Reinforcement Learning for Selfish Load Balancing in a Distributed Memory Environment

Stephen M. Majercik and Michael L. Littman

20 December 1996

Abstract

Load balancing is a difficult problem whose solution can greatly increase the speedup one achieves in a parallel distributed memory environment. The necessity for load balancing can arise not only from the structure or dynamics of one's problem, but from the need to compete for processor time with other users. Given a lengthy computation, the ability to exploit changes in processor loads when allocating work or deciding whether to reallocate work is critical in making the computation time-feasible. We show how this aspect of the load balancing problem can be formulated as a Markov decision process (MDP), and describe some preliminary attempts to solve this MDP using guided off-line Q-learning and a linear value-function approximator. In particular, we describe difficulties with value-function approximator divergence and techniques we applied to correct this problem.

1 Introduction

In order to use multiple processors efficiently, we need to be able to balance the workload on these processors such that the fewest number of processors are idle at any given time while there is still work to be done. This load balancing can be viewed from three perspectives. The first, which we call *altruistic load balancing*, is an operating systems perspective. Given a distributed memory environment, such as a network of workstations, and the arrival of a series of processes with varying interprocess communication needs, the goal is to schedule the processes on the network in order to minimize the time to completion of *all* processes.

The second perspective, which we call *solipsistic load balancing*, is essentially a scientific computation perspective. This viewpoint assumes that one computationally intensive, parallelizeable computation has exclusive use of all the processors on the system, and the goal is to distribute the computation on these processors in order to minimize the time to completion of this *single* computation.

The need for such load balancing generally arises when a problem is defined on a domain whose subdomains have very different computational costs associated with them. These differences can be the result of a problem decomposition driven by the physics of the problem or by the use of multiple computational methods, or may arise as the solution is computed if the dynamics of the problem concentrate the necessary computations into a particular subdomain.

Both altruistic and solipsistic load balancing have an extensive literature. There is, however, another reason for load balancing scientific computations which has not been explicitly considered in the literature. This third perspective, which we call *selfish load balancing*, shares with solipsistic load balancing the focus on completing one particular computation as fast as possible but, like altruistic load balancing, recognizes that there are other jobs on the system and that no one job has exclusive use of the processors. The aim of this type of load balancing is to distribute a particular computation such that, given the varying demands of other jobs on the system, we minimize the time to completion of this computation.

For computations which would consume relatively small amounts of wall-clock time on a dedicated processor (several hours or less), the gain in parallel time from this type of load balancing may not be worth the overhead incurred by the balancing process. But for computations that would take months, or even years, of wall-clock time, such as large molecular dynamics simulations, the benefits of load balancing can greatly outweigh the costs.

2 Molecular Dynamics

We will focus on selfish load balancing for the Molecular Dynamics Multipole Algorithm (MDMA), an efficient multipole-based algorithm for computing accurate nonbonded forces among particles in a molecular dynamics (MD) simulation [1]. This focus is appropriate since the calculation of nonbonded forces consumes approximately 90% of the execution time in an MD simulation [2].

The MDMA constructs a tree based on a hierarchical decomposition of the simulation space. Starting at the root, levels of the tree represent increasingly fine decompositions of the simulation space. For example, the root represents the entire simulation space while the eight children (in 3-D) of the root represent the eight octants obtained by dividing the volume evenly along all three dimensions.

Once the tree has been constructed, a time step in the simulation is accomplished by an upward pass through the tree, during which multipole expansions are computed, and a downward pass, during which the multipole expansions are translated and accumulated. Given the total force acting on each molecule computed by the previous phase, the MDMA can now update the velocity and position of each molecule.

We will make certain assumptions that allow us to focus on load balancing due to imbalances arising from other jobs on the system rather than imbalances arising from the molecular dynamics.

- We assume a uniform distribution of molecules throughout the simulation, and that the occasional particle that needs to be transferred to another cell does not represent a significant communication cost. Thus, we always have a balanced tree with the appropriate level of granularity, so we never have to rebuild the tree. (If a buffer zone is constructed around the simulation volume, the tree usually does not need to be rebuilt for hundreds or thousands of time steps [1], making this assumption somewhat realistic.)

- We assume that the distributed system is small enough to allow a centralized load balancer, i.e. the overhead associated with collecting the processor information necessary to load balance is significantly smaller than the potential gain from load balancing.

Given this simplified molecular dynamics problem, most interprocessor communication is eliminated if cells from the third level of the oct-tree (along with all their descendants) are assigned to the available processors [1]. Thus, load balancing for 64 or fewer processors means allocating the 64 subtrees rooted at the level-3 cells to the processors so as to minimize the time spent in simulating the motions of the molecules or, equivalently, to maximize, for each unit of system time, the number of processors that have a portion of our simulation in their run queue.

3 The MD Simulation Model

We constructed a simulation to model the allocation and execution of the work required in an MD simulation in a multiple processor environment, and applied reinforcement learning techniques to optimize the behavior of an agent controlling this allocation process.

Our simulation models the following process. Given n processors and t MD steps to compute (each of which will take multiple processor time steps), an agent allocates the MD work among the processors at the beginning of each MD step. The agent monitors the processor loads (as measured by the number of jobs in the processors' run queues) and the progress of the processors in completing the MD work (but only insofar as it knows whether a given processor still has MD work, not how much it has). At each processor time step, the agent decides whether to let the computation continue or to intervene and reallocate the work, losing all the computation done so far on this MD step in an attempt to correct what has become a poor allocation.

The simulation uses quantized wall-clock time as the governing time scale (1 quantum = 5 seconds). We quantify the amount of work allocated to each processor by estimating the total dedicated processor time needed to calculate the MD step. Each processor p_i receives J_i work, a fraction of that total time (proportional to the number of subtrees that would be allocated to that processor).

During each quantum, once the work has been allocated and until all the MD work in the MD step has been completed, the simulation does the following:

1. Updates the size of the processor run queues assuming Poisson arrivals with a specified mean that varies across processors and according to the time of day, and exponential service times with a specified mean that does not vary. This allows us to simulate processors with different usage patterns and different processing speeds.

2. Calculates f_i, the fraction of the quantum available for the MD computation on processor p_i:

$$f_i = e^{-\beta_i r_i}, \tag{1}$$

where β_i is a constant and r_i is the number of jobs in the run queue of p_i. Thus, the amount of MD work that gets done in a quantum, w_i is:

$$w_i = f_i q, \tag{2}$$

where q is the wall-clock duration of a quantum. The amount of MD work remaining on processor i, J_i, is decremented by w_i.

3. Advances wall-clock time a single quantum.

This process of simulating an MD step is repeated until we have simulated the number of steps required in the MD simulation.

The decision required of the agent at each quantum is whether to reallocate the cells of the tree to the processors. The consequences of the decision differ depending on whether we are at the beginning or in the middle of an MD step. If we are at the beginning of an MD step, a decision to rebalance may incur two types of costs, a cost due to the time needed to gather load information from the processors and a cost to redistribute the cells among the processors (time during which no actual computation is being done).

If we are in the middle of an MD step, the agent has the option of rebalancing, but only by completely restarting the simulation of that MD step. Thus, a decision to rebalance in the middle of an MD step incurs the additional cost of the wasted computation up to that point. Since an MD step for a 100,000 particle simulation can take on the order of an hour [1] on a dedicated processor, however, the decision to rebalance in the middle of an MD step might make sense in some situations.

4 Load Balancing as an MDP

Given the description of the simulation model in Section 3, the MDP formulation is relatively straightforward. A state s is a vector $s = (\vec{L}, \vec{D}, \vec{J}, H, M, Q, R, B)$, where:

- $\vec{L} = (L_1, \ldots, L_n)$ and L_i indicates the load on processor i, expressed as the number of jobs in the run queue of that processor,
- $\vec{D} = (D_1, \ldots, D_n)$ and D_i indicates the amount of MD work currently allocated to processor i,
- $\vec{J} = (J_1, \ldots, J_n)$ and J_i indicates the amount of MD work remaining to be done on processor i in the simulation of the current MD step,
- $H : M$ is the 24-hour wall-clock time,
- Q is the number of quanta used so far in an MD step,
- R is a binary scalar indicating whether we are at the beginning of an MD step, and
- B is a bias feature.

If the agent were given complete freedom to allocate and shift work among processors, the number of possible actions would be potentially enormous. We reduce the size of the action set by limiting the possible actions to the application of one of two balancing heuristics — even allocation of work among processors and allocation according to a Boltzmann distribution. In the latter allocation method, processor p_i gets a fraction of the total work equal to $\exp(-r_i)/(\sum_i \exp(-r_i))$, where r_i is the number of jobs in the run queue of p_i. In addition, the agent has two other action choices. Once at least one work allocation has been done, the agent can choose to simply repeat the previous allocation. And, finally, the agent can always do nothing.

From this set of actions, the agent can formulate more complicated allocation policies, e.g. choosing to use the less costly even allocation when the processors are all similarly loaded and the time of day is such that this situation is likely to continue. This idea of framing actions as policy applications and forming a more complex policy from the simpler action policies has been used before [3], although we do not allow policies to be mixed to form hybrid actions.

Costs are both explicit and implicit. An explicit cost (1 or 10 in the tests conducted) is assessed for each action taken, and an explicit reward (1000 in the tests conducted) is awarded when the entire sequence of MD steps has been computed.

There is an additional implicit cost for choosing to allocate according to a Boltzmann distribution. Allocating using this method (unlike allocating work evenly, or according to the allocation of the previous MD step) we need to know the size of each processor's run queue. The time needed to poll the processors to obtain this information is reflected in a delay of 5 quanta when this allocation method is chosen; i.e. the work becomes available on the processors 5 quanta after this allocation method is chosen. (The allocation, however, is based on the run queue information obtained when the allocation decision was made. This reflects one of the primary obstacles encountered in load balancing—obtaining and acting on system information before it is out of date.)

5 Solving the MDP

We use guided off-line Q-learning with linear function approximation to solve the MDP. The learning is guided in that actions are chosen with a bias toward the naively correct action. At the beginning of an MD step, the agent chooses to allocate the work with a probability of 0.8 and to do nothing with a probability of 0.2. Once the work had been allocated, the agent chooses to do nothing with a probability of 0.9 and chooses to reallocate the work (losing any work done on this MD step so far) with a probability of 0.1. The update equation for the linear-function coefficients c_i is:

$$c_i = c_i + \alpha x_i dv, \tag{3}$$

where α is the learning rate, x_i is the i^{th} feature of the current state vector, and dv is the difference between the target value and the value of the current state (standard delta rule).

Previous work with function approximation [4] suggests that using a learning rate which decays at an appropriate rate is important for ensuring convergence of the function approximator. We used a decaying learning rate over 1000 epochs of training, where the learning rate α_m for epoch m was $\alpha_m = 0.001/(m+1)$. Although the coefficients of the value-function approximator did not converge within 1000 epochs, their rate of growth steadily decreased. Tests of the resulting value functions, however, indicate that a learning rate which decays sufficiently rapidly to prevent divergence will also prevent useful learning. Agents using value functions produced with this technique are not able to direct an MD simulation through even one step. In fact, in every test case, the agent refrains from making even an initial allocation for the first MD step.

When we tried training with a constant learning rate, we found that for most reasonable settings of the problem parameters, the coefficients of the value-function approximator grew explosively. Again, agents using the value functions produced were not able to guide the MD simulation through even one step. Our analysis indicated that the cause of this value function divergence is the presence of three features in the state which always, or almost always, increase during training and whose magnitude can become large relative to the magnitudes of the other features in the state. In particular, there are relatively long periods during training during which all three of these variables are constantly increasing.

Since these features quickly become substantially larger than the other features, they tend to dominate the value of a given state, and their tendency to increase during the training period translates into a tendency for the state values to increase during the training period. Increasing state values produce larger value function coefficients as well which, in turn, exacerbate the increase in the state values produced by the increasing features. This positive feedback cycle leads to divergence.

The problem is treated more effectively by changing the update rule (Eq. 3) to include a sigmoid which squashes dv, making the update rule:

$$c_i = c_i + \alpha x_i \text{ sigmoid}(dv). \tag{4}$$

This novel update rule has the beneficial property that when the error dv is close to zero (as it should be if learning is successful), updates are consistent with Eq. 3, but when errors are large, the c_i coefficients are not changed too drastically. Agents using the value functions produced in this manner are able to direct an MD simulation through a complete set of 4 MD steps using a median of 40 quanta after 250 epochs of training, as illustrated in Figure 1. These policies, which are reached after approximately 200 epochs of training, are reasonable in their choice of actions. When allocation of work is needed, the agent does so and, after a brief period during which the agent needlessly reallocates the work, the agent does nothing, thus allowing the MD work that has been allocated to be processed.

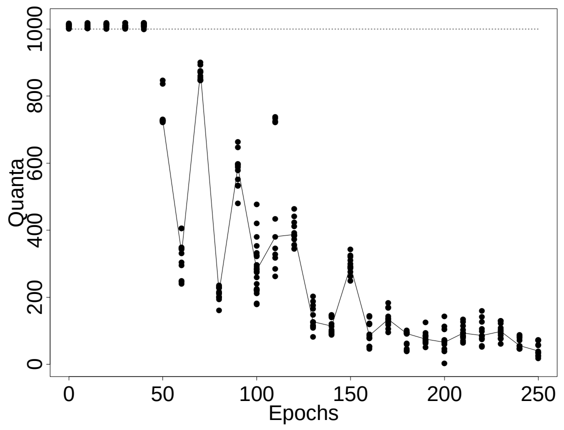

Figure 1: Median quanta to completion vs. training epochs (runs above dotted line are stuck).

Our future efforts will focus on exploring the usefulness of the squashing technique (Eq. 4) over a wider range of training runs and determining the empirical limits of linear-function approximation in this problem.

References

[1] J. A. Board, Z. S. Hakura, W. D. Elliott, D. C. Gray, W. J. Blanke, and Jr. J. F. Leathrum. Scalable implementations of multipole-accelerated algorithms for molecular dynamics. Technical Report 94-002, Duke University, Department of Electrical Engineering, 1994.

[2] Y. Hwang, R. Das, J. H. Saltz, M. Hodoscek, and B. R. Brooks. Parallelizing molecular dynamics programs for distributed-memory machines. *IEEE Computational Science and Engineering*, Summer:18–29, 1995.

[3] S. P. Singh, A. G. Barto, R. Grupen, and C. Connolly. Robust reinforcement learning in motion planning. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 2*, pages 655–662. Morgan Kaufmann, 1992.

[4] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, Massachusetts Institute of Technology, March 1996.