Branch and Bound Based Load Balancing for Parallel Applications

Shobana Radhakrishnan, Robert K. Brunner, and Laxmikant V. Kalé

University of Illinois at Urbana-Champaign, Urbana, IL, USA

Abstract. Many parallel applications are highly dynamic in nature. In some, computation and communication patterns change gradually during the run; in others those characteristics change abruptly. Such dynamic applications require an adaptive load balancing strategy. We are exploring an adaptive approach based on multi-partition object-based decomposition, supported by object migration. For many applications, relatively infrequent load balancing is needed. In these cases it becomes economical to spend considerable computation time toward arriving at a nearly optimal mapping of objects to processors. We present an optimal-seeking branch and bound based strategy that finds nearly optimal solutions to such load balancing problems quickly, and can continuously improve such solutions as time permits.

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, with the computational costs of their subcomponents changing 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 a large number of processors.

We have been exploring a solution to this problem that 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 criterion of keeping the communication overhead within a pre-specified bound. A system that supports data driven objects, (*e.g.* Charm++ [1]) 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 directs 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. Charm++ supports such migration of objects with automatic and optimized forwarding of messages. With these prerequisites (multi-chunk object-based decomposition, and support for object migration), all that one needs is a strategy to decide when and where to move objects.

Even in irregular and dynamic programs, one can find a basis for predicting future performance. Just as in sequential programs one can rely on the principle of locality, in a parallel program one can utilize the principle 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, or abruptly but infrequently (as in adaptive 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 principle of locality and working sets. Although the program may jump out of its working set from time to time, the caching technique, which assumes that the data referenced in the recent past will continue to be referenced, still pays large performance dividends.

Based on the above performance prediction principle, we have developed an adaptive load balancing framework. It provides automatic measurement of computation times and automatic tracing of communication events of a parallel object program. A load balancing strategy can obtain the necessary performance data from this framework, and decide to migrate some objects to new processors.

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. Different classes of applications require different load balancing strategies. In a significant class of applications, focused on in this paper, only periodic, and *infrequent* rebalancing is necessary.

Our experience with molecular dynamics [2] for biophysical simulations shows, for example, 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. However, 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 provably optimal solution. One thus appears to be stuck between the bimodal choice of, a low-cost, low-quality heuristic method, or an unrealistic, optimum-finding algorithm. 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.

2 The Object Model

This section describes how our algorithm approaches the load balancing problem, by modeling parallel applications as 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 (*e.g.* [3]). Also, Metis [4] provides a graph based partitioning scheme that is meant for partitioning large, million-element unstructured meshes. The vertices in the graph represent the computation cost of the objects to be balanced and each edge represents communication, parameterized by the pair <*number of messages, total bytes sent*>. If the sending and receiving objects are assigned to different processors, the processors are charged:

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

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. Load balancers should still consider their computation and communication cost as background load, but do not have the freedom to move them.
2. **Proxy Communication:** This refers to multicast communication where several objects require data from one particular object. Should the receiving objects all be placed on the same processor, a single message may supply the data to all of the receivers. We model 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 of an incoming multicast communication arc if another recipient of the same multicast has already been assigned to this processor.

3 Branch and Bound Algorithm

Branch and bound algorithms are a good choice for load balancers, because they exhibit the property that they can produce an optimal solution if given enough time, but produce “good” sub-optimal solutions if stopped prematurely. To provide the flexible tradeoff between decision time and solution-quality, we limit the load balancing algorithm to a caller-specified time limit. Although this usually 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.

Our branch and bound load balancer follows the design of common branch and bound algorithms, with the addition of a few optimizations particular to the load-balancing problem.

- **Sorting objects before assignment:** The objects are ordered in decreasing sequence of their computation costs for assignment. Thus, more expensive objects are assigned at higher levels of the search tree.
- **Search ordering:** At each level of the search tree, the child that assigns the new object to the least loaded processor is considered first.
- **Greedy Initial Estimate:** States are pruned based not on the first state evaluated. Instead, a quickly-obtained greedy estimate is used as the initial lower bound which results in more states being pruned early.
- **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 of states previously assigned to obtain the current lower bound, we compute an optimistic estimate of the cost of assigning the remaining states to obtain a more accurate lower bound, which allows the search to prune more states.

As suggested by Wah and Yu [5], one could narrow the search space by aiming for a solution guaranteed to be within a small percentage (say two percent) of the optimal. This is accomplished by comparing the lower bound to $0.98 \times$ *upper_bound* 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.

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 a greedy heuristic 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. This refinement procedure looks at each processor with a load above the average by a certain threshold, and moves objects from them to under-loaded processors, until no further movement is possible.
4. **Random-Refine:** The refinement procedure is applied to the solution found with the random algorithm.

All of these algorithm (except Random) consider the processor overhead of communication in the assignment process, in accordance with the cost model in Sect. 2.

Table 1 shows the results obtained when runs were made of the sequential implementation of the branch and bound strategy using a recursive method

Table 1. Efficiency

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

for various cases. In all cases, the same object graph is used, with 100 objects and randomly generated computation cost and communication volumes. The *efficiency* is calculated as $T_{sequential}/(P \cdot T_{parallel})$, where P is the number of processors, and $T_{parallel}$ is computed by taking communication into account. We observe that, even when run for limited time (so that the search tree is not exhaustively searched), the branch and bound strategy gives the most efficient solution among the algorithms implemented.

From these results, we observe that the efficiency of the solution for each algorithm decreases as the communication overhead increases. This occurs because the optimal efficiency itself goes down with increase in the communication overhead.

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 converges on an optimum value. It can be verified from small problem instances, that further search time spent on proving the near-optimality of the solution quickly exceeds the time savings resulting from the slightly improved load balance. This result is consistent with observations in the operations research community regarding hard search problems.

We observed that applying the refinement algorithm does not greatly increase the time spent by any of the load balancing algorithms, but produces a much better solution in many cases. For example, in most cases the refine applied to Greedy takes about 1 second more, and results in about 10 percent efficiency improvement. For Random, refinement requires proportionally more time, but the resulting efficiency is improved even more dramatically.

Often load balancing strategies concentrate on balancing computation costs alone. To understand the effects of ignoring communication, we evaluated the performance of the algorithms after modifying them to ignore communication costs. For instance, we found that the modified Random-Refine strategy led to an efficiency of 33 percent, compared to 39 percent obtained with original strategy.

5 Summary and Planned Work

In this paper, we presented a branch and bound based strategy that uses this data to generate a near-optimal mapping of objects to processors. This strategy is a component of our object based load balancing infrastructure to effectively parallelize irregular and dynamic applications. The framework instruments parallel programs consisting of intercommunicating parallel objects, and collects performance and communication data. A useful property of the branch and bound strategy is that it is tunable: it has the ability to use the available time to produce increasingly better mappings. Also, the object communication costs are fully modeled. Intelligent greedy strategies were also developed, and are seen to be quite effective. The new strategy performs satisfactorily, irrespective of the communication to computation ratio.

The branch and bound algorithm itself is suitable for execution in parallel; indeed we have developed such a parallel variant. Using this, we plan to conduct extensive performance studies. In particular, we plan to perform further studies using various parallel machines and applications, rather than just the simulation model described in this paper.

Due to space limitations, this paper does not include a survey and comparison with the extensive load balancing literature. We only note that many strategies described in the literature are either not oriented toward an object-graph model or do not present a tunable strategy.

References

1. L. V. Kale 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. 194
2. 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 of Computational Physics*, 1998. In press. 195
3. P. M. A. Sloot A. Schoneveld, J. F. de Ronde. Preserving locality for optimal parallelism in task allocation. In *HPCN*, pages 565–574, 1997. 196
4. George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proc. Supercomputing '96*, Pittsburg, PA, November 1996. 196
5. B. W. Wah and C. F. Yu. Stochastic modeling of branch-and-bound algorithms with best-first search. *IEEE TSE*, 11:922–934, 1985. 197
6. Chengzhong Xu and Francis C. M. Lau. *Load Balancing In Parallel Computers Theory and Practice.* Kluwer Academic Publishers, 1997.