Decoupled Load Balancing

Olga Pearce[⋆†], Todd Gamblin[†], Bronis R. de Supinski[†], Martin Schulz[†], Nancy M. Amato[⋆]

[⋆]Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA
[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA
{olga,amato}@cse.tamu.edu, {olga,tgamblin,bronis,schulzm}@llnl.gov

Abstract

Modern scientific simulations divide work between parallel processors by decomposing a spatial domain of mesh cells, particles, or other elements. A balanced assignment of the computational load is critical for parallel performance. If the computation per element changes over the simulation time, simulations can use dynamic load balance algorithms to evenly redistribute work to processes. Graph partitioners are widely used and balance very effectively, but they do not strong scale well. Typical SPMD simulations wait while a load balance algorithm runs on all processors, so a poorly scaling algorithm can itself become a bottleneck.

We observe that the load balance algorithm is separate from the main application computation and has its own scaling properties. We propose to decouple the load balance algorithm from the application, and to offload the load balance computation so that it runs concurrently with the application on a smaller number of processors. We demonstrate the costs of decoupling and offloading the load balancing algorithm from a Barnes-Hut application.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; I.6.8 [*Simulation and Modeling*]: Types of Simulation—Parallel; C.4 [*Performance of Systems*]: Performance attributes, Modeling techniques

Keywords load balance; parallel algorithm; performance

1. Introduction

With increasing levels of concurrency in modern supercomputers, more and more processes may wait for the slowest process to reach a synchronization point. Load balancing, or assigning equal amounts of work to processes, is increasingly important for high performance scientific applications. Because simulation workloads evolve over time, many large-scale parallel applications use load balance algorithms to redistribute work evenly. Graph partitioners [2, 4] are frequently used to assign work evenly and to optimize communication locality. However, graph partitioners are computationally intensive and require sophisticated parallelization. Poor scaling limits their use at large scale.

We describe an approach to decouple and to offload the load balance algorithm to resources that the application does not use. In this MPMD configuration, partitioning executes concurrently with the application and with higher parallel efficiency than if it were run on the same processors as the simulation. Work is reassigned *lazily* as assignment directions become available, and the application does not have to wait for the load balance algorithm to complete. Our lazy load balancing approach allows an application to decouple load balance computation from the main application processing and, as a result, to offload it from the critical path on properly sized processor set for the load balance algorithm.

2. Decoupled Approach

SPMD simulations divide the simulated domain into logical elements, which are assigned to processors in the parallel machine. Often, the computational work per element varies over time, and an effective load balance algorithm attempts to ensure that the total work assigned to each processor is equal. Figure 1(a) shows the main components of the traditional approach to load balancing an application. The main steps are:

1. *Evaluate Imbalance*: Decide whether to correct load imbalance at this point in execution;
2. *Run Load Balance Algorithm*: Use a load balance method to compute directions on how to rebalance;
3. *Rebalance* the application if needed.

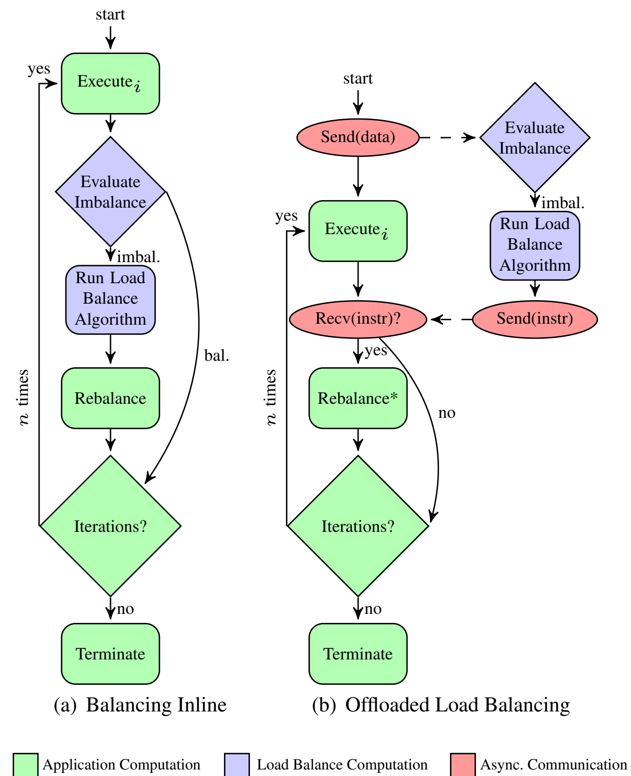
Figure 1(a) demonstrates how these steps are typically performed as a blocking phase of the application's primary computation, which is paused while load balance decisions are made. As discussed, this approach is not well suited to using a graph partitioner load balance algorithm, as these algorithms do not scale to the process counts that the application does. Potentially thousands of application processes may wait for the load balancing algorithm while it runs at sub-optimal efficiency. While the load balance algorithm can be parallelized, it will scale differently than the application, either causing an inefficient execution on all available processors or fewer active processors in the load balancing phase while the other (potentially thousands or more) application processes long sit idle.

The load balance algorithm is distinct from the application calculation so we *decouple* them by moving and merging the data to be partitioned onto a (often smaller) set of processes. To avoid pausing the application computation while computing a load balance assignment, we can *offload* the load balance algorithm computation to a separate balancing partition from those used by the application. Separation allows our load balance algorithm to run concurrently with the application, overlapping application computation and load balance algorithm computation. Assignments are applied by the application *lazily* as they become available.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the author/owner(s).

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688539



(a) Balancing Inline (b) Offloaded Load Balancing

Figure 1. Inline vs. Offloaded Load Balancing

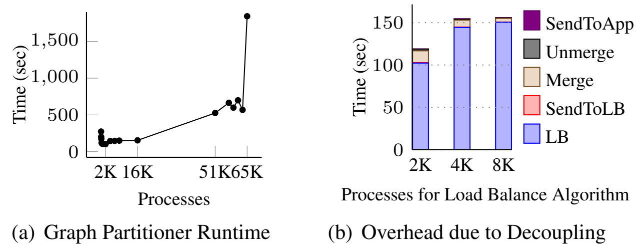
Figure 1(b) corresponds to the offloaded load balance algorithm execution. Transparently to the application, our framework reserves the resources for the load balance algorithm, and sends data from the application to the load balance processes. Our framework merges application data from the application processes to run on a smaller number of load balance processes, and runs the load balance algorithm more efficiently at a smaller scale than the application. The application continues running while the load balance algorithm computes corrections. When the load balance computation is completed, our framework sends the assignment instructions to the application, and the application rebalances.

A potential problem with lazy load balancing is that application state changes over time. Work per element may change, as may the number of elements. We call this change *drift*. However, we observe that the work distribution in most parallel SPMD applications changes slowly. In many cases, a balanced assignment computed from a past application state is a good approximation of a balanced assignment for the current state, so we *can* compute the assignment asynchronously and apply it *lazily* when the result is available.

2.1 Evaluation

For our experimental studies, we balance a Barnes-Hut [1] simulation [3] with 306.5M million interactions on 65,536 processes of an IBM Blue Gene/Q system, a tightly coupled massively parallel processing (MPP) system that contains PowerPC based compute nodes with 16 cores (64 hardware threads) each. Nodes are connected by five dimensional torus network and run a simplified compute node OS, the CNK. We use GCC 4.4.6 and IBM's MPI implementation.

Application Drift. Barnes-Hut is a gravitational force simulation where the number of particles remains the same throughout the simulation, but the interactions computed per particle can change as particles move because the simulation only computes gravitational



(a) Graph Partitioner Runtime (b) Overhead due to Decoupling

Figure 2. Load Balance Algorithm Overhead (265K Vertices)

interactions within a cutoff radius. We developed application drift metrics and empirically observe that the use of a drifted assignment results in an imbalance of less than 5%.

Load Balancing Overhead. Figure 2(a) shows strong scaling performance of a parallel graph partitioner running on 32 to 65,536 processors. Peak efficiency is achieved with 2,048 processors, and from this point on, runtime increases. On 65,536 processes, the algorithm spends all of its time in communication, and the runtime skyrockets. At scale, we could do better by running the graph partitioner on fewer processes and avoiding these excessive costs.

Figure 2(b) shows the costs of decoupling the load balance algorithm as a function of the resources provided to the load balance algorithm; we show the fastest cases only, with the load balance algorithm using 2k to 8k processes. These costs include sending the data to the load balancing processes, merging the data from several application processes on a single load balancing process, running the load balance algorithm in parallel on load balancing processes, unmerging, and sending the load balancing instructions back to corresponding application processes. The communication overhead imposed by the decoupling has a strong relationship with the ratio of application processes to load balancing processes, consistent with costs that one would expect.

Figure 2 demonstrates that decoupling the load balance algorithm from the application allows us to run it more efficiently without introducing significant overhead due to the extra communication. Decoupling the load balance algorithm and running it on $\frac{1}{32}$ of the available resources in this case results in 15x shorter runtime of the load balance computation.

3. Conclusions

We have presented an approach to load balancing based on decoupling the load balance algorithm from the application and offloading the load balance computation to overlap it with application execution. We implemented a framework that performs the decoupling and offloading of the load balance algorithm transparently to the application; analogously to the inline case, the application is required to provide information for the load balance mechanism. We show that decoupling the load balance computation can reduce the load balancing overhead by a factor of 15.

References

[1] J. Barnes and P. Hut. A Hierarchical O(N log N) Force-Calculation Algorithm. *Nature*, 324:446–449, December 1986.

[2] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, J. Teresco, J. Faik, J. Flaherty, and L. Gervasio. New Challanges in Dynamic Load Balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, 2005.

[3] O. Pearce, T. Gamblin, B. R. de Supinski, T. Arsenlis, and N. M. Amato. Load Balancing N-Body Simulations with Highly Non-Uniform Density. In *Intl. Conf. on Supercomputing (ICS)*, June 2014.

[4] K. Schloegel, G. Karypis, and V. Kumar. A Unified Algorithm for Load-Balancing Adaptive Scientific Simulations. In *SC'00*, Nov. 2000.