Diffusive Parallelism: A Parallel Programming Model for Large Scale Distributed Computation Systems

Peter D. Stout and Brian N. Bershad
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
pds@cs.cmu.edu

June 25, 1992

Abstract

The spread of networks and powerful workstations has created an attractive source of parallel computing power. We are exploring a new parallel programming model, called diffusive parallelism, designed specifically for use with large scale, distributed computation systems. The model provides a simple, yet powerful, abstraction to the programmer, while making it possible to build a secure, robust, distributed computation system in the presence of long delays, failures, and untrusted user programs. In contrast, most existing distributed computation systems have attempted to extend programming models appropriate to a single node. Diffusive parallelism provides the programmer with a task heap and a weakly consistent, logically centralized, shared data store, or blackboard. We are implementing diffusive parallelism in a system called Wax.

1 Overview

We are exploring a new parallel programming model, called diffusive parallelism, for use with large-scale distributed computation systems. The model provides the programmer with easy to use programming tools, while making it possible to build a secure, robust, distributed computation system in the presence of long delays, failures, and untrusted user programs. The model consists of a distributed task heap for scheduling and a distributed blackboard for inter-process communication. The blackboard is a weakly consistent, logically centralized, shared data store. The name of our model comes from the idea that tasks and blackboard data diffuse outward from where they are created.

The spread of networks and powerful workstations has created an attractive source of parallel computing power. A number of systems have been built to turn idle workstations on a network into a distributed "multiprocessor." Existing small scale, local area systems permit sophisticated programmers to use tens of processors together on a single problem. The enormous number of machines connected to wide area networks, such as the Internet, make them attractive resources for building large-scale distributed computation systems. Such systems would be useful in solving a large number of problems, such as circuit simulations, primality testing, film animation, and NP-complete approximations, that can be decomposed into many smaller pieces. In these problems, the potential coarse-grain parallelism often exceeds the parallel processing facilities at any single site.

Existing local area computation systems, such as Condor [6], Marionette [8], Sprite [4], and Zilla [3], are usually designed to function as "virtual" extensions of the machine where a computation is started. The main characteristic of these systems is that they provide single system semantics for consistency, security, and reliability. In Condor, Marionette, and Sprite, a central agent is responsible for providing the same consistent view of shared data that the operating system would provide on a single machine. Security is provided by requiring system users to have accounts on all machines they use. Because these systems are designed as virtual extensions of a single machine, they all require that the initial machine be available during an entire computation.

While the single-system programming model is a familiar one to most programmers, it is not particularly well suited to distributed computing environments. There are several problems that are not well addressed by this "virtual machine" model. First, as the number of machines involved in a computation increases, so does the probability that one of those machines will fail. Second, the cost of inter-process communication
in a distributed environment, particularly, one using a wide area network, is greater than on a single machine. This is true both in terms of bandwidth and, perhaps more importantly, latency. Third, strict consistency in a distributed system is harder to maintain than on a single (multi-)processor. This is due to both the potentially greater number of machines involved and the increased communication overhead of local/wide area networks.

By using a programming model that is designed for distributed computing, it is possible to build a more robust distributed computing system. In addition, by suitably restricting the process model, it will also be possible to build a system that provides security guarantees to the resource providers. To demonstrate that diffusive parallelism works, we are building a prototype system called Wax.

1.1 Related Work

Several systems, including Amoeba [5] and Isis [1], use a distributed programming model based on process groups. In this model, groups of processes cooperate using reliable multicast. Typically, the multicast mechanism implemented in these systems provides causal or global ordering of messages among group members. When a member of a group fails or becomes unreachable, the remaining members of the group use some form of a recovery protocol to reform the group.

The process group model is aimed at a different type of distributed environment from that targeted by diffusive parallelism. While the process group model is well suited for local area systems with moderate numbers of processes, it would have problems in a large-scale, wide-area environment. First, providing reliable multicast requires buffering messages until all members of the group have received them. In a high latency environment, such as, across a wide area network, this buffering requirement may be prohibitively expensive. Second, the mechanisms used to order messages are unlikely to scale to large process groups. Third, as the number of processes in a process group grows the probability of failure, and the cost of recovering from that failure, increases. In general, the process group model is not well suited to large scale, distributed computing, where large amounts computing power are available for speculative execution and communication.

The rest of this paper presents the details of diffusive parallelism and describes the prototype system we are building. Section 2 describes diffusive parallelism and explores the purpose of the various parts of the model. Section 3 describes our prototype system called Wax. The paper concludes with a brief summary.

2 Diffusive Parallelism

In diffusive parallelism, the user decomposes computations into tasks that the distributed computation system stores in a task heap. This task heap diffuses across the available processors, with cooperating tasks communicating via a blackboard, which is a weakly consistent, logically centralized, shared data store. The user adds tasks to the task heap, the system executes the tasks, and the user reads the results from the blackboard. The system propagates the information at its convenience. While the system controls where programs are executed, users can modify the execution flow of their programs using feedback tools.

Diffusive parallelism is primarily intended for use in large-scale distributed computation systems. As a result, computations with relatively coarse-grain parallelism and limited inter-task communication will perform best. The model is not appropriate for computations with tightly-coupled tasks or strong data consistency requirements.

As an example of how diffusive parallelism works, consider a job to compute and display the Mandelbrot set. To do this on a system implementing diffusive parallelism, the user would submit an initial task, with arguments specifying the size and origin of the pixel grid, to the computation system. This initial task would then divide the pixel grid into smaller pieces, for example, 1000 pixel chunks, and spawn tasks to compute each of the chunks. As each task finished computing, it would store its pixel values in the blackboard. When ready to display the results, the user would run a program to retrieve the pixel values from the blackboard and display them as appropriate. Because of the asynchronous nature of computations in diffusive parallelism, this program might need to wait for pieces of the image that have not yet been computed.

Diffusive parallelism is designed to satisfy two opposing constraints. The first constraint is that a system supporting diffusive parallelism should be easy to program. The second constraint is that it be feasible to implement a system supporting diffusive parallelism. This constraint often required limiting what the programmer is allowed to do. The rest of this section describes the reasoning behind the design of the diffusive parallelism process and data models.

2.1 Process Model

One of the purposes of the diffusive parallelism process model is to define what users may do, so that it is possible to implement a secure, robust, distributed computation system. In order to build a secure system, it is necessary to limit a computation's ability to inter-
act with the operating system and other processes on the machine on which it is executing. Therefore, diffusive parallelism provides tasks with only two resources: processor cycles and the blackboard. In using a system implementing diffusive parallelism, a programmer may create new tasks, monitor the status of existing tasks, destroy tasks, use a blackboard to communicate between tasks, and compute. Programmers may not create sub-processes, access files, or establish inter-process communication links other than through the blackboard, because any of these activities could be used to explore or attack the machine that is being used as a cycle server.

Restricting the resources available to the programmer also makes it easier for the computation system to automatically manage, that is, assign and migrate, executing tasks. A system that provides automatic process management can potentially make a larger pool of machines available to the user of the system. By enabling the system to assign tasks, a user no longer needs to have direct access to, or even know of, the machines that are used. Given the large variety of machines connected to a network, though, the user of a distributed computation system may still need to be able to specify the types of machines on which a task can run.

The use of a task heap model, rather than a hierarchical or master-slave model, offers the possibility of more robust implementations. Diffusing the task heap across multiple sites eliminates the possibility that a single failure, such as, the initial or master task crashing, can stop a computation. With a distributed task heap, all of the sites that have not crashed may continue working as long as they have tasks to execute.

2.2 Data Model

The diffusive parallelism data model consists of a logically centralized shared data store that we refer to as a blackboard. Our blackboard is similar to a Linda [2] tuple space, in that information is stored as (key,value)-tuples. The data on a blackboard are only weakly consistent, that is, different tasks may read different values for a tuple or a tuple may only be visible to some tasks. This means that, in contrast to Linda, blackboard tuples cannot reliably be used for synchronization.

The asynchronous diffusion of information across the network and the weak data consistency requirements provide reasonable performance across local or wide area networks, and simplify system implementation. When communication is expensive, it may be cheaper to recompute a result instead of moving data. The weak consistency model also allows systems that implement diffusive parallelism to cache information, in order to reduce network communication, and to continue processing during network partitions. The ability to tolerate network failures becomes more important as programmers attempt to use more machines for a single computation.

The weak data consistency requirement allows a system implementing diffusive parallelism to propagate blackboard information at its convenience. This allows a system implementing diffusive parallelism to avoid synchronous updates and/or central servers, either of which would make scaling the system difficult. In addition, by not keeping everything strongly consistent, those computations, such as, primality testing, that only need weak consistency guarantees can avoid unneeded overhead.

2.3 Failure Model

In the diffusive parallelism model, three things can fail: a machine executing a task, a network link, and the machine that started a computation. The failure of a machine executing a task is handled by using another machine to execute the task. If a network link fails, the weak consistency of the data model allows the computation to continue processing on either side of the partition. Failure of the machine that started a computation is harder to handle, because that machine is the user's connection to the computation. When that machine is unavailable, the user may not be able to access the computation's blackboard. An implementation of diffusive parallelism can minimize this problem by providing replicated access to the task heap and blackboard.

Computations in the diffusive parallelism model can be viewed as having an expansion phase and a contraction phase. During the expansion phase, new tasks and blackboard data are produced and diffused. During the contraction phase, the results of the computation diffuse back to the machine that started the computation. If a computation stops expanding and the original machine is available, then the blackboard will eventually contain the computation's final output.

3 Prototype System

We are currently implementing a prototype system, called Wax, that supports diffusive parallelism. The system will run on a variety architectures running UNIX or Mach. It is intended to provide users with access to the idle resources connected to a wide area network.

The implementation uses the nation-wide Andrew file system (AFS) [7] to store and transfer the task heap and blackboards. We decided to use AFS because of its availability and the facilities (local caching, intra-cell security) it provides. In addition, AFS was designed to
support a large number of users and machines spread across a wide area network.

The machines that make up Wax will be grouped into autonomous administrative domains. These domains serve two purposes in our implementation. First, they make the system easier to scale by providing an intermediate level for data diffusion. Domains only need to know about other domains; they do not need to know how many or which machines make up each domain. Domains also provide an additional level for the caching mechanism. Second, domains are important for administrative reasons. The machines that are part of Wax belong to individual organizations that will want to maintain control of their machines. By dividing the system into domains, we make it possible for each domain to manage its machines and define access policies for external users. For administrative convenience, Wax's domains will typically correspond to AFS cells.

A task is the basic unit of execution in Wax. Every task is a member of a task family or job. A user submits a job for processing by adding it to the task heap. Tasks may spawn new tasks to work in parallel on the job. All members of a task family are considered to be peers; spawning a new task does not create any hierarchical relationship between the spawning and spawned tasks.

Wax does not interpret the contents of a blackboard, leaving each task family free to use its own blackboard organization. A common blackboard defines a collection of tasks as belonging to a task family. As tasks diffuse through the wide area multi-processor, the information in a blackboard may be cached to speed access by executing tasks. Because the system only provides weak consistency for the cached data, caching can be disabled on a per blackboard basis so that higher level programming models, such as Linda, which require stronger consistency guarantees, can be supported. Blackboards will be implemented as AFS directories. If a blackboard is cachable, then its tuples will diffuse independently.

3.1 System Processes

Wax is implemented using three types of active agents: Task Managers, Host Managers, and Communication Managers. The Task Managers coordinate processing within a domain and drive the diffusion of the task heap. The Host Managers are Wax's local agents on the machines executing tasks. The Communication Managers transfer tasks and blackboard data between domains. The various components of Wax communicate via remote procedure calls.

Each domain has one or more Task Managers that maintain the task heap. When a Host Manager requests a task, the Task Manager selects the highest priority task whose resource requirements are satisfied by the requesting machine. After recording which machine is receiving the task, the Task Manager sends the task descriptor to the Host Manager. The Task Manager monitors the status of all machines executing tasks. If it decides that a machine is down or malfunctioning, it marks the task as available for assignment to another machine. It also manages the AFS access control lists that are used to protect the task heap and blackboards. The Task Managers also drive the diffusion of the task heap by providing the Communication Manager with tasks to diffuse and by monitoring already diffused tasks.

Every machine that executes tasks for the wide area multi-processor has a Host Manager running on it. The Host Manager has three purposes: it monitors activity on its machine, it starts tasks for the system, and it serves as a task's connection to the system. When the Host Manager determines that its machine is idle, it contacts the local Task Manager, requesting a task to execute. If there is an appropriate task available, the Task Manager sends it to the Host Manager. The Host Manager then starts the task and waits for it to finish. While a task is running, the Host Manager acts as the task's connection to the rest of the world. This involves processing the task's blackboard operations and relaying task heap operations to the Task Manager. The Host Manager also periodically "pings" the Task Manager to allow the Task Manager to detect host crashes and network partitions. If a local user becomes active while the task is executing, the Host Manager kills the task and notifies the Task Manager. After a task has terminated, the Host Manager frees any resources that it had used on the local machine.

Communication Managers diffuse tasks among domains and provide remote domains with copies of cachable blackboard objects. They also enforce restrictions on the spread of information between domains. The diffusion process does not create a hierarchy of domains. In particular, the creator of a job does not coordinate the processing of the job. This decentralized organization allows the system to continue functioning in the face of network partitions. Users are able to ask the system to update the local view of a blackboard by querying other domains for additional information.

3.2 Status

The implementation of Wax is proceeding in two phases. In the first phase, we have built an emulation library that supports the Wax user interface and programming environment on a single node. In the second phase, we are implementing the Wax interface in a wide-area distributed system, as described in this paper. Our two phase approach has allowed us to stress test our pro-
gramming model without having to deal with many of the implementation details of distributed systems. The emulator runs on a variety of machines that support the Mach operating system. We have started implementing the distributed version of Wax.

As a result of our initial experiences with the emulator, we have made some changes that we believe improve the system's usability. In particular, we have made the user interface synchronous but multi-threaded to make it easier for programmers to interleave computation and waiting for blackboard data produced by other tasks.

4 Summary

We have described a new parallel programming model called diffusive parallelism that is suitable for large scale, distributed computation systems. Diffusive parallelism provides programmers with a simple, yet powerful, computation model: computations are divided into tasks that communicate via a logical centralized blackboard. A key aspect of our model is that it does not present a single system image, thereby tolerating the failure of networks and machines used to implement the model. We are in the process of building a prototype system, called Wax, to demonstrate that diffusive parallelism is a useful alternative to the programming models used by existing distributed computation systems.

References


