Branch and Bound Based Load Balancing for Parallel Applications

Shobana Radhakrishnan, Robert K. Brunner and Laxmikant V. Kalé
{rdhkrshn,rbrunner,kale}@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave. Urbana, IL 61801

Abstract. Many parallel applications are highly dynamic in nature. The computation and communication patterns change either slowly on abruptly during the course of computations. Such applications require an adaptive load balancing strategy. We are exploring an adptive strategy based on multi-partition object-based decomposition, supported by object migration. For a large class of applications that require such load balancing, the load varies relatively slowly (or infrequently) over time. It is then feasible to spend significant amount of computation time towards arriving at a close-to-optimal mapping of objects to processors. To utilize this opportunity, it is necessary to develop an object mapping strategy that can produce increasingly better solutions continuously. We present an optimal-seeking branch-and-bound based strategy that satisfies this requirement. They strategy takes as input a object communication graph, specifying the computation time for each object and the number and size of messages it sends to any other object. The strategy then continuously produces a stream of successively better mappings of objects to processors. One can stop the strategy from execution at any point, and take its current best solution as the new mapping. When communication costs are significant, we show that this strategy performs substantially better than several random and intelligent greedy strategies.

1 Introduction

Development of efficient parallel applications becomes difficult when they are either irregular or dynamic or both. In an irregular application, the computational costs of its subcomponents cannot be predicted accurately. Other applications are dynamic, and the computational costs of their subcomponents change over time. In either case, performance problems manifest themselves in the form of load imbalances. Although such imbalances are typically small and tolerable while running applications on a small number of processors, they often become major performance drains on systems with hundreds of processors. As the computational science and engineering community attempts to tackle a wider range of problems using parallel machines, and as the number of processors available for such simulations increases with technology, this problem of adapting program behavior to unpredictability and dynamic variations becomes much more important.

A solution to this problem that we have been exploring involves breaking the problem into a large number of chunks, such that the total number of chunks is significantly larger than the number of available processors. In fact, the size of a chunk can be decided independently of the number of processors, by using the criteria of keeping the communication overhead within a pre-specified bound. A system that supports data driven objects, (in our case, Charm++ [3]) is used to implement each chunk as an independent object. Thus, these objects send messages to other *objects*, in contrast to an MPI program (for example), which direct messages to specific processors. As a result, the runtime system is free to move the objects from one processor to another, without disturbing the application. The Charm++ system supports such migration of objects with automatic forwarding of messages, and with automatic minimization of forwarding overhead. With these prerequisites, namely the multi-chunk object-based decomposition, and object migration, all that one needs is a load balancing strategy that will decide when and where to move objects.

Even in irregular and dynamic programs, one can find a basis for predicting future events. Just as in sequential programs one can rely on the principal of locality, in a parallel program one can utilize the principal of "temporal persistence of computation and communication patterns". In irregular computations, each subcomponent's computation time may be unpredictable *a priori*, but once the program starts executing, each component will persist in its behavior over the iterations of the program. In dynamic applications, the behavior of a component changes, but even here, either the behavior changes slowly over time (as in molecular dynamics applications [4], where the load may shift slowly as the atoms move) or abruptly but infrequently, (as in adapting refinement strategies). In either case, it is a reasonable heuristic to assume such a persistence of behavior, over some horizon in the future. This is not unlike the idea of using caches based on the principal of locality and working sets. Although the program may jump out of its working set from time to time, the caching technique, which assume that the data referenced in the recent past will continue to be referenced, still pay great performance dividends.

Based on the above performance prediction principle, we have developed a framework that facilitates development of such strategies. The framework provides automatic measurement of computation times and automatic tracing of communication events of a parallel object program. A load balancing strategy can tap into this framework, obtain the necessary data, and decide to migrate some objects to new processors. The framework also facilitates implementation of such decisions by providing mechanisms to migrate individual objects. The framework is broad enough to admit a wide variety of strategies.

Within the context of this framework, we are engaged in developing a suite of load balancing strategies, and applying them in a variety of applications. It is clear that different class of applications will require different load balancing strategies. For example, for applications running on multi-user workstation clusters, the strategies must adapt to the extraneous load presented by the jobs of other users [2]. On a dedicated parallel machine, some applications may change their behavior so rapidly that only a highly localized, continuously monitoring receiver-initiated strategy will suffice. In other contexts, periodic rebalancing may be acceptable, as long as it is done often enough. Here, a balancer based on a quick heuristic is necessary, because otherwise the frequent nature of the load balancing will make the cost of the decision itself a substantial drain on performance. However, there exists a significant class of applications, where only periodic, *infrequent* rebalancing is necessary.

As an example, our experience with molecular dynamics for biophysical simulations shows that the load balance stays relatively stable over several hours as the atoms slowly migrate over domain boundaries. In such a situation, spending as much as a few minutes on deciding a new mapping is not that expensive. The question is whether it is possible to produce a higher quality solution, beyond what simple heuristic strategies can produce, by expending more computation time. The problem of optimum mapping is NP-hard. So, even with minutes of time on a parallel machine it typically will not be possible to find the optimal solution. One thus appears to be stuck between the bimodal choice of a low-cost low-quality heuristic method on one hand, and unrealistic optimum-finding algorithms on the other hand. This paper presents a branch-and-bound based strategy that fills in the middle ground: depending on the available computation time, it can produce a continuum of solutions from the simple heuristic ones to provably optimal ones.

In the next section, we describe the object model that our framework and the branch-and-bound strategy are based on. The algorithm itself, along with several key optimizations, are described in the next section. Section 4 describes the performance attained by the new strategy, and compares it with some heuristic strategies. The branch-and-bound algorithm itself is implemented as a parallel Charm++ program, so as the utilize all the available compute power for load balancing.

2 The Object Model

Efficient communication and work-load balancing are prerequisites to utilizing the full performance of parallel and distributed systems. In such high-performance applications it is critical to be able to balance the load, comprised of both computation and communication. Thus, a load balancer tries to achieve an optimal distribution of load across processors in such a way that the task can complete in the least possible time.

This section describes how our algorithm approaches the load balancing problem, by modeling parallel applications as a collections of computation objects which communicate among themselves. Communication costs between objects are modeled based on the characteristics of the particular machine, and objects on the same processor are assumed to exchange data for free. Furthermore, the load balancer has the freedom to reassign these objects to any processors to optimize program performance.

The objects that are to be balanced are represented as a network of communicating entities in the form of a directed graph. Graph-based models have been used earlier for the task allocation problem [1]. (Also, Metis [5] provides a graph based partitioning scheme that is meant for partioning large, million-element unstructured meshes.) The vertices in the graph represent the computation cost of the objects to be balanced and the edges represent a pair (number of messages, total bytes sent) of each communication. Since it is a directed graph, each pair of edges may have zero, one or two edges connecting them but if more than one, they are in opposite directions. If both the sending and receiving entities are assigned to the same processor, the model assumes that no time is required for communication; otherwise, the sender and receiver pay:

$$T_{send} = \alpha_{send} \cdot N_{messages} + \beta_{send} \cdot N_{bytes}$$
$$T_{receive} = \alpha_{receive} \cdot N_{messages} + \beta_{receive} \cdot N_{bytes}$$

In addition to migratable objects and communication patterns, our object model also includes the following features:

1. **Non-migratable Objects:** Non-migratable objects are objects which must remain on particular processors throughout their lifetime. The load balancers still considers their computation and communication cost, but does not have the freedom to move them.
2. **Proxy Communication:** This refers to a kind of communication where several objects require data from a particular object. Should the receiving objects be placed on the same processor, a single message may supply the data to all of the receivers. We have implemented this by adding an attribute, the proxy-id, for each message arc. While calculating the communication cost resulting from the assignment of an object to a processor, we ignore the cost if some message from this sending object to another object residing on the same processor as the object being currently assigned having the same proxy-id has already been accounted for.
3. **Background load:** In some applications, a significant amount of work may not be easily attributed to any object, although the total time spent which is not accounted for by objects can be determined. The load balancers can take this time into account while distributing objects.

Given this kind of input about the load and the processors, the load balancer tries to achieve an optimal load distribution within a limited search time. This object model provides enough information to implement a variety of load balancing algorithms.

3 Branch and Bound Algorithm

This section describes the algorithm for load balancing we implemented based on the branch and bound strategy for solving search problems. Here the problem of finding a good load distribution for the processors is modeled as an optimization problem. While the greedy algorithm leads to a local optimum at each step and therefore may lead to a suboptimal overall distribution, the branch and bound algorithm always leads to the optimal distribution. appreciably.

The algorithm uses a "state" data structure to store a partial mapping decision, and its consequences. A *state* contains the following components.

- cost[p]: for each processor p, cost[p] is the sum of the computation costs of all the objects assigned to that processor and the communication cost due to the interaction of these objects with those residing on other processors.
- map[i] for each object i map[i] represents the processor to which is has been assigned in the state under consideration. (may be -1 for unassigned objects).
- lowerBound: This is the maximum of all the cost[p] values
- nextObject: This is the next object to be considered for assignment
- stepsLeft: This is an integer that indicates the number of unassigned objects in this state.
- minRemaining[p]: This indicates the potential additional cost that the processor p would incur for sure in any solution that extends the current state.
- totalCost: This is simply the sum of all costs[p]

Except the first two arrays, all others are derivable from these, but have been added as part of the state itself as their values are required often and it computationally less expensive to maintain them this way. The pseudocode for the branch and bound algorithm is given below:

```
Initialization:
 - Initialize state S to empty (no object is assigned yet)
 - Define the initial upperBound using a greedy algorithm
 - Assign, in S, all non-migratable objects to
their current processors
 - Search(S)

search(S)
  x = nextObject(S)
  for each processor p (between 0 and P-1)
     copy S into S1
     assign x to p in S1
     if S1 is viable (lowerBound(S1) < current upperBound)
       if S1 is a complete mapping
          replace current UpperBound with S1
       else
         search(S1)
     else (S1 is not viable. Prune (i.e. ignore) it.
```

The recursive call to search in this simple recursive formulation can be replaced by (a) parallel search via creation of an object to search under S1 or (b) a best-first formulation, where the next best state is selected for exploration at each stage. Both of these variants have been implemented in our framework.

Termination of the algorithm: To provide the flexible tradeoff between decision time and solution-quality, we limit the algorithm to a caller-specified time limit. Although this does not let the algorithm pursue all possible states, our optimized algorithm still gives the solution quite close to optimal as compared to the other algorithms we have implemented.

3.1 Optimizations

The following optimizations have been implemented in the load balancer, improving its speed beyond the basic scheme described above.
Sorting objects before assignment: The objects are ordered in decreasing sequence of their computation costs for assignment. Thus, the heavier objects are assigned at higher levels of the search tree.
Search ordering: In the simple recursive formulation, at each node, the children are ordered in decreasing order of their lowerBounds. I.e. the child that assigns the new object to the least loaded processor is considered first.
Greedy Initial Estimate: Instead of starting from a default initial state and then updating with the first solution obtained, thereafter pruning states based on this, we have used a quickly obtained greedy estimate as the initial lower bound which has led to pruning more states and thereby a faster implementation.
Symmetry: If all the processors have identical communication and computation capacities, then any processor with no assigned objects is equivalent to another such processor. This reduces the branching factor of the tree at the top levels.
Future-Cost Estimates: Instead of just using the costs so far of the processors to arrive at the lowerBound, we also use an estimate of future cost (`minRemaining[p]`), which leads to effective pruning. When an object i is being assigned to processor p, its computation cost is incremented by the computation cost of i. The cost is also updated by adding the cost incurred due to messages sent between i and objects that have already been assigned to any processor other than p. Now, in order to get the estimate of the minimum remaining cost on processor p due to the additional knowledge of the assignment of i to p, we add the minimum of the computation and communication costs of each unassigned object j that communicates with i. This is the additional cost that this processor will incur, irrespective of whether j is assigned to p or not. (when j is finally assigned to some processor, this component is subtracted from minRemaining[p]). The lower bound of a state is then computed as the maximum of cost[p] + minRemaining[p] over all P.
Greedy Heuristic: Switching from using computation cost to communication cost of objects when the number of assigned objects exceed L/2 while obtaining the greedy estimate. This is because, as we are considering the objects in decreasing order of their computation cost, we expect the communication cost to become a more predominant factor when we come to this. Using this heuristic has helped in making the greedy solution a better one to start with and thereby leads to elimination of a greater number of states. Other greedy variants are worth exploring in future.

The following ideas were also explored, but found to be non-productive.

Narrowing the search space: As suggested by Wah and Yu [6], one could narrow the search space by aiming for a solution guaranteed to be within a small percentage (say 2%) of the optimal. This is accomplished by comparing the lower bound to `0.98 x upperBound` in the pruning step. In the context of our strategy, which uses a fixed time limit, such a narrowing may seem to be even more beneficial, as it allows the search to "sample" a larger portion of the search space. However, in almost all the runs we conducted, with using 1, 2 and 4 percent tolerance, we found no improvement in solution quality within fixed time.
Refinment: Due to time constraints applied to the branch-and-bound algorithm, it will not necessarily give the exact optimal solution. We attempted applying refinement (See the performance section for a description of "refinement"). to the solution found by branch-and-bound so that the efficiency of this could be improved further. As the solutions for the cases we investigated were already close enough to the optimal, we observed that the refinement did not improve the efficiency of the obtained solution significantly.

4 Performance Results

In this section, we compare the branch and bound load balancer with four other algorithms. These algorithms include:

1. **Greedy:** This algorithm uses the greedy heuristic of Section 3.1, without performing the branch and bound search.
2. **Random:** Objects are randomly distributed among the processors.
3. **Greedy-Refine:** The greedy algorithm is run to obtain an initial distribution, and then a refinement procedure is applied. The refinement procedure looks at each processor with a load above the average by a certain threshold, and moves objects from them to underloaded processors, without making them overloaded, until no further movement is possible.
4. **Random-Refine:** The refinement procedure is applied to the solution found with the random algorithm.

4.1 Load balancer performance

Table 1 shows the results obtained when runs were made of the sequential implementation of the branch and bound strategy using a recursive method for various cases: In all cases, the same object graph is used, with 100 objects, and with a randomly generated computation cost and communication volumes. The efficiency refers to the ratio of the time that a sequential execution of the objects would take versus a parallel execution of the same. As can be observed, even when run just to a limited time(not investigating the entire search tree), the branch and bound strategy gives the most efficient solution among the algorithms implemented, even though it often pays a higher communication overhead (see Figure 1).

Case #	Procs.	Comm. Cost	Greedy	Greedy-Refine	Random	Random-Refine	Branch & Bound
1	9	0	99.7	99.7	69.1	69.1	99.8
2	20	0	98.4	98.4	57.5	57.5	99.4
3	9	120	51.4	55.6	58.5	68.6	81.0
4	20	120	28.8	31.7	50.6	67.7	78.4
5	9	250	34.4	37.0	48.4	55.9	64.4
6	20	250	26.3	28.5	41.2	44.7	60.1
7	9	300	37.1	40.9	46.0	50.9	60.3
8	20	300	26.7	30.0	39.1	42.1	56.2
9	9	400	44.2	52.2	41.8	50.5	54.6
10	20	400	21.2	24.0	35.4	36.9	49.6
11	9	500	26.9	28.9	38.4	46.4	49.5
12	20	500	27.4	30.0	32.3	42.3	43.7
13	9	600	29.9	34.7	35.4	41.7	44.3
14	20	600	13.6	14.2	29.6	38.0	39.5
15	9	700	20.9	22.2	32.9	38.4	41.1

Table 1. Efficiency

It can be inferred from the results that the efficiency of the solution for each algorithm decreases as the communication overhead increases. This is because the maximum efficiency we can achieve ideally itself goes down with increase in the communication overhead. For all the cases, branch and bound gives the best solution as compared to the others.

We also monitored the quality of solution as a function of time spent by the load balancer. As expected, the quality increases with more search, but at some time it stabilizes to an optimum value. It can be verified from small problem instances, that the time beyond this point, spent on proving the optimality of the solution, is huger compared with the time spent in obtaining the solution. This is consistent with observations in the OR community regarding hard search problems.

We also compared the efficiencies of the solution obtained by considering the communication cost versus one without considering it. We found that for a run with random-refine strategy, while considering communication cost led to an efficiency of about 39 percent, not considering it gave one with efficiency of about 33 percent. Thus, as we had expected, the efficiency increases while communication cost is considered. We have also noticed that the refinement applied after getting a solution, does not require much computation time, but can give a much better solution in many cases. For example, in most cases the refine applied to greedy takes about 1sec more and gives an improvement of about 10percent on an average. For random the time taken is more but the improvement it gives is also more. For example for case 3, it gives an improvement of about 17 percent, taking about 1 sec more than pure random.

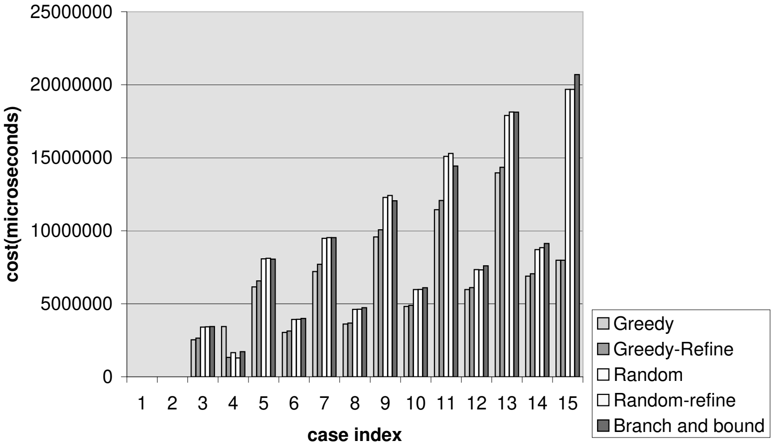

Fig. 1. Average Communication Cost Per Processor

4.2 Parallel Branch-and-bound

We have implemented a parallel version of the branch-and-bound algorithm, as described in the section above. A "grain-size" parameter is used in the parallel search, which is the depth of the node in the search tree below which the search switches from parallel(spawning objects) to sequential(plain recursive) mode. This grainsize control is useful to amortize the cost of parallel object creation. A parallel search allows us to search a P-fold larger search space within the same time limit using P processors, which are available anyway. Quantitative measurements of effectiveness of parallel search are difficult due to the non-complete search used in a time-limited procedure. We plan to study this issue further in the context of large applications.

4.3 Application in Parallel Program

All the above results were for runs made of the different algorithms implemented using simulated data. We have also integrated the branch and bound strategy with a parallel benchmark program and the load balancing framework described in the introduction. The program used is similar to a ring program. A certain number of concurrent objects are created (this number can be specified at the command line). These perform computation and communicate with each other in a pre-determined fashion. The computation time is generated randomly at run time. The communication pattern is as follows. The i'th object sends a pre-defined sized message to the (i+1)th object(mod the number of objects) and also to every third element from then on. Using automatic load and communication measurement provided by the framework, the branch-and-bound strategy successfully redistributed objects to processors with improved performance. More extensive quantitative studies are being conducted.

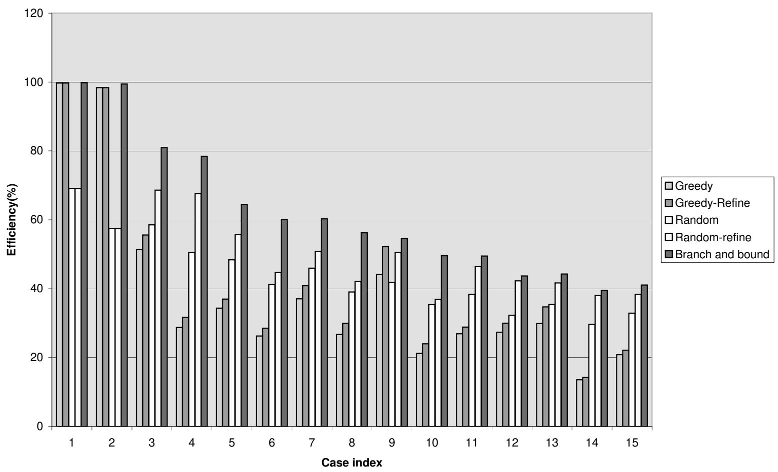

Fig. 2. Efficiency

References

1. P. M. A. Sloot A. Schoneveld, J. F. de Ronde. Preserving locality for optimal parallelism in task allocation. In *HPCN*, pages 565–574, 1997.
2. Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.
3. L. V. Kalé and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.
4. Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal Computational Physics*, 1998. In press.
5. George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of Supercomputing '96*, Pittsburg, PA, November 1996.
6. B. W. Wah and C. F. Yu. Stochastic modeling of branch-and-bound algorithms with best-first search. *IEEE Transactions on Software Engineering*, 11:922–934, 1985.
7. Chengzhong Xu and Francis C. M. Lau. *Load Balancing In Parallel Computers Theory and Practice*. Kluwer Academic Publishers, 1997.