Scalable Load Distribution and Load Balancing for Dynamic Parallel Programs

E. Berger and J. C. Browne
Department of Computer Science
University of Texas at Austin
Austin, Texas 78701 USA
01-512-471-{9734,9579}

{emery,browne}@cs.utexas.edu

ABSTRACT

This paper reports design and preliminary evaluation of an integrated load distribution-load balancing algorithm which was targeted to be both efficient and scalable for dynamically structured computations. The computation is represented as a dynamic hierarchical dependence graph. Each node of the graph may be a subgraph or a computation and the number of instances of each node is dynamically determined at runtime. The algorithm combines an initial partitioning of the graph with application of randomized work stealing on the basis of subgraphs to refine imbalances in the initial partitioning and balance the additional computational work generated by runtime instantiation of subgraphs and nodes. Dynamic computations are modeled by an artificial program (k-nary) where the amount of parallelism is parameterized. Experiments on IBM SP2s suggest that the load balancing algorithm is efficient and scalable for parallelism up to 10,000 parallel threads for closely coupled distributed memory architectures.

Keywords scalable, load distribution, load balancing, work stealing

1. INTRODUCTION

The motivation for this research is the emergence of dynamically structured computations executing on dynamic parallel and distributed resource sets. Maintenance of a "good" balance of work across processing elements is essential for such computational systems.

The parallel programs most in need of effective load distribution and load balancing (LD/LB) are those which are both large and dynamically structured. LD/LB is often rendered even more difficult by the constraints on moving of units of computation created by the large volumes of data often associated with the units of computation. Scalability of LD/LB algorithms have received relatively little attention.

Little attention has been paid to combining load distribution and load balancing for parallel programs which dynamically create and delete units of computation during execution.

This paper proposes a load distribution/load balancing (LD/LB) algorithm which combines an initial static graph partitioning with dynamic thread level workload distribution by the "work stealing" algorithm [3,4]. Work stealing is attractive as a dynamic scheduling algorithm since it can be shown to be optimal under plausible circumstances and because it admits derivation of bounding values for important problem parameters. This study, while it is ultimately experimental, is based on theoretical models of the variants of work stealing which we use.

We describe several methods of load distribution based on simple (non-optimal) but plausible graph partitioning and analyze the effect of combining these methods with work-stealing. We show that integration of work-stealing with load distribution improves performance, and that by combining work-stealing with a load distribution algorithm which utilizes information on data locality in the graph, we get the best performance improvement. It is further shown that the algorithm scales according to the definition of scalability given following.

Load Distribution – A load distribution is an assignment of work to a set of processing elements.

Load Balancing – Load balancing is the process of transferring units of work among processing elements during execution to maintain balance across processing elements.

Scalability - A scalable LD/LB Algorithm must be both efficient in execution and must attainment load balance for high degrees of parallelism and large numbers of processing units. Let n be the number of units of computation which are to be executed during the execution of the total computation. Let p be the number of processors over which the units of computation are to be distributed. Scalability efficient implies that the total time taken to execute the LD/LB method should be negligible for small values of n and should grow no faster than O(p) for large values of n and p. Scalability effective implies that the degree of load balance attained should be independent of n and p.

2. ALGORITHM DEFINITION

2.1 Overview

The most commonly used load distribution algorithms are based on graph partitioning algorithms [references]. We adopt the representation of computations as directed acyclic graphs (*dag*s) where arcs represent dependencies (serial and parallel) and nodes represent computations.

Let *w* be the total amount of computation, or work in the dag. Let d be the *dept*h of the *dag* (the critical path of the computation). Then *w* corresponds to the runtime for a serial execution of the program, while *d* is the runtime on an infinite-processor machine.

Let *s* (=speedup) be the runtime for a serial execution divided by the runtime for a parallel execution. Then the maximum *s* for *p* processors *maxs(p) = w/(w/p + d)* (It's not possible to do better than to do all of the work in parallel and then add the critical path). Let *degree of parallelism* be the maximum exploitable parallelism in a given computation. This corresponds to the maximum possible speedup on an infinite number of processors (*maxs*()) = *w/d*.

The central result of [3,4] states that for any dag, if *work-stealing* is used to schedule computations, the expected runtime on *p* processors T_p = w/p + {O}(d). However, this result ignores the problem of *load distribution*, that is, the mapping of processes to processors. While the Cilk model of [4] allows computations to be executed on any processor, for many high-performance codes, processes must be more or less permanently assigned to processors because they rely on large amounts of data that would be expensive to move.

We describe several methods of load distribution based on simple (non-optimal) but plausible graph partitioning and analyze the effect of combining these methods with work-stealing. We show that integration of work-stealing with load distribution improves performance, and that by combining work-stealing with a load distribution algorithm which utilizes information on data locality in the graph, we get the best performance improvement. It is further suggested that the algorithm scales according to the definition of scalability given preceding.

2.2 Program Representation

Effective load distribution algorithms for the class of computations of concern here must include consideration of data access patterns. The representation of the program which is used as the basis for partitioning is a dynamic, hierarchical structured dependence or data flow graph. The program representation we have chosen is the CODE [2,7] graph-oriented parallel programming language. A program can be constructed via a graphical user interface by placing *nodes* representing computations (typically calls to C or C++ functions) or subgraphs (a basic node is a degenerate case of a subgraph) and connecting them with *arcs* which represent both data and control flow. A CODE program is referred to as a *graph*. A *graph* may contain any number of *node*s, each corresponding to a *subgraph* or an alias to a *subgraph* (thereby permitting recursion). CODE nodes contain *firing rules* that determine when the node is ready to be executed, the computation to be executed, and *routing rules* that determine where data should be sent after the computation. They may also contain static (i.e., persistent) variables and automatic variables. CODE graphs are *dynamic*}: new nodes and *subgraphs* may be created at runtime (each instantiation of a new object has one or more *indices* associated with it).

The CODE model of computation is general and architecture-neutral. Programs written in CODE may be compiled for execution on shared-memory multiprocessors (SMP's), distributed-memory (DMPs) or cluster architecture multiprocessors. The problem is providing an efficient and scalable mechanism for executing CODE programs on DMP's and on clusters.

2.3 Load Distribution Algorithms

Because CODE graphs are dynamic, instantiations of graphs and computation nodes occur at runtime and it is not known at compile-time how many objects will be created: placement of these objects must be managed at runtime.

2.3.1 Load Distribution at the Node Level

Vokkarne [11] describes an initial distributed version of CODE which maps objects to processors after the manner of the early versions of the widely used PVM [6] distributed programming environment. We use this mode of initial distribution as a basis for discussion of the issues in partitioning graphs of dynamic programs where the nodes may have substantial data associated with them. Each object has a path, defined as the list of graphs (with their indices) which contain the object plus the object name and its indices. The path sum is defined as the sum of the indices along the path plus the unique identifiers of every object. The object (i.e., execution and storage associated with the object) is placed on processor number (path sum/mod(N)), where N is the total number of processors and processors are indexed {0 …N-1}. The algorithm we used is different in a small but important way. The "path sum" is defined as the product of the indices. This is important for scalability since mod(N) of the original path sum will generate substantial variations in workload for large numbers of processors.

This algorithm which we subsequently refer to as *node-level* mapping has possibly serious flaws for graphs with substantial communication among nodes:
(i) It destroys locality. Objects are dispersed across all processors, and no attempt is made to ensure that objects which are ``close together" (e.g., in the same graph) are kept together. Opportunities for avoiding unnecessary communication are not exploited and in fact are almost inevitably discarded. This is only

acceptable if communication is *very* small compared to computation.
(ii) It makes load balancing difficult. Since objects have a fixed processor assignment, it is not possible to balance the computational load among processors by moving computations.
(iii) Additionally, in the original formulation, the same nodes in recursive graphs would end up mapped to the same processor. This problem can be readily fixed. Since this failure can occur whenever the unique ID and the number of processors are not relatively prime, it can be solved by requiring unique ID's to be prime numbers larger than the pre-defined maximum number of processors. In the experiments, this modification has been made to the node-level mapping algorithm.

2.3.2 Load Distribution at the Graph Level

To avoid unnecessary communication, we also distinguish two types of computation nodes: *static* and *stateless* nodes. A static node has state that must be persistent. A node is static iff it has any static variables declared or if any firing rule depends on two or more inputs simultaneously (therefore, it must have a home processor). Stateless nodes may be executed on any processor -- they are analogous to pure function calls.

To avoid destroying locality, we change the mapping algorithm so that only *subgraphs* are distributed. All nodes inside a subgraph will be mapped onto the same processor; the subgraphs (with their nodes) will be distributed randomly across all processors using the path sum algorithm described above (this is equivalent to computing the path sum by omitting the object's name and indices when the object is not a subgraph). This has the important effect of limiting communication to inter-graph arcs.

The major drawback to the graph-level mapping algorithm described above is that if there is a high degree of parallelism within a single graph, this parallelism will not be exploited. Intuitively, it seems that a load balancing scheme would help.

2.4 Work Stealing Model

To address load imbalance, we augment the placement algorithm with work stealing. Work-stealing, as mentioned above, is a provably optimal load-balancing mechanism [3] It functions as follows: when a processor is idle, it sends out a request for work to another processor chosen at random. If this processor is busy, it rejects the request and the idle processor must try again. However, if the request is accepted, the idle processor becomes a *thief* and the busy processor a *victim*. Work is stolen from the busy processor's work queue and is executed by the thief. The thief then returns the results of the execution to the victim.

The work-stealing algorithm is slightly altered for CODE. Every processor gets two work queues: a *heavy* queue and a *light* queue. The heavy queue contains static nodes while the light queue contains stateless nodes. Local execution prefers the heavy queue, while theft prefers the light queue. This is done to reduce communication costs: while work-stealing a stateless node just requires a unique identifier for the node and its inputs, work-stealing a static node also requires two-way communication of the node's state.

Further, CODE attempts to first do work-stealing locally (among other processors on an SMP). If this fails and no remote work-stealing requests are outstanding, a remote work-steal request is sent to a random machine on the network. This machine checks only one of its processors' queues for work, and transmits work if some is found or a work-steal denial message otherwise.

3. Experimental Evaluation

We first define the experiments and then give a brief summary of the results from one experiment.

3.1 Experiment Definition

To evaluate the scalability, efficiency, and architecture-neutrality of this system, we wrote a CODE version of the *knary* benchmark [4]. This benchmark allows the creation of graphs with a wide range of degrees of parallelism so we can verify scalability with respect to the degree of parallelism. The knary(h,d,s) synthetic benchmark grows a tree of height h and degree d in which for each non-leaf node, the first s children are generated serially and the remaining children are generated in parallel. When it generates a node, the program first executes a fixed number of iterations of ''work" before generating the children.

Parameters	Degree of Parallelism	Maximum Speedup for 64 processors
(4,6,0)	64.75	32.18
(3,14,0)	70.33	33.3
(3.16,0)	91	37.57
(3,18,0)	114.33	41.03
(3,20,0)	140.33	43.95
(4,8,0)	146.25	44.52
(4.10,0)	277.75	52.01
(5,6,0)	311	53.07

Table 1 - Parameterization of knary workload

One of the advantages of knary is that it is possible to analytically solve for the work and critical path, as well as to place a loose upper bound on achievable speedup on a given number of processors.
$w(h,d,s) = (d^h - 1) / (d - 1)$,
$d(h,d,s) = (s^h - 1) / (s - 1)$.
By choosing d and s appropriately, we can vary the degree of parallelism and thereby test our algorithms to ensure that as parallelism grows, speedup approaches the maximum possible speedup. We graph speedup (w/T_p) versus degree of parallelism (w(h,d,s) / d(h,d,s)). Table 1 defines the knary workloads for some of the experiments described following. The entries in the table are

(i) parameters = (height, parallel degree, serial degree).
(ii) Degree of parallelism = work / critical path.
(iii) Maximum speedup = work / (work / 64 + critical path) { Amdahl's formula }

3.2 Experimental Results

We have made runs on several different platforms. The results of interest come largely from the IBM SP2 which is a commonly used platform for large parallel computations. The SP2 used for the results reported herein were obtained on 64 processors of a 128 processor configuration at the NSF NPACI Computation Facility. The metric we report is speedup on 64 processors. Using the knary benchmark enables us to estimate the maximum possible speedup for a given workload on a given configuration. (Zero time communication is assumed for the computation of maximum possible speedup.) Figure 1 gives the speedup for the benchmark specified by knary cases given in Table 1. There are several features of interest in Figure 1. The first is that both load distribution and work stealing are important in attaining scalable load balancing. The most important feature is that as the number of threads (degree of parallelism) grows the speedup attained by the best load balancing algorithms remain close to the maximum possible speedup. Bearing in mind that the maximum possible speedup assumes no communication time, that there is some communication cost in the benchmark and that the runs are being made on 64 processors of a 128 processor machine with the workload on the other 64 processors sharing the switch resource the asymptotic speedup obtained is quite satisfactory. (The granularity of the units of computation on this run was chosen to be large (six seconds) to minimize the effects of communication costs.) The dip in the speedup in the range of degree of parallelism of 100-150 is due to the large granularity of the computations. Recall that CODE uses run-to-completion scheduling for computation nodes. Therefore the 6 second granularity combined with a degree of parallelism close to the number of processors means that work steals are delayed so that some processors cannot obtain work. This anomaly will be corrected by separating communication management and computation into separate threads. Additional experiments will be made with this extended version of CODE. The mapping strategies we studied include node-mapping (where nodes are mapped across processors using their path hash function), graph mapping (where graphs and all nodes within them are mapped), and a variant of graph-mapping where non-static computation nodes are always mapped locally, that is, to any processor that wants to execute it.

The results shown indicate that work-stealing plus graph-mapping provides the best speedup for high degrees of parallelism. The non-work-stealing runs are virtually indistinguishable, and perform substantially worse, as expected. It is clear that work-stealing in combination with mappings improves their performance substantially.

It was somewhat surprising that graph-mapping with local non-statics performed somewhat more poorly than graph-mapping alone. We attribute this to the large grain size. For a smaller granularity of computation, the savings in communication costs (required to obtain work via work-stealing) is more significant than the load imbalance that can result. With large grain computations (especially because of latency in message-handling), it is more important to maintain an effective load balance.

4. Related Work

Space precludes any significant discussion of related work. The only system described in the literature with almost all of the above characteristics is Paralex [1]. Paralex is a distributed system that, like CODE, is a graphical, coarse-grain dataflow programming system. Paralex, like CODE, performs an initial mapping of tasks to processors that is subsequently revised by dynamic load balancing. However, the Paralex model is a quite restricted model of parallelism: the dataflow graph must be acyclic (while CODE graphs may be cyclic and can even support recursion), and the graph itself is static. Further, the load balancing mechanism is very restricted: load can only be migrated among those processors that host the same replicated tasks In the case when no replication is used, no dynamic load balancing is performed.

We know of two other systems that are similar to CODE, in that the schemes they use for mapping are similar. Feitelson and Rudolph [5] describe a system that performs partially distributed scheduling. the intent of their system is different (to enable distributed gang-scheduling of very coarse-grain tasks), their mapping scheme is similar to CODE's, although theirs is restricted to static task graphs. The other somewhat similar system is described by Subhlok [8, 9, 10]. It provides mapping for task and data parallel programs by generating a static task graph from a source program and then partitioning the static task graph among processors. It is a feedback-driven system that allows refinement of mappings for better load balancing (using communication and memory requirements as parameters for mapping), but these mappings are entirely static.

5. Acknowledgements

This work was supported by DARPA/ITO under Contract N66001-97-C-8533, End to End Performance Modeling of Large Heterogeneous Adaptive Parallel/Distributed Computer and Communication Systems

6. REFERENCES

[1] Babaoglu, O., et. al.: Parallel Computing in Networks of Workstations with Paralex. IEEE Transactions on Parallel and Distributed Systems . 7:4, April 1996, pp. 371-384

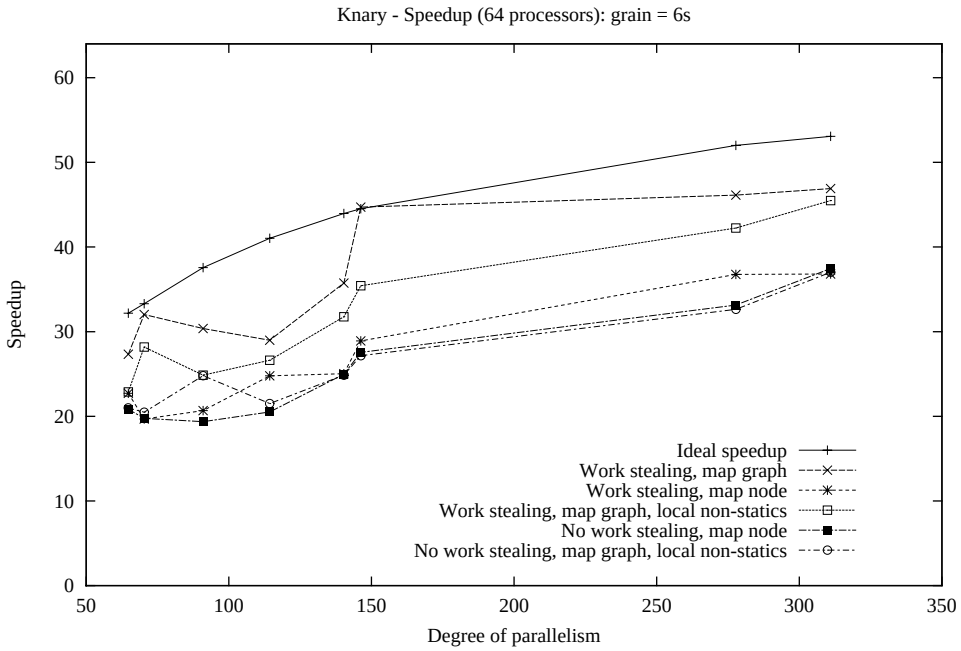

Figure 1 - Speedups for knary cases in Table 1 on IBM SP2

[2] Berger, E. The CODE Visual Parallel Programming System, WWW page. http://www.cs.utexas.edu/users/code.

[3] Blumofe, R. and Leiserson, C.E. Scheduling Multithreaded Computations by Work Stealing. Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS), pages 356-368 (Santa Fe, New Mexico, November 20-22, 1994.)

[4] Blumofe,R., et.al. Cilk: An Efficient Multithreaded Runtime System. The Journal of Parallel and Distributed Computing, 37(1), pages 55-69, August, 1996

[5] Feitelson, D.G. and Rudolph, L. Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control, Proceedings of the 1990 International Conference on Parallel Processing. Volume 1: Architecture, pp. 1-8

[6] Geist, A. et al., PVM3 User's Guide and Reference Manual, Oak Ridge National Laboratory, Tennessee, 1994.

[7] Newton, P. The CODE 2.0 Graphical Parallel Programming Language, Proceedings of the 1992 International Conference on Supercomputing, (Washington, DC, July 1992), pp. 167-177

[8] Subhlok, J. and Yang, B. A New Model for Integrated Nested Task and Data Parallel Programming", Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,(Las Vegas, NV, June 1997) pp.1-12

[9] Subhlok,J and Vondran, G. Optimal Mapping of Sequences of Data Parallel Tasks, Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (Santa Barbara, CA, July 19-21), pp.134-143",August 1995",

[10] Subhlok, J., el.al. Programming Task and Data Parallelism on a Multicomputer Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming" (May, 1993","San Diego, CA)

[11] Vokkarne, R.. Distributed Execution Environments for the CODE 2.0 Parallel Programming System}, Master's Thesis, University of Texas at Austin, Department of Computer Sciences, 1994.