Load Balancing by Changing the Graph Connectivity on Heterogeneous Clusters

Kalyani Munasinghe[1,2] and Richard Wait[2]

[1] Dept. of Computer Science,
University of Ruhuna, Sri Lanka
[2] Dept. of Information Technology,
Uppsala University, Sweden
{kalmun, richard}@it.uu.se

Abstract. This paper examines the problem of adapting parallel applications on a cluster of workstations. The cluster is assumed to be a heterogeneous, multi-user computing environment so that efficient load balancing within the application must take external factors into account. At any time the users of the network are competing for resources. Performance of a particular processor, as a component in the parallel (message passing) computation, depends on both static factors, such as the processor hardware, and dynamic factors, such as the system load and the activities of other users. For each processor, the external factors can be condensed into a single parameter, the load index, which is a normalised measure of the current spare capacity of the processor available to the application.

Numerical experiments show the efficiency of the load balancing strategies on a finite element application with a domain decomposition and the effect on overall computation time.

1 Introduction

Shared cluster networks provide a useful platform for parallel applications because of their cost performance ratio. The cluster environment can offer high performance if resources are managed efficiently. One of the problems in achieving high performance in clusters is that resources may not be fully under control of the individual application. In this environment, parallel programs may be competing for resources with other programs and may be subject to resource fluctuation during execution. In such a system, an important issue is to find effective techniques that distribute the tasks of a parallel program appropriately on processors. One problem is how to schedule the tasks among processors to achieve goals such as minimizing execution time or maximizing resource utilization.

For example, an irregular finite element mesh, may be partitioned into subdomains and each subdomain assigned to a single processor. Assuming that the

P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, pp. 1040–1047, 2005.

computational effect is proportional to the size of the subdomain, two questions arise:

1. What is the optimal size for each subdomain?
2. How can the problem domain be partitioned into such subdomains?

On a homogeneous cluster of dedicated processors (e.g. Beowulf [2]) with a fixed problem size, the partition may be uniform and static.

For some irregular grid applications, the computational structure of the problem changes from one computational phase to another. For example, in an adaptive mesh, areas of the original graph are refined in order to model the problem accurately. This can lead to a highly localized load imbalance in subdomain sizes. Alternatively, load imbalance may arise due to variation of computational resources. For example in a shared network of workstations, computing power available for parallel applications is dynamically changing. The reasons may be that the speed of machines are different or there are other users on some part of the cluster, possibility with higher priority. The partitioning has to be altered to get a balanced load. We propose an algorithm which reduces the load imbalance by local adjustments of current loads to reduce the load spike as quickly as possible and to achieve a load balance. It is assumed that the connections for data transfers between the processors are determined by the data locality but data movement should be kept as low as possible. The load balance is adjusted in general by migrating data to adjacent processors with modifications to the connectivity where necessary.

2 Background

2.1 Some Definitions

Let p be the number of processors. The processor graph is represented by a graph (V, E) with $|V| = p$ vertices and $|E|$ edges. Two vertices i and j form an edge if processors i and j share a boundary of the partitioning. Hence the processor graph is defined by the topology of the data subdomains. As the edges of the processor graph are defined by the partitioning of the domain, when the partition changes the graph topology may change. Each vertex i is associated with a scalar l_i, which represents the load on the processor i.
The total load is

$$L = \sum_{i=1}^{p} l_i \tag{1}$$

The average load per processor is

$$\bar{l} = \frac{1}{p} L \tag{2}$$

and we can define the vector, b, of load imbalances as

$$b_i = l_i - \bar{l} \tag{3}$$

This definition is based on the assumption that in order to achieve a perfect balanced computation, all the loads should be equal. If however the processor environments are heterogeneous and corresponding to each processor there is a load index α_i which can be computed using current system and/or processor information, then the ideal load $\tilde{l_i}$ is defined as

$$\tilde{l_i} = \alpha_i \frac{1}{\sum_j \alpha_j} L \tag{4}$$

Load difference from the ideal load can be defined as

$$d_i = l_i - \tilde{l_i} \tag{5}$$

A processor is therefore overloaded if $d_i > 0$. These simple definitions assume that the computation can be broken down into a large number of small tasks each of which can be performed on any processor for the same computational cost. This is not necessarily true as for example in a finite element computation, the cost of the computation might depend on the number of edges between subdomains in addition to the cost proportional to the size of the subdomains. So the total distributed computational cost is not necessarily equal after any redistribution of the data.

A processor is highly overloaded if the load difference is excessive

$$d_i > cl_i (\text{for some constant } c < 1)$$

typically $c \approx 0.3$ was used in the experiments, a partition is balanced if no processor is overloaded.

In order to reduce any unnecessary fragmentation of data, data will in general only be moved between contiguous subdomains. It is assumed that any processor is equally accessible from all other processors.

3 Related Work

Different techniques have been proposed for adapting parallel applications running on clusters. Different dynamic load balancing and migration strategies have been proposed.

There are many studies dealing with the problem of load balancing for distributed memory systems. Some work [6] assume that the processors involved are continuously lightly loaded, but commonly the load on a workstation varies in an unpredictable manner.

There are algorithms exist for scheduling parallel tasks. The Distributed Self Scheduling (DSS) [7] technique uses a combination of static and dynamic scheduling. During the initial static scheduling phase, p chunks of work are assigned to the p processors in the system. The first processor to finish executing its tasks from the static scheduling phase designates itself as the centralized processor and it stores the information about which tasks are yet to be executed, which

processors are idle and dynamically distributes the tasks to the processors as they become idle.

Alessandro [3] introduced a method to obtain load balancing through data assignment on a heterogeneous cluster of workstations. This method is based on modified manager-workers model and achieves workload balancing by maximizing the useful CPU time for all the processes involved.

Dynamic load balancing scheme for distributed systems [5] considers the heterogeneity of processors by generating a relative performance weight for each processor. When distributing the workload among processors, the load is balanced proportional to these weights.

The AppLes approach [1] uses parameterizable application and system specific models to predict application performance using a given set of resources. Using these models and forecasts of expected resource load, an AppLeS agent selects a resource set and an application schedule by evaluating candidate mappings. The mapping with the best expected performance is implemented on the target resource management system.

Many methods proposed in the literature to solve the load balancing problem are applicable to adaptive mesh computation. One of the earliest schemes was an iterative diffusion algorithm [4]. At each iteration, new load is calculated by combining the original load and the load of neighbouring processors. The advantage of this approach is, it requires local communication only, but the problem is its slow convergence. Several scratch-remap [8] and diffusion based [9] adaptive partitioning techniques have also been proposed. These different approaches are better for different system environments and different computational environments. In our approach, we try to identify sharp increases and to reduce them quickly as possible without necessarily achieving a perfect load balance.

4 Repartitioning with Minimum Data Movement

In this section, we describe our proposed approach. It operates on the processor graph which describes the interconnection of the subdomains of a mesh that has been partitioned and distributed among the processors.

We assume that an overloaded node initiates the load balancing operation whenever it detects that it is overloaded.

An important feature of our approach is to capture the need for the processor load to adapt very quickly to external factors for example, a key press or a mouse click may indicate that machine is no longer available. This is useful assuming we can use workstations only if the owner not using it and we need to move load when ever the owner returns. A subdomain is then deleted and the corresponding processor emptied of all load. If at a later time, the processor becomes available again a new subdomain may be created, possibly in another part of the graph.

If a processor is to be removed then the load has to be distributed onto neighbouring processors that are lightly loaded. The neighbouring processors are defined by the subdomain connectivities. The load to be distributed of partitioned into sections that are proportional to the load differences d_i of the neighbours

that are not already overloaded. The redistribution has two phases, the data partition and the data movement. The partitioning uses a greedy algorithm. In a typical finite element computation with an unstructured mesh distributed as subdomains, the partitioning starts from the subdomain boundaries adjacent to the lightly loaded neighbours and reallocates the old subdomain into appropriately sized sections. The mesh data is then transferred to the new subdomains, the processor connectivities are modified to take account of the new subdomain topology and the processor is released.

Those processors that are overloaded to a lesser degree, i.e. that need to shed some load but will remain as part of the computational cluster with a nontrivial load after the redistribution, also redistribute load to their lightly loaded neighbours using a similar greedy approach selected parts of the subdomain to be redistributed starting from the boundaries. These modifications may also result in changes to the subdomain topology and hence to the processor connectivities.

Additional processors, when available, may be (re)introduced at the point in the processor graph were the data movement is greatest.

5 Experimental Results

The experiments were performed on the problem of using the finite element method on an unstructured grid. Here we assumed that the computation is element based so that the load to be redistributed can be considered as repartitioning of the elements into subdomains, i.e. partitioning the dual graph.

Our proposed algorithm was implemented in *C* and *MPI* on 8 *Sun* workstations connected by 100 Mb/s Ethernet. All *Sun*s share a common file server and all files are equally accessible from each host due to the implemented NFS (Network File System). *Unix* provides a large amount of statistical information that can be used to describe a workload. Here we used a simple load sensor which uses *Unix* commands to collect the system information. The load sensor calculates percentage of unused CPU of each machine. Here we used a combination of processor speed and unused CPU amount as a load index. For loosely coupled linux clusters that do not incorporate NFS it same results can be gathered us-

Table 1. System Information

Processor	Speed Mb/s	Unused CPU	Load Index
1	300	99	29700
2	360	92	33120
3	333	99	32967
4	450	70	31500
5	333	99	32967
6	300	99	29700
7	450	71	31950
8	333	98	32634

Table 2. Initial Distribution

Processor	1	2	3	4	5	6	7	8
Load	540	540	604	540	444	617	540	495

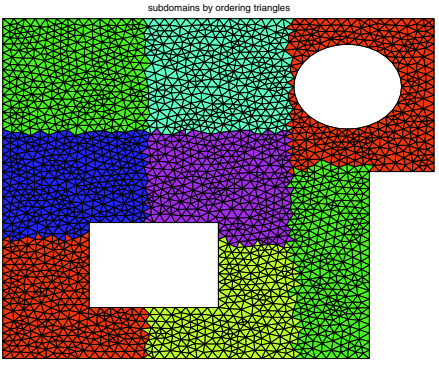

Fig. 1. Original Grid

Table 3. Test1: Running Times in milliseconds

Initial Partition	With Load Balance
16.7276	
	iteration 1 2.51483
	iteration 2 2.35808
	iteration 3 2.31213

ing software agents. In our environment, some machines can only be used if the owner is not using it and the processes should be moved if the user returns before they finish. If a mouse is moved or a key pressed, we need to move application workload from that particular machine and this information overides the normal load index. In order to determine when such a machine can be returned to the cluster, it is necessary to identify inactive time of a machine, this is achieved by a simple script in the background which gives the idle time of the machine.

The table 4 gives a typical snapshot of the system information on each machine in the cluster used in the experiments.

The experiment illustrated is a small finite element calculation, the initial partitioning of the grid into subdomains is shown in figure 1. The sizes of the subdomains are shown in table 5, the mean size is 540.

The finite element solution was computed iteratively, the times given in table 5 are for one iteration with the initial distribution and the first three iterations with a load balancing step between each iteration.

In the second experiment, the initial load was modified so that it was more unbalanced an the results shown in figure 2 illustrate how quickly the load on

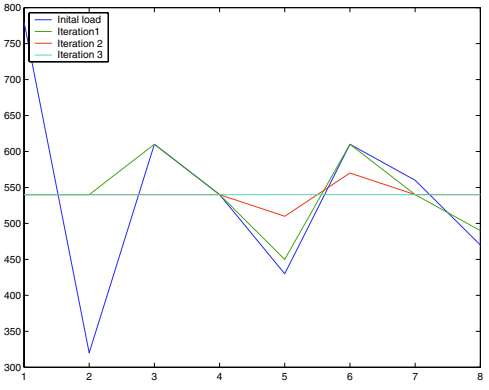

Fig. 2. Test2:Reduction of Load Spike

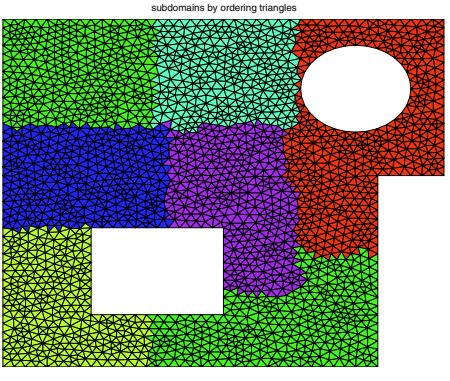

Fig. 3. Test3: Grid after removing one processor

Table 4. Test3: Redistribution of load

Processor	1	2	3	4	5	6	7	8
Load	737	618	618	619	539	588	601	0

a heavily overloaded node is reduced, again a single load balancing step was allowed after each iteration.

The results of the third experiment, in figure 3, show the redistribution of the load if one processor is removed, the distribution of the load is given in table 5 shows how the load is removed from the processor that is no longer available, it is not necessarily distributed evenly as the load indices of the machines may vary.

In the fourth test, the additional processor was reintroduced after several iterations when the load had become evenly balanced between the other processors (assuming equal load indices in this case).

Table 5. Test4: Reintroduction a processor

Processor	1	2	3	4	5	6	7	8
Iteration 1	618	618	617	617	629	618	603	0
Iteration 2	618	540	617	540	540	540	540	385
Iteration 3	540	540	540	540	540	540	540	540

Table 6. Different Load Index

Load Index	Run time
Processor speed x Unused CPU	2.51483
Unused CPU	6.2142

The final results in table 5 illustrate how the timings depend on the choice of load index.

6 Conclusions

Assuming that a single overloaded node has a greater effect on overall efficiency than a single underloaded node, we have presented an approach to load balancing that attempts to reduce an imbalance due to a load spike as quickly as possible. The experimental results show a performance improvement with the approach. According to the experimental results, we can see that the load index also plays an important role. Our future work includes experimenting on larger clusters with larger data sets.

References

1. `http://www.-cse.ucsd.edu/users/breman/apples.html/`.
2. `http://www.beowulf.org/`.
3. Alessandro Bevilacqua, *A dynamic load balancing method on a heterogeneous cluster of workstations*, Informatica **23** (1999), no. 1, 49–56.
4. G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, Parallel and Distributed Computing **7** (1989), 279–301.
5. Zhilling Lan and Valerie E. Taylor, *Dynamic load balancing of SAMR applications on distributed systems*, Scientific Programming **10** (2002), 319–328, no. 21.
6. C. K. Lee and M. Hamdi, *Parallel image processing application on a network of distributed workstations*, Parallel Computing **26** (1995), 137–160.
7. J. Lin and V. A. Saletore, *Self scheduling on distributed memory machines*, Super-Computing (1993), 814–823.
8. L. Oliker and R. Biswas, *Plum: Parallel load balancing for adaptive structured meshes*, Parallel and Distributed Computing **52** (1998), no. 2, 150–177.
9. Kirk Schloegel, George Karypis, and Vipin Kumar, *Multilevel diffusion schemes for repartitioning of adaptive meshes*, Journal of Parallel and Distributed Computing **47** (1997), no. 2, 109–124.