1-edge, the accumulated loads and the desired loads.

The following is an example, showing the schedule for optimal balancing of the given loads on four processors.



Here the number in the center of each node is the accumulated load, the number behind that in parenthesis is the desired load. The columns on the left and on the right of each

node contain the amount of load that has to be received or sent along the specified edge in the hypercube.

In case we have to send to a node that is to the left of the current node, then we have to take that amount of load off the available load from the left. The calculated schedule has to be interpreted in the following order. First a node sends to or receives from its neighbor along the coarsest edge (the (d-1)-edge), then it communicates with the next finer one and so forth until it communicates with the finest one (the 0-edge).

On each level, l, we need to communicate along each edge of the sub-hypercube that the level represents, that makes for order $\log(2^l)$, which is equal to l. We need to do that sequentially for each level, starting at the coarsest, so the complexity is of order $(\log(p))^2$, since $1 + 2 + 3 + ... + \log(p) = \log(p)(\log(p) - 1)/2$.

Why doesn't this part of the algorithm fail, due to lack of available data? Processing a given edge we only know about the accumulated load on that node and the load that was sent to it along coarser edges. There are cases where a node has to both send and receive along the same edge. However, it can never happen that a node needs to send data that it receives along the same edge, since that would just mean sending load back, to where it came from.

3.4 Phase 4: Handling of Indivisable Loads

After phase 3 we have all the information we need to deduce the communication schedule that leads to the optimal balancing of the load. Unfortunately this only holds for constant load functions. In the more general case we cannot always guarantee that we can achieve the optimal balancing, since the discrete load function may contain big as well as small values. This may lead to a situation where we want to send a certain load along an edge, but are not able to, since the entries in the load function do not add up to that particular load. In that case we should send a load that is as close as possible to the optimal load that is to be sent.

We process only the finest level and in the following way. Starting with the coarsest edge we determine what the amount of load is that we have to send and where we have to send it. Then looking at the discrete load function and the pieces of load function that were already sent to us, we determine what the piece of load function is that we have to send. The difference of the load that is associated and that of this piece is added to the loan variable. This variable is added to the load to be sent on the next finer levels. Then we send the piece to the neighbor along the current edge and proceed with the next finer level. Upon receipt of a piece of load function from a neighbor we have to subtract the difference of the load of that piece with the load that is expected along that edge.

The introduction of the loan variable ensures that we take into account on finer levels that we may have sent more than we were supposed to, or have received less than we were supposed to along a coarser edge. If we send more than we are supposed to, we send data, was supposed to be sent along a finer edge, hence we are taking out a loan from finer edges.

Since we need to send and receive at most once along each edge on the finest level the complexity according to the communication is of order $\log(p)$.

Why doesn't the loan variable method cause problems in a situation where data has to just pass through a node? Let node (a) and node (b) be connected by a k-edge. If node (a) sends data to node (b), which has to hand it through to node (c), then node (b) and node (c) are connected by the (k+1)-edge. Hence if node (a) sends less (or more) than it is supposed to, node (b) finds out about it and adjusts its loan variable. This loan variable is

taken into account for the communication along the (k+1)-edge. If node (b) receives more, it is going to send more along the next finer edge by the exact amount that it receives more from node (a). Similarly, if it receives less it is going to send less along the (k+1)-edge, by the exact amount that it receives less from node (a). So the loan variable mechanism preserves the contiguity of the load array in a hand through situation as described above.

4 Complexity of the Algorithm

The complexity of the load balancing algorithm is interesting, but the dominate cost of the use of load balancing is in the redistribution of the data. This is why so much effort has been made to generate an efficient communication schedule for this latter phase.

The complexity of the algorithm is really broken up into two parts, computational complexity and communication complexity. The obvious computational complexity is that each element of the load array is summed on each processor, this is a simple process and typical within load balancing. Since the load array is distributed this complexity is of order $\frac{n}{p}$. The more important complexity is the computational complexity since the cost of the load balancing is dominated by the communication required. Phase 1 of the MLB algorithm is of complexity $\log(p)$, since data has to be communicated along each edge of each node. Phase 2, the calculation of the desired loads, does not require any communication, hence the complexity is of order 1. The pre-processing step, or phase 3 of the algorithm, is of complexity $(\log(p))^2$. On every level l we have to do $l-1$ steps of communication. Since there are $\log(p)$ levels the compexity is of order

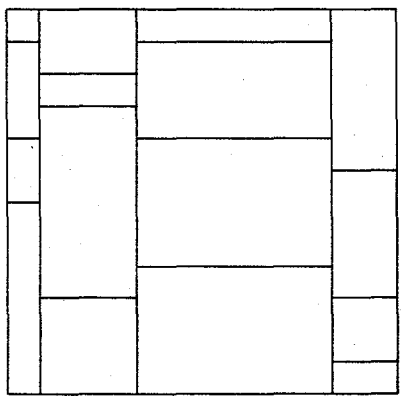
$$\sum_{l=1}^{d} l = \frac{d(d-1)}{2}$$

in the case where $p = 2^d$. In phase 4 communication along every edge has to be performed, so the complexity is of order $\log(p)$.

Overall the communication complexity of the MLB algorithm is $(\log(p))^2$. However, the complexity of generated communication schedule is of order $\log(p)$.

5 Extension to more Dimensions

The MLB algorithm can be extended to higher dimensional problems in a rather straightforward way by applying the one dimensional algorithm to the coordinate axis successively. In order to do that we have to restrict the way in which the load function can be distributed. The following figure illustrates this for a 4×4 processor array.



Cuts in one coordinate direction have to go all the way through the load function, the distribution along the other coordinate direction can be random. In the two dimensional case the MLB algorithm can be thought of as first adjusting the vertical cuts in such a way that the load is balanced over all the vertical strips. Then parts of the load function have to be communicated in order to actually obtain the balanced state. Now the communicated parts of the load function have to be rearranged such that we obtain a distribution in every strip that only shows straight horizontal cuts. Eventually the MLB algorithm is applied again, now to each strip, which results in a balanced distribution of the load. Present work on MLB is being done to address load balancing of multidimensional load arrays as can occur within multidimensional structured applications. Within the load balancing for our adaptive mesh refinement we use the one dimensional MLB algorithm, even though the problems are two and three dimensional. Thus we get effective use of the MLB algorithm even for higher dimensional applications, and even though the multidimensional applications are distributed along all axis. Thus the dimension of the structured grid application is not necessarily tied to the dimension of the load balancing.

6 Availability of Software

The the time of this writing this software is not available in its multidimensional form. We would like to wait until this work is finished before releasing the software publicly. However, we do intend to release this software mid summer 1997. Details will be available from http://www.c3.lanl.gov/cic19/teams/napc/.

References

[1] S. McCormick, D. Quinlan, *Multilevel Load Balancing*, Internal Report, Computational Mathematics Group, University of Colorado, Denver, 1987.

[2] S. McCormick, D. Quinlan, *Asynchronous Multilevel Adaptive Methods for Solving Partial Differential Equations on Multiprocessors: Performance Results*, Parallel Computing, Vol. 12, 1989, pp. 145-156.

[3] S. McCormick, D. Quinlan, *Dynamic Grid Refinement for Partial Differential Equations on Parallel Computers*, Proceedings of the Seventh International Conference on Finite Element Methods in Flow Problems, 1989, pp 1225-1230.

[4] D. Balsara, D. Quinlan, *Parallel Object-Oriented Adaptive Mesh Refinement*, in this proceedings, SIAM, 1997.

[5] D. L. Brown, G. S. Chesshire, W. D. Henshaw, and D. Quinlan, *OVERTURE: An Object-Oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments*, in this proceedings, SIAM, 1997.

[6] K. D. Brislawn, D. L. Brown, G. S. Chesshire, and D. J. Quinlan, *Overture code*, 1996. Los Alamos National Laboratory Computer Code LA-CC-96-04.

[7] D. Quinlan, *Adaptive Mesh Refinement for Distributed Parallel Processors*, PhD thesis, University of Colorado, Denver, June 1993.