TITLE

Distributed Memory Load Balancing

BYLINE

Aaron Becker, Gengbin Zheng, and Laxmikant Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
abecker3@illinois.edu, gzheng@illinois.edu, kale@illinois.edu

SYNONYMS

DEFINITION

Load balancing in distributed memory systems is the process of redistributing work between hardware resources to improve performance, typically by moving work from overloaded resources to underloaded resources.

DISCUSSION

In a parallel application, when work is distributed unevenly so that underloaded processors are forced to wait for overloaded processors, the application is suffering from *load imbalance*. Load imbalance is one of the key impediments in achieving high performance on large parallel machines, especially when solving highly dynamic and irregular problems. *Load balancing* is a technique that performs the task of distributing computation and communication load across the hardware resources of a parallel machine so that no single processor is overloaded. It can reduce processor idle time across the machine while also reducing communication costs by co-locating related work on the same processor.

Balancing an application's load involves making decisions about where to place newly created computational tasks on processors, or where to migrate existing work among processors. Orthogonally, some applications only require static load balancing: they have consistent behaviors over their lifetimes, and once they are balanced they remain balanced. Other applications exhibit dynamic changes in behavior and require periodic re-balancing. Typically it is far too expensive to compute an optimal distribution of work in any realistic application, but a wide variety of heuristic algorithms have been developed that are very effective in practice across a wide variety of application domains.

The load associated with an application can be conceptualized as a graph, where the nodes are discrete units of work and an edge exists between two nodes if there is a communication between them. To solve the load balancing problem one must split this graph into parts, with each

part representing the work to be associated with a particular hardware resource. This partitioning process is at the heart of load balancing.

Given the heuristic nature of load balancing, which makes perfect load balance an unrealistic goal, it is important to remember that the goal of load balancing is not so much to equalize the amount of work on each processor as it is to minimize the load of the most heavily loaded processor in the system. The most overloaded processor is the bottleneck that determines time to completion. A heuristic that leaves some processors idle is preferable to one that reduces the variance in load, as long it reduces the load on the most loaded processor. Load balancing heuristics can also take factors beyond time to completion into account, for example by trying to minimize power usage over an application's lifetime.

Modern HPC systems include clusters of multi-core nodes. The general load balancing strategies described in this article are still applicable at the level of nodes (i.e. work is assigned to nodes). Finer grained load balancing within a node (work assigned to cores) can be performed independently. In the simplest case, it can be done by applying the same load balancing strategies that are used for node-level load balancing, except that the cost of communication among cores within a node is much smaller than the inter-node communication, and the number of cores involved within a node is relatively small.

Periodic Load Balancing

In a periodic load balancing scheme, computational tasks are persistent; load is balanced only as needed, and the balancing consists of migrating existing tasks and their associated data. With periodic load balancing, expensive load balancing decision making and data migration are not continuous processes. They occur only distinct point in the application when a decision to do balancing has been made. Periodic load balancing schemes are suitable for iterative scientific applications such as molecular dynamics, adaptive finite element simulation, and climate simulation, where the computation typically consists of a large number of time steps, iterations (as in iterative linear system solvers), or a combination of both. A computational task in these applications is executed for a long period of time, and tends to be persistent. During execution, at periodic intervals, partially executed tasks may be moved to different processors to achieve global load balance.

The Load Balancing Process

In an application which rebalances its load periodically, there are four distinct steps in the load balancing process. The first step is load estimation, which tries to gauge what the load on each processor will be in the near future if no rebalancing is done. This may involve the use of a model that predicts future performance based on current conditions, or it may be based directly on past measured load, with the assumption that the near future will closely resemble the near past. The second step is making a decision of *when* to execute load balancing. This is a trade-off between the cost of load balancing itself (that is, the cost of determining a new mapping of application tasks to hardware and the cost of migration) and the savings one can expect from a better load distribution. The nature of the application being balanced and the cost of the load balancing method to be used factor heavily into this decision. The third step is determining how the load will be rebalanced. There are many algorithms devoted to computing a good mapping of work onto processors, each with its own strengths and weaknesses. This section describes some of the most widely used

methods. The general problem of creating a mapping of work onto processors which minimizes time to completion is NP-hard, so all strategies we discuss are heuristic. The final step in the load balancing process is migration, the process by which work is actually moved. This may simply be a matter of relocating application data, but it can also encompass thread and process migration.

Initial Balancing

For some classes of application, there is very little change in load balance over time once the application reaches a steady state. Thus, once good load balance is achieved, no further balancing need be done. However, load balancing is often still an important part of such applications. For example, consider the case of molecular dynamics, where one must distribute simulated particles onto processors. These applications do not experience significant dynamic load imbalance, but determining a good initial mapping may be difficult because of the difficulty of estimating the load of multiple computational tasks accurately a priori.

In cases like this, one can simply start the application with an unoptimized distribution of work, run the application long enough to accurately measure the load, rebalance based on those measurements, and then continue without the need for any further balancing.

Classifying Load Balancers

There are many families of load balancing approaches that can be classified according to how they answer the fundamental questions of load balancing: how do we estimate the load, at what granularity should one balance the load, and where should the load balancing decisions be made.

Load *estimation* underlies all load balancing algorithms. Load balancing is fundamentally a forward-looking task which aims to improve the future performance of the application. To do this effectively, the load balancer must have some model of what the future performance of the application will be in different scenarios. There are two common approaches to estimating future load. The first is to measure the current load associated with each piece of work in the application and to assume that this load will remain the same after load balancing. This assumption is based on the *principle of persistence*, which posits that, for certain classes of scientific and engineering applications, computational loads and communication patterns tend to persist over time, even in dynamically evolving computations. This approach has several advantages: it can be applied to any application, it accurately and automatically accounts for the particular characteristics of the machine on which the application is running, and it removes some of the burden of load estimation from the application developer.

The alternative is to build some model of application performance which will provide a performance estimate for any given distribution of work. These models can be sophisticated enough to take into account dynamic changes in application performance that a measurement-based scheme will not account for, but they must be created specifically to match the actual characteristics of the application they are used for. If the model does not match reality then poor load balancing may result.

Load balancing may take place at any of several levels of *granularity*. The greatest flexibility in mapping work onto hardware resources can be achieved by exposing the smallest possible meaningful units of data to the load balancer. For example, these may be nodes in a finite element application or individual particles in a molecular simulation. Alternatively, one may group this

data into larger chunks and only expose those chunks to the load balancing algorithm. This makes the load balancing problem smaller while guaranteeing respect for locality within the chunks. For example, in an molecular simulation the load balancer might only try to balance contiguous regions which may contain a large number of particles without being directly aware of the particles. It is also possible to balance load by migrating entire processes, avoiding the need for application developers to write code dedicated to migrating their data during the load balancing process.

Load balancers may be further categorized according to *where* decisions are made: locally, globally, or according to some hierarchical scheme. Global load balancers collect global information about load across the entire machine and can use this information to make decisions that take the entire state of the application into account. The advantage of these schemes is that load balancing decisions can take a global view of the application, and all decisions about which objects should migrate to which processors can be made without further coordination during the load balancing process. However, these schemes inherently lack scalability. As problem sizes increase, the object communication graph may not even fit in memory on a single node, making global algorithms infeasible. Even if the load balancing process is not constrained by memory, the time required to compute an assignment for very large problems may preclude the use of a global strategy. However, with coarse granularity, it is possible to use global strategies up to several thousand processors, especially if load balancing is relatively infrequent.

Parallelized versions of global load balancing schemes are also possible. These use the same principles for partitioning as a serial global scheme, but to improve scalability the task graph is spread across many processors. This introduces some overhead in the partitioning process, but allows much larger problems to be solved than a purely serial scheme. One example of a parallel global solver is ParMETIS, the parallel version of the METIS graph partitioner.

Distributed load balancing schemes lie at the opposite end of the scale from global schemes. Distributed schemes use purely local information in their decision making process, typically looking to offload work from overloaded processors to their less-loaded immediate neighbors in some virtualized topology. This leads to a sort of diffusion of work through the system as objects gradually move away from overloaded regions into underloaded regions. These schemes are very cheap computationally, and do not require global synchronization. However, they have the disadvantage of redistributing work slowly compared to global schemes, often requiring many load balancing phases before acheiving system-wide load balance, and they typically perform more migrations before load balance is achieved.

Hybrid load balancing schemes represent a compromise between global and distributed schemes. In a hybrid load balancer, the problem domain is split into a hierarchy of sub-domains. At the bottom of the hierarchy, global load balancing algorithms are applied to the small sub-domains, redistributing load within each sub-domain. At higher levels of the hierarchy, the load balancing strategies operate on the sub-domains as indivisible units, redistributing these larger units of work across the machine. This keeps the size of the partitioning problem to be solved low without giving up some global decision-making capabilities.

Algorithms

An application's load balancing needs can vary widely depending on its computational characteristics. Accordingly, a wide variety of load balancing algorithms have been developed. Some do careful analysis of an application's communication patterns in an effort to reduce inter-node com-

munication; others only try to minimize the maximum load on a node. Some try to do a reasonably good job as quickly as possible; others take longer to provide a higher quality solution. Some are even tailored to meet the needs of particular application domains. Despite the wide variety of algorithms available, there are a few in wide use that demonstrate the range of available techniques.

Parallel Prefix

In some cases, the pattern of communication between objects is unimportant, and we care only about keeping an equal amount of work on each processor. In this scenario, load may be effectively distributed using a simple parallel prefix strategy (also known as scan or prefix sum).

The input to the parallel prefix algorithm is simply a local array of work unit weights on each processor reflecting the amount of work that each unit represents. The classic parallel prefix algorithm computes the sum of the first i units for each of the n unit in the list. Once the prefix sum is computed, all work can be equally distributed across processors by sending each unit to the processor number given by its prefix sum value divided by the total work divided by the number of processors.

The primary advantage of the parallel prefix scheme is its fast, inexpensive nature. The computation can be shared across p processors using only $\log p$ communications, and efficient prefix algorithms are widely available in the form of library functions like `MPI_Scan`. An assignment of work units to processors based on parallel prefix can be computed virtually instantaneously even for extremely large processor counts, regardless of variations in work unit weights.

The corresponding disadvantage of using the parallel prefix algorithm for load balancing is its simplicity. It does not account for the structure of communication between processors and will not attempt to minimize the communication volume of the resulting partition. It also does not attempt to minimize the amount of migration needed.

Recursive Bisection

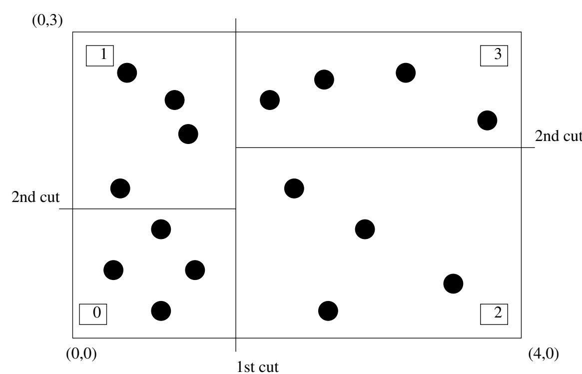

Figure 1: Orthogonal recursive bisection partitions by finding a cutting plane that splits the work into two approximately equal pieces and recursing until the resulting pieces are small enough.

Recursive bisection is a divide-and-conquer technique that reduces the partitioning problem to a series of bisection operations. The strategy of recursive bisection is to split the object graph in

two approximately equal parts while minimizing communication between the parts, then proceeding to subdivide each half recursively until the required number of partitions is achieved. This recursive process introduces parallelism into the partitioning process itself. After the top level split is completed, the two child splits are independent and can be computed in parallel, and after n levels of bisection there are 2^n independent splitting operations to perform. In addition, geometric bisection operations can themselves be parallelized by constructing a histogram of the nodes to be split.

There are many variations of the recursive bisection algorithm based on the algorithm used to split the graph. Orthogonal recursive bisection (ORB) is a geometric approach in which each object is associated with coordinates in some spatial domain. The bisection process splits the domain in two using an axis-aligned cutting plane. This can be a fast way of creating a good bisection if most communication between objects is local. However, this method does not guarantee that it will produce geometrically connected subdomains, and it may also produce subdomains with high aspect ratios. Further variations exist which allow cutting planes that are not axis-aligned.

An alternate approach is to use spectral methods to do the bisection. This involves constructing a sparse matrix from the communication graph, finding an eigenvector of that matrix, and using it as a separator field to do the bisection. This method can produce superior results compared with ORB, but at a substantially higher computational cost.

For many problems, recursive bisection is a fast way of computing good partitions, with the added benefit that the recursive procedure naturally exposes parallelism in the partitioning process itself. The advantages and disadvantages of recursive bisection depend greatly on the nature of the bisection algorithm used, which can greatly affect partition quality and computational cost.

Space-Filling Curve

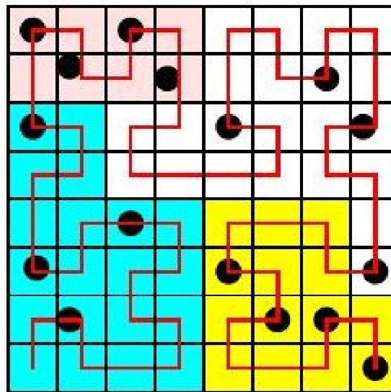

Figure 2: To partition data using a space-filling curve, a very coarse approximation of the curve is replaced with finer and finer approximations until each unit of work is in its own segment. The curve is then split into pieces, with each piece containing an equal number of work units. This figure shows a Hilbert curve.

A space-filling curve is a mathematical function that maps a line onto the entire unit square (in 2 dimensions, or the entire unit N-cube in N dimensions). There are many such curves, and

they are typically defined as the limit of sequence of simple curves, so that closer and closer approximations to the space filling curve can be constructed using an iterative process. The value of space-filling curves for load balancing is that they can be used to convert n-dimensional spatial data into one-dimensional data. Once the higher-dimensional data has been linearized, it can be easily partitioned by splitting the line into pieces with equal numbers of elements.

Consider the case where each object in the application has a two-dimensional coordinate associated with it. Starting with a coarse approximation to the space-filling curve, a recursive refinement process can be used to create closer and closer approximations until each object is associated with its own segment of the approximated curve. This creates a linear ordering of the objects, which can then be split into even partitions.

A good choice of space-filling curve will tend to keep objects which are close together in the higher-dimensional space. This property is necessary so that the partitions that result respect the locality of the data. Many different curves have been used for load balancing purposes, including the Peano curve, the Morton curve, and the Hilbert curve. The Hilbert curve is a common choice of space-filling curve for partitioning applications because it provides good locality.

Graph Partitioning

The pattern of communication in any parallel program can be represented as a graph, with nodes representing discrete units of work, and weighted edges representing communication. Graph partitioning is a general purpose technique for splitting such a graph into pieces, typically with the goal of creating one piece for each processor. There are two conflicting goals in graph partitioning: achieving equal-size partitions, and minimizing the total amount of communication across partition boundaries, known as the edge cut. Finding an optimal solution to the graph partitioning problem is NP-Hard, so heuristic algorithms are required.

Because it is computationally infeasible to attempt to partition the whole input graph at once, the graph partitioning algorithm proceeds by constructing a series of coarser representations of the input graph by combining distinct nodes in the input graph into a single node in the coarser graph. By retaining information about which nodes in the original graph have been combined into each node of the coarse graph, the algorithm maintains basic information about the graph's structure while reducing its size enough that computing a partitioning is very simple.

Once the coarsest graph has been partitioned, the coarsening process is reversed, breaking combined nodes apart. At each step of this uncoarsening process, a refinement algorithm such as the Kernighan-Lin algorithm can be used to improve the quality of the partition by finding advantageous swaps of nodes across partition boundaries. Once the uncoarsening process is complete, the user is left with a partitioning for the original input mesh.

This technique can be further generalized to operate on hypergraphs. Whereas in a graph an edge connects two nodes, in a hypergraph a hyperedge can connect any number of nodes. This allows for the explicit representation of relationships that link several nodes together as one edge rather than the standard graph representation of pairwise edges between each of the nodes. For example, in VLSI simulations hypergraphs can be used to more accurately reflect circuit structure, leading to higher quality partitions. Hypergraphs can be partitioned using a variation of the basic graph partitioning algorithm which coarsens the initial hypergraph until it can be easily partitioned, then refining the partitioned coarse hypergraph to obtain a partitioning for the full hypergraph.

The primary advantages of load balancing schemes based on graph partitioning are their flexibility and generality. An object communication graph can be constructed for any problem without depending on any particular features of the problem domain. The resulting partition can be made to account for varying amounts of work per object by weighting the nodes of the graph and to account for varying amounts of communication between objects by weighting the edges. This flexibility makes graph partitioners a powerful tool for load balancing.

The primary disadvantage of these schemes is their cost. Compared to the other methods discussed here, graph partitioning may be computationally intensive, particularly when a very large number of partitions is needed. The construction of an explicit communication graph may also be unnecessary for problems where communication may be inferred from domain-specific information such as geometric data associated with each object.

Rebalancing with Refinement vs. Total Reassignment

There are two ways to approach the problem of rebalancing the computational load of an application. The first approach is to take an existing data distribution as a baseline and attempt to incrementally improve load balance by migrating small amounts of data between partitions while leaving the majority of data in place. The second approach is to perform a total repartitioning, without taking the existing data distribution into account. Incremental load balancing approaches are desirable when the application is already nearly balanced, because they impose smaller costs in terms of data motion and communication. Incremental approaches are more amenable to asynchronous implementations that operate using information from only a small set of processors.

Software Frameworks

Many parallel dynamic load balancing libraries and frameworks have been developed before for specialized domains. These libraries are often particularly useful because they allow application developers to specify the structure of their application once and then easily try a number of different load balancing strategies to see which is most effective in practice without needing to reformulate the load measurement and task graph construction process for each new algorithm.

The Zoltan toolkit [1] provides a suite of dynamic load balancing and parallel repartitioning algorithms, including geometric, hypergraph, and graph methods. It provides a simple interface to switch between algorithms, allowing straightforward comparisons of algorithms in applications. The application developers provide an explicit cost function and communication graph for the Zoltan algorithms to use.

Charm++ [2] adopts a migratable object-based load balancing model. As such, Charm programs have a natural grain size determined by their objects, and can be load balanced by redistributing their objects among processors. Charm provides facilities for automatically measuring load, and provides a variety of associated measurement-based load balancing strategies that use the recent past as a guideline for the near future. This avoids the need for developers to explicitly specify any cost functions or information about the application's communication structure. Charm++ includes a suite of global, distributed, and hierarchical load balancing schemes.

DRAMA [3] is a library for parallel dynamic load balancing of finite element applications. The application must provide the current distributed mesh, including information about its computation and communication requirements. DRAMA then provides it with all necessary information

to re-allocate the application data. The library computes a new partitioning, either via direct mesh migration or via parallel graph re-partitioning, by interfacing to the ParMetis or Jostle graph partitioning libraries. This project is no longer under active development.

The Chombo [4] package was developed by Lawrence Berkeley National Lab. It provides a set of tools including load balancing for implementing finite difference methods for the solution of partial differential equations on block-structured adaptively refined rectangular grids. It requires users to provide input indicating the computational workload for each box, or mesh partition.

Task Scheduling Methods

Some applications are characterized by the continuous production of tasks rather than by iterative computations on a collection of work units. These tasks, which are continually being created and completed as the application runs, form the basic unit of work for load balancing. For these applications, load balancing is essentially a task scheduling or task allocation problem. This task pool abstraction captures the execution style of many applications such as master/worker and state-space search computations. Such applications are typically non-iterative.

Load balancing strategies in this category can be classified as centralized, fully distributed, or hierarchical. Hierarchical strategies are hybrids that aim to combine the benefits of centralized and distributed methods. In centralized strategies, a dedicated "central" processor gathers global information about the state of the entire machine and uses it to make global load balancing decisions. On the other hand, in a fully distributed strategy, each processor exchanges state information only with other processors in its neighborhood.

Fully distributed load balancing received significant attention from the early days in parallel computing. One way to categorize them is based on which processor initiates movement of tasks.

Sender-initiated schemes assign newly created tasks to some processor, chosen randomly or from one of the neighbors in a physical or virtual topology. The decision to assign it to another processor, instead of retaining the task locally may also be taken randomly, or based on a load metric such as a queue size. Random assignment has some good statistical properties but suffers from high communication costs, by requiring that most tasks be sent to remote processors. With neighborhood strategies, global balancing is achieved as tasks are moved from heavily loaded neighborhood diffuse onto lightly loaded processors. In the adaptive contraction within neighborhood (ACWN) scheme, tasks always travel to topologically adjacent neighbors with the least load, but only if the difference in loads is more than a predefined threshold. In addition, ACWN does saturation control by classifying the system as being either lightly, moderately or heavily loaded.

In receiver-initiated schemes, the underloaded processors request load from heavily loaded processors. You may chose the victim to request work from randomly, or via a round-robin policy, or from among "neighboring" processors. Randomized work stealing is yet another distributed dynamic load balancing technique, which is used in some runtime systems such as Cilk.

The distinction between sender or receiver initiation is blurred when processors exchange load information, typically with neighbors in a virtual topology. Although the decision to send work is taken by the sender, it is taken in response to load information from the receiver. For example, in neighborhood averaging schemes, after periodically exchanging load information with neighbors in a virtual (and typically, low-diameter) topology, each processor that is overloaded compared with its neighbors, sends equalizing work to its lower-loaded neighbors. Such policies tend to be proactive compared with work stealing, trading better load balance for extra communication.

Another strategy in this category, and one of the oldest ones, is the gradient model. Here each processor participates in a continuous fixed-point computation with its neighbors to identify the neighbor that is closest to an idle processor. Overloaded processors then send work towards the idle processors via that neighbor.

The gradient model is a demand-driven approach. In gradient schemes, underloaded processors inform other processors of their state, and overloaded processors respond by sending a portion of their load to the nearest lightly loaded processor. The resulting effect is a form of a gradient map that guides the migration of tasks from overloaded to underloaded processors.

The dimensional exchange method is a distributed strategy in which balancing is performed in an iterative fashion by "folding" an P processor system into $\log P$ dimensions and balancing one dimension at a time. I.e. in phase i, each processor exchanges load with its neighbor in the ith dimension so as to equalize load among the two. After $\log P$ phases, the load is balanced across all the processors. This scheme is conceptually designed for a hypercube system but may be applied to other topologies with some modification.

Several hierarchical schemes have been proposed that avoid the bottleneck of a centralized strategies, while retaining some of their ability to achieve global balance quickly. Typically, processors are organized in a two level hierarchy. Managers at the lower level behave like masters in centralized strategies for their domain of processors, and interact with other managers as processors in a distributed strategies. Alternatively, load balancing is initiated at the lowest levels in the hierarchy, and global balancing is achieved by ascending the tree and balancing the load between adjacent domains at each level in the hierarchy.

Often, priorities are associated with tasks, especially for applications such as branch-and-bound or searching for one solution in a state-space search. Task balancing for these scenarios is complicated because of the need to balance load while ensuring that high priority work does not get delayed by low priority work. Sender-initiated random assignment as well as some hierarchical strategies have shown good performance in this context.

RELATED ENTRIES

CHACO
Graph Partitioning
Hypergraph Partitioning
Parallel Prefix
Partitioning
SFC - Space-filling Curves
Topology Aware Task Mapping
Task Graph Scheduling

BIBLIOGRAPHIC NOTES AND FURTHER READING

There is a rich history of load balancing literature that spans decades. This article is only able to cite a very small amount of this literature, and attempts to include modern papers with broad scope and great impact. Kumar et al. [5] describe the scalability and performance characteristics

of several task scheduling schemes, and Devine et al. [6] gives a good overview of the range of techniques used for periodic balancing in modern scientific applications and the load balancing challenges faced by these applications. Xu and Lau [7] give an in-depth treatment of distributed load balancing articles, covering both mathematical theory and actual implementations.

For practical information on integrating load balancing frameworks into real applications, literature describing principles and practical use of such systems as DRAMA [3], Charm++ [2, 8], Chombo [4], and Zoltan [1] is a crucial resource.

More information is available on the implementation of task scheduling methods, whether they are centralized [9, 10], distributed [11, 12, 13], or hierarchical [13]. Work stealing is described in detail by Kumar et al. [5], while associated work on Cilk is described in Frigo et al. [14]. Dinan et al. [15] extended work stealing to run on thousands of processors using ARMCI.

BIBLIOGRAPHY

[1] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. In *Proc. Intl. Conf. Supercomputing*, May 2000.

[2] Laxmikant V. Kale and Gengbin Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[3] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J.-M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw. Dynamic load balancing of finite element applications with the DRAMA Library. In *Applied Math. Modeling*, volume 25, pages 83–98, 2000.

[4] Chombo Software Package for AMR Applications. http://seesar.lbl.gov/anag/chombo/.

[5] V. Kumar, A. Y. Grama, and Nageshwara Rao Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[6] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challenges in dynamic load balancing. *Appl. Numer. Math.*, 52(2–3):133–152, 2005.

[7] Chengzhong Xu and Francis C. M. Lau. *Load Balancing In Parallel Computers Theory and Practice*. Kluwer Academic Publishers, 1997.

[8] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *Proceedings of the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, San Diego, California, USA, September 2010.

[9] Yaun-Chien Chow and Walter H. Kohler. Models for dynamic load balancing in homogeneous multiple processor systems. In *IEEE Transactions on Computers*, volume c-36, pages 667–679, May 1982.

[10] L. M. Ni and Kai Hwang. Optimal load balancing in a multiple processor system with many job classes. In *IEEE Trans. on Software Eng.*, volume SE-11, 1985.

[11] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load balancing policies for dynamic applications. In *IEEE Concurrency*, pages 7(1):22–31, 1999.

[12] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, September 1993.

[13] Amitabh Sinha and L.V. Kalé. A load balancing strategy for prioritized execution of tasks. In *International Parallel Processing Symposium*, pages 230–237, New Port Beach, CA., April 1993.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, volume 33 of *ACM Sigplan Notices*, pages 212–223, Montreal, Quebec, Canada, June 1998.

[15] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.