Load Balancing Algorithms with Partial Information Management for the DLML Library

Abstract—Load balancing algorithms are an essential component of parallel computing reducing the response time of applications. Frequently, balancing algorithms have a centralized behavior requiring a lot of messages to operate, thus causing scalability problems. A solution to improve scalability is to define a decentralized algorithm, avoiding the generation of bottlenecks. DLML (Data List Management Library) is a tool that, in a transparent way, allows the parallel processing of data that is organized through a List. One drawback of this tool is the global bidding algorithm used to distribute the data (work) generated during the execution. In this paper two load balancing algorithms for DLML handling partial information are proposed. The first algorithm considers a logical Torus topology and the second one follows a Binary Tree topology for communications. The results show how the scalability of DLML was improved, using two clusters of 40 and 1024 processing units, and executing applications with different granularity.

I. INTRODUCTION

The need of reducing the response time of parallel applications, and optimizing the use of hardware resources, makes load balancing an essential component of parallel computing. There are many works that use load balancing, taking into account global or partial information policy to decide the transfer of load among processors. With a global policy, load balancing algorithms require the knowledge about the load state of all the processors in the system [3], [4], [11], [17]. Clearly, to obtain this information when the number of processors increases, a large amount of communications needs to be done generating possible bottlenecks. In this case, the performance is deteriorated showing a lack of scalability of the global information strategy. Using a partial information policy, only the load state of a set of processors needs to be known, distributing the amount of communications and diminishing the bottlenecks in the system [7], [12].

The implementation of a load balancing strategy to distribute the data of an application is not always simple, specially for researchers who are not in contact with parallel programming. For this reason, to facilitate the integration and use of a load balancing policy into parallel applications several programming tools have been proposed [8], [13], [1]. A common objective of these tools is to allow researchers to program their parallel applications without having to deal with communication or synchronization problems, by applying transparently dynamic load distribution.

In particular, DLML is a library for developing parallel applications based on programming with data lists [2]. DLML is implemented under the message passing model, by using MPI-C [15]. Through standard list primitives (Insert and Get), DLML allows the parallel manipulation of the list transparently to the programmer. DLML internally partitions and distributes the list among all the processors of a cluster, and applies the primitives on each local list.

When a processor finishes the processing of its local list and becomes idle, DLML activates a bidding load distribution policy [16] in order to migrate data from active processors to this one that owns the empty list. However the implementation of the bidding algorithm in DLML uses a global information policy, that is, many communications need to be generated in order to gather the information about the load state of all processors in a cluster, resulting in a non scalable policy. To improve DLML scalability some load distribution protocols with partial information management should be taken into account.

In this work, we propose two load balancing algorithms (LBA) with partial information management for DLML. These LBA consider a partial communication based in a Torus and Binary Tree (BT) logical topologies (Fig. 1).

These topologies allow to distribute the communication among all the processors. A processor communicates only with a set of processors belonging to its neighborhood. The number of neighbors of a processor depends on the position in each topology. In this way, each processor only communicates with its neighborhood to transfer data and balance the system load. The objective of using partial communication is to reduce the number of messages required in the balance phase of DLML, decreasing the response times. A comparison was made by using DLML with the bidding global algorithm and with our two proposed partial information algorithms ("Torus" and "Binary Tree"). Two clusters with 40 and 1024 processing units (cores) were used in our tests, executing two types of applications. The results show that our proposed Torus and Binary Tree algorithms improve considerably the DLML
performance and its scalability.

The structure of this paper is organized as follows. Section II presents the concepts of dynamic load balancing. Section III briefly presents the DLML library and its architecture. Sections IV and V describe the Torus and Binary Tree algorithms, respectively. Sections VI and VII are devoted to show the experimental platform and results. Finally, in Section VIII we present our conclusions and future work.

II. DYNAMIC LOAD BALANCING

Dynamic load balancing consists in maintaining a load equilibrium among all the processors of a system during the execution of an application. For this purpose, a LBA needs to determine an indicator of a processor load state (load index).

In general, a LBA considers two policies related with gathering of load states information (information policy) and control to decide load transfer (control policy) [9]. An important point in the information policy is the establishment of a metric to estimate a processor load state (related with its data or processing quantity). Another point is to determine how the information will be collected, in a global or partial way. The control policy is responsible for defining when to make a load transfer and which processor can make such a decision. Only one processor (centralized policy) or all of them (distributed control policy). On another hand, the control policy needs also to define what process will be the receptor in the load transfer.

When the information is collected in a global way, the load index of all processors are sent to one processor, so when a large number of processor are used, possible bottlenecks can occur producing serious limitations on the LBA scalability [6]. An adequate solution to improve scalability is by defining partial information policies that only need the load indexes of some processors. The distribution of communications through some logic topologies is an important factor in LBA scalability [10].

The basis of our proposal is the use of two communication topologies, Torus and binary tree, to define two LBAs of the DLML programming tool. By using these topologies communications are reduced and distributed, however, the implementation of a partial information policy requires additional considerations. Since a global load state of the whole system is not available, each processor has only a partial view of it, thus it is necessary to define a termination protocol in order to know if all the processors have already finished their work. In the next Section the DLML tool and its global LBA are described, then, in Sections IV and V the definitions of the complete Torus and tree algorithms are explained.

III. DLML LIBRARY

DLML (Data List Management Library) is a library for developing parallel applications in clusters [2]. The library is useful for applications whose data can be organized as list items. A List is accessed using typical operations, such as get and insert. These basic operations apparently work on a single list but, internally, DLML works with several local lists distributed in the processors of a cluster. When a local list in a processor \( X \) has no more elements, DLML gets more data from a non empty remote local lists allocated in a processor \( Y \), by applying a bidding load balancing algorithm. In this way the system data (load) is redistributed among the processors.

A. The bidding policy of DLML

A graphic explanation of the bidding policy [16] used in DLML [2] is shown in Fig. 2. In this figure a system with five processors \( X, Y, S, W, \) and \( R \) is shown. The number inside each processor (a circle) represents the local list length in that processor, i.e. the processor load index. Fig. 2.a represents the moment when the processor \( X \) finishes the processing of its local list (thus the local list length is 0), then \( X \) starts a global bidding policy by sending a message to all processors, requesting their local load index. When all the responses are received (Fig. 2.b) \( X \) identifies the processor with the largest local list length, in this example processor \( Y \) with 200 items in the local list. After sending a data request to \( Y \) (Fig. 2.c), \( X \) receives the half of the local list (100 items) of \( Y \) and continues processing the new local list items (Fig. 2.d). This policy clearly lacks of scalability due to the global information policy that is used.

B. DLML Architecture

DLML architecture allows to separate the dynamic load distribution part of the data processing. To this end, DLML uses two processes per processor, Application and Balancer, as shown in Fig. 3. The Application process is in charge of invoking the basic operations of lists (Insert and Get) to obtain the next local list item to be processed or to add a new item to the local list. Moreover, the Balancer process is responsible for communicating with other Balancer processes located on other processors, when the bidding policy needs to be executed. The
Application process communicates with the Balancer in order to exchange information related with synchronization or data transfer.

With the invocation to the Get list operation an item is obtained from the list. However, if the list is empty, the Application process sends a message to the local Balancer in order to ask for more remote data. Then the Application stays blocked until a response message from the Balancer is received. The response message can indicate two possibilities: the transfer of the required data or a request for termination.

An important part of the DLML architecture is the need to define a protocol termination. After a bidding procedure, if it is discovered that all lengths of the local lists are zero, the Balancer sends a termination message to its Application process and ends its execution.

C. DLML programming

Programming with DLML is almost as doing it in a sequential way. The next code is a single program representing the basic DLML programming model. DLML follows a SPMD programming model, so that the code is copied and executed in each processor/core of a cluster.

1. initialize(&L);
2. while( DLML_Get(&L, item) ) {
3.   processing(item, &L);
4. }
5. final_protocol();

At the beginning (line 1 of code), an Application process can insert some elements to its list $L$. Then, it is necessary to define a main loop, in which the function DLML_Get is successively invoked. DLML_Get obtains an element from the list $L$ to be processed, the returned value of DLML_Get is 1/0; 1 indicates that the operation was successful and 0 indicates that there are no more data in the system. As a consequence of the processing of an item (line 3) it is possible that some other items be added to $L$ (e.g. in divide an conquer or branch and bound problems). Finally, because each processor can contain part of a solution, there is a final protocol, where all processors send their partial result to a coordinator process through a reduction DLML primitive.

IV. TORUS ALGORITHM

The proposed Torus algorithm works with the same DLML architecture, the difference now is that the set of processors are virtually connected along a 2-D mesh wraparound topology (Torus). In this topology each processor can communicate with other 4 processors, called neighbors. To describe the Torus algorithm, we refer to the Balancer process using the processor tag.

Four basic stages are considered during the execution of the algorithm: initialization, load distribution, searching of a global load view and termination. Below we describe these stages.

A. Initialization Stage

In the initialization we start by running all the processes (Applications and Balancers), and at least some data are added to the local list of processor zero.

B. Load Distribution Stage

When a processor is unloaded (it can be at the beginning or when the local list becomes empty) it activates the distribution policy, following a local bidding algorithm. An example of the bidding steps is shown in Fig. 4.

In Fig. 4a, the unloaded processor 4 asks (through a message) the load index to its neighboring processors (LIR Load Index Request). Then the processors 1,3,5,7 send their load indexes 7,1,20, and 10 (Fig. 4b) to processor 4 (LI Load Index). Consequently, in Fig. 4c processor 4 selects the processor with the highest load index (processor 5) to send a data transfer request message (DR). Finally, processor 5 sends some data (D) to processor 4 (Fig. 4d).
When a Balancer receives a load transfer request message (it means that it has been elected by a neighbor processor to transfer data), so it enqueues the id of the sender processor in a transfer queue and then sends a message to its corresponding Application process, requesting the local list. When the Balancer process receives the list, if the list length is greater than the number of elements in the transfer queue, it divides the list among the enqueued petitions number plus one (itself) and then makes the corresponding data transfers. Otherwise, if the list length is less than the number of requests in the transfer queue, only some of the enqueued requests can have a data transfer, so a message indicating no-data is sent to the rest of the enqueued processors. In this case, the processors that receive a no-data message restarts again the bidding policy.

A particular situation arises when the neighbors of an unloaded processor X are also unloaded. In the example of Fig. 5 processor 4 first send LIR to its neighbors, then it receives LI messages with a zero value. Consequently, processor 4 sends an availability message (A) to each one of its neighbor processors notifying its availability of processing, and then it switches to a standby state (staying blocked until the reception of a message).

When a processor Y receives an availability message from a processor X, it enqueue the id of the sender processor in a availability queue. If after a local bidding policy some data are sent to Y, Y sends a restarts bidding policy message to all the processors in the availability queue. The processors that receive this message, restarts again the bidding policy.

At the end of the data processing, most of the processors become unloaded and the principal problem is to know when a processor must start the termination stage. This decision is not simple, because the processors do not know the load state of the entire system, they only known the load index of its neighborhood. It may happen that, if a processor starts the termination stage, some distant processor could have data and could need help. To avoid this situation a protocol has been implemented to allow the gathering of a Global Load View.

C. Search Stage of a Global-load-View

The processor 0 is responsible for starting the search of a global load view. After executing the local bidding policy, and if no data is obtained, processor 0 starts the execution of a protocol consisting of two parts, propagation for the spanning tree construction and propagation with feedback.

Propagation for the spanning tree construction

In the first part a PIF (Propagation of Information with Feedback) protocol has been implemented in order to construct a spanning tree on the Torus. In the original PIF version a process, for example process 0, sends a propagation message to its neighbor processors (see Fig. 6a). Then, the processor 0 neighbors receive the message and resend it to their neighbors, skipping the processor from which they received the propagation message (Fig. 6b). This procedure is executed by all processors that receive a propagation message. Note that a processor only re-dispatch the first received propagation message, the subsequent messages are only received and counted. The original PIF protocol ends when all processors have received the propagation message as many times as the number of neighbors. The PIF protocol has the property of generating a minimum weight spanning tree (MST) when it is invoked (Fig. 6c). However the original PIF protocol do not maintain the MST after the execution.

To achieve this, in our proposal we modified the basic PIF algorithm to build and maintain a MST. The first time a processor receives a propagation message, it forwards the message to the neighbors (like the original PIF algorithm), but it also sends a recognition of paternity message to the source processor (parent) (Fig. 6b). In this way a processor can know who are its children processors and who is its parent (Fig. 6c). In Fig. 8 the tree obtained after the execution of the propagation part is shown. The remaining functioning of the modified PIF is the same as the original PIF.

If we identify and maintain the MST generated by the propagation execution it can help to reduce the message number in the subsequent feedback executions, for this reason the propagation for the spanning tree construction part is invoked only once. Through the obtained MST each processor can know who are the processors for sending or receiving messages (the children or the parent) in order to recover a global load view of the system.

Propagation with feedback

In this part, there is a propagation of a message on the built MST asking for load information, and then there is a feedback procedure for gathering the global load view. Processor 0 sends a data confirmation message to its children processors (as shown in Fig. 7d), which, in turn, re-send the message to their own children, so on until the message reaches a processor with no children. In this case that processor sends to its parent the value 1 if its local list is not empty, and value 0 otherwise. After the reception of all the responses from its children, a processor responds to its parent in an equivalent way: 1 if there are some data in its local list or in any list of the offspring, 0 otherwise.

Eventually the processor 0 that started the search of the global load view receives the responses from their children having a an overall indicator of the existence of data in the system (Fig. 7e). If there are still data in the system, processor 0 starts again the local bidding policy, otherwise, it begins the termination stage.
b) Propagation phase execution

D. Termination Stage

In this stage a termination message is propagated to all the processors through the tree generated in the previous stage. The message notifies to the processors that there are no load in the system, and thus they have to start the termination protocol. This propagation goes from the root (processor 0) to the leaf processors in the tree. When a leaf processor receives the termination message it sends the partial result (obtained through the work that the leaf did) to its father and then finishes its execution. A parent processor collects all the children responses and its own response to resend all together to its own father, and then it also stops the execution. This procedure is performed successively until the root processor get an overall result.

The following section presents the Binary Tree distribution algorithm for DLML.

V. BINARY TREE ALGORITHM

The tree load balancing strategy uses a partial information policy. The processors are connected by means of a established Binary Tree topology (shown in Fig. 9), where root processor has 2 neighbors (Fig. 9.a), internal processes have 3 neighbors (Fig. 9.b) and leaf processors has 1 neighbor (Fig. 9.c)

As in the previous Torus algorithm, in the proposed Binary Tree Algorithm also four stages are considered during the execution: initialization, load distribution, searching of a global load view, and termination. All the processors (leaves and internal processors) participate in all the stages but only the root (processor 0) is responsible for starting the search of the global load view and the termination protocol, behaving like in the Torus algorithm.

The Initialization Stage is the same as in the Torus algorithm, so we start with the description of the load distribution stage.

A. Load Distribution Stage

In the Binary Tree Algorithm, when a processor becomes unloaded, it starts the bidding policy as in the Torus algorithm. The only difference is the variable number of neighbors involved in the local bidding policy of each processor. The number of neighbors could be one, two, or three depending
on what processor starts the bidding protocol. If processor 0 is the only one that has data, the load can be distributed toward its branches.

When the root processor starts the bidding protocol sending to its children a load index request message, if its neighbors don’t have data to transfer, it is probably that there are no more data in all the system. To be able to make a termination decision, the root processor runs the search stage of the global load view.

B. Search Stage of the Global Load View

Unlike the Torus algorithm, now is no longer necessary to apply a propagation algorithm to build a tree, because this topology has been defined from the start. The search consists in broadcast a data confirmation message from the root toward the leaves processors. The gathering of information behaves as in the feedback procedure of Torus algorithm, from the leaves to the root processor. When the root processor receives all the responses from their children (through a binary value), it can determine whether there are data in the system or not. If there is still some data, the root processor starts again the bidding protocol, otherwise it starts the termination stage.

C. Termination Stage

The Termination Stage of the Tree Algorithm behaves in the same way than the Torus Termination stage, considering the established Binary Tree topology. Fig. 10 illustrates this stage; first processor 0 sends through the Binary Tree a termination message (Fig. 10a), then the leaves send back their partial results to their parents and finish their execution. Middle processors continue sending back their collected results to their parents until the root processor receives all the results.

VI. EXPERIMENTAL PLATFORM

In order to estimate the performance and the scalability of the Torus and Binary Tree algorithms in DLML, we made a comparison with the initial version (that uses the global bidding algorithm). The comparison includes how load distribution happens, and especially scalability. Data distribution can be an indication about the balance degree, and scalability can reflect if the partial information management was efficient. The experimental platform considers two different clusters and two kinds of applications.

A. Infrastructure

To test the algorithms, two clusters were used:

Cluster 1. The first platform is a heterogeneous cluster of 10 nodes, 4 nodes with two Intel Dual-Core processors at 3GHz., 2GB RAM, and CentOS 4.4. The other 6 nodes have an Intel Quad-core processor at 2.4GHz., 4GB RAM, running Fedora 8. All nodes are connected with a Gigabit Ethernet switch.

Cluster 2. The second platform is the Aitzala cluster located in Mexico City at the Laboratory of Supercomputing and Parallel Visualization (rank 439 of the TOP500 June 2009). In this cluster 128 nodes were used, each node with two Intel Xeon Quad processors at 3GHz, 16GB RAM, running CentOS 5.2. These nodes are connected with an Infiniband switch.

B. Applications

The proposal was evaluated with static and dynamic applications. The static application is a matrix multiplication \( A \times B \), where \( A \) and \( B \) are each \( 1200\times1200 \) matrix. In this application each item in the DLML List, contains a complete row \( \omega_i \) (of the matrix \( A \)) and a complete column \( \omega_j \) (of the matrix \( B \)), to obtain a position \( C_{i,j} \) in the resulting matrix. Under this partitioning scheme, initially the local list of processor 0 contains \( 1200\times1200 \) processing elements.

The dynamic application solves the N-Queens problem [5]. The objective consists in the placement of N-Queens on a chessboard of size \( N \times N \) without any attack among them. In order to find all the possible solutions it is necessary to explore exhaustively the search space modeled through a tree, where the branches that are not candidates for being solutions, are not further explored. In this application the list elements are generated during the execution. Each list item contains the positions of \( k \) Queens placed so far, so that after an item processing new items can be added to the list if there are new possibilities of placing the Queen number \( k+1 \). Note that the length of a list varies according to the addition or elimination of potential solutions (items), making the load balancing an important requirement for the performance improvement.

VII. RESULTS

Fig. 11 shows the data distribution, made by the three algorithms (Global, Torus and Binary Tree) for the N-Queens problem, with \( N = 17 \), running in Cluster 1. The data distribution is measured in terms of the explored solutions (millions scale in the y-axis) by each core on each node (x-axis).

In general, it is noted that the number of explored solutions (the processed data) is higher in nodes 1-4, and lower in nodes 5-10. This is because the application is executed in a heterogeneous cluster, where the first 4 nodes are faster than the rest. Regarding the efficiency of each algorithm, we observe that the partial information algorithms balance better compared to the global algorithm.

To verify this, we calculated the standard deviation (\( \sigma \)) of the explored solutions. In Table II the deviation of the three algorithms in both regions of the heterogeneous cluster is shown. From the table it is observed that the Torus algorithm
TABLE I

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Standard Deviation (σ)</th>
<th>Node 1-4</th>
<th>Node 5-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>9,978,768</td>
<td>5,556,479</td>
<td></td>
</tr>
<tr>
<td>Torus</td>
<td>2,350,132</td>
<td>1,952,682</td>
<td></td>
</tr>
<tr>
<td>Binary Tree</td>
<td>6,414,427</td>
<td>3,812,180</td>
<td></td>
</tr>
</tbody>
</table>

is the best to balance data, then the Binary Tree, and finally the global bidding.

In Fig. 12 the response times (y-axis) for the N-Queens problem (N=16) using the three algorithms are plotted. The response times (displayed in seconds and logscale) were obtained in the Aitzaloa cluster from 8 until 1024 cores. Initially it is noticed that the three algorithms reduce the response time when the number of cores increases. However, above 64 cores the global algorithm loses scalability, increasing the execution time. In this point the number of messages that the global bidding algorithm requires is higher. However this situation does not occur with the two proposed algorithms (Torus and BT), which continues decreasing their response times until 512 cores, from there, the execution time began to increase slightly. We thought that the N-Queens problem with N=16, may not take benefit of all the 1024 used processors, it may adversely affect, rather than help.

We continued our evaluation, but now using the N-Queens application with N=17. In Fig. 13 we show its response times measured in seconds (y-axis logscale). In general, the response times of the Global Bidding algorithm, Torus and BT disclose a time reduction when the number of cores increases. However the scalability in the Global Bidding algorithm collapses from 128 cores. With the BT algorithm the execution times lapses when using 640 cores or more. The Torus algorithm the minimum time is obtained by using 896 cores, being a little bit higher with 1024 cores.

Finally, in Fig. 14 we can see the response times (y-axis) for the matrix multiplication application. In general, it is noticeable that there is not a significant reduction in the response times by increasing the number of cores (x-axis). This is due to the application being static, that is, the
whole data set is created from the beginning, and then it is directly partitioned and distributed to the cluster cores. We also believe that there is no reduction of time due to the processing of a list item, this is very small compared to the cost of sending data from one processor to another. However, in the individual comparison of the algorithms we found that the two proposed algorithms behave better than the Global Bidding algorithm. The Torus and Binary Tree algorithms more or less maintain their times constant for different numbers of cores (with minimum at 640 and 256 cores respectively), while the Global Bidding algorithm has an exponential growth from 8 cores.

VIII. CONCLUSIONS AND FUTURE WORK

In this work, we proposed a Torus and Binary Tree algorithms for load balancing with the DLML programming tool. Both algorithms handled a partial information policy with the aim of distributing communications, thus reducing the response time of DLML and improving its scalability.

Torus and Binary Tree algorithms are composed by the same 4 stages: initialization, load distribution, search of a global load view, and termination. For the Torus algorithm, a PIF protocol was applied in order to obtain a spanning tree which was used in searching of a global view of the system and in termination. The results showed that both algorithms balanced more efficiently the system load and improved scalability compared to the initial global bidding algorithm in DLML.

The balance improvement was measured with the standard deviation of the number of processed data by each node running the dynamic N-Queens application in an heterogeneous cluster. The scalability was also tested by using a cluster with 1024 cores. Each load balancing algorithm executed the N-Queens problem with N=16 and N=17.

We noticed that the Torus algorithm obtained a better distribution than the Binary Tree algorithm. The regular number of neighbors in the 2-D Torus (it is always 4) allowed a more efficient data propagation, whilst in the Binary Tree algorithm the neighbor number was not regular, varying from 1 to 3. Other advantage of Torus algorithm was that the PIF algorithm builds the spanning tree dynamically (it is created at run-time and take into account the communication costs), while Binary Tree uses a static preestablished tree.

Executing a static application consisting on a matrix multiplication, the two proposed algorithms in general obtained a little reduction in the response time when increasing the number of cores, and consistently maintained their response times. Concerning the global bidding algorithm, the execution time increased exponentially. In this test the Binary Tree algorithm was better than Torus because it executed less frequently the distribution policy and for this static application that was better.

Future work includes the development of other partial information algorithms for DLML, with logical topologies such as 3D-mesh, ternary-tree or hypercubes. Also these algorithms could be designed in order to allow DLML to function in Grid technology.

ACKNOWLEDGMENT

The authors would like to thank to the Laboratory of Supercomputing and Parallel Visualization of Universidad Autónoma Metropolitana in Mexico City.

REFERENCES

[1]
[2]