Arjav J. Chakravarti and Gerald Baumgartner and Mario Lauria

Self-Organizing Scheduling on the Organic Grid

CRC PRESS
Boca Raton Ann Arbor London Tokyo

Contents

0-8493-0052-5/00/$0.00+$.50

1

Self-Organizing Scheduling on the Organic Grid

Arjav Chakravarti, Gerald Baumgartner, and Mario Lauria
MathWorks, Inc; Louisiana State University; Ohio State University

CONTENTS

The Organic Grid is a biologically inspired and fully-decentralized approach to the organization of computation that is based on the autonomous scheduling of strongly mobile agents on a peer-to-peer network. Through the careful design of agent behavior, the emerging organization of the computation can be customized for different classes of applications.

We report our experience in adapting the general framework to run two representative applications on our Organic Grid prototype: the NCBI BLAST code for sequence alignment, and Cannon's algorithm for matrix multiplication. The first is an example of an independent task application, a type of application commonly used for grid scheduling research because of its easily decomposable nature and absence of intra-node communication. The second is a popular block algorithm for parallel matrix multiplication, and represents a challenging application for grid platforms because of its highly structured and synchronous communication pattern.

Agent behavior completely determines the way computation is organized on the Organic Grid. We intentionally chose two applications at opposite ends of the distributed computing spectrum having very different requirements in terms of communication topology, resource use, and response to faults. We detail the design of the agent behavior and show how the different requirements can be satisfied. By encapsulating application code and scheduling functionality into mobile agents, we decouple both computation and scheduling from the underlying grid infrastructure. In the resulting system every node can inject a computation onto the grid; the computation naturally organizes itself around available resources.

0-8493-0052-5/00/$0.00+$.50

1.1 Introduction

Many scientific fields, such as genomics, phylogenetics, astrophysics, geophysics, computational neuroscience, or bioinformatics, require massive computational power and resources, which might exceed those available on a single supercomputer. There are two drastically different approaches for harnessing the combined resources of a distributed collection of machines: traditional grid computing schemes and centralized master-worker schemes.

Research on Grid scheduling has focused on algorithms to determine an optimal computation schedule based on the assumption that sufficiently detailed and up to date knowledge of the system state is available to a single entity (the metascheduler) [1, 3, 20, 41]. While this approach results in a very efficient utilization of the resources, it does not scale to large numbers of machines. Maintaining a global view of the system becomes prohibitively expensive and unreliable networks might even make it impossible.

A number of large-scale systems are based on variants of the master/workers model [2, 6, 13, 15, 16, 21, 24, 25, 30, 31, 39, 46]. The fact that some of these systems have resulted in commercial enterprises shows the level of technical maturity reached by the technology. However, the obtainable computing power is constrained by the performance of the single master (especially for data-intensive applications) and by the difficulty of deploying the supporting software on a large number of workers.

At a very large scale much of the conventional wisdom we have relied upon in the past is no longer valid, and new design principles must be developed. First, very few assumptions (if any) can be made about the systems, in particular about the amount of knowledge available about the system. Second, since the system is constantly changing (in terms of operating parameters, and resource availability), self-adaption is the normal mode of operation and must be built in from the start. Third, the deployment of the components of an infrastructure is a non-trivial issue, and should be one of the fundamental aspects of the design. Fourth, any dependence on specialized entities such as schedulers, masters nodes, etc., needs to be avoided unless such entities can be easily replicated in a way that scales with the size of the system.

We propose a completely new approach to large scale computations that addresses all these points simultaneously with a unified design methodology. While known methods of organizing computation on large systems can be traced to techniques that were first developed in the context of parallel computing on traditional supercomputers, our approach is inspired by the organization of complex systems. Nature provides numerous examples of the emergence of complex patterns derived from the interactions of millions of organisms that organize themselves in an autonomous, adaptive way by following relatively simple behavioral rules. In order to apply this approach to the organization of computation over large complex systems, a computation must be broken into small self-contained chunks, each capable of expressing autonomous behavior in its interaction with other chunks.

The notion that complex systems can be organized according to local rules is not new. Montresor et al. [33] showed how an ant algorithm could be used to solve the problem of dispersing tasks uniformly over a network. Similarly, the RIP routing table update protocol uses simple local rules that result in good overall routing behavior. Other examples include autonomous grid scheduling protocols [26] and peer-to-peer file sharing networks [19, 40].

Our approach is to encapsulate computation and behavior into mobile agents, which deliver the computation to available machines. These mobile agents then communicate with one another and organize themselves in order to use the resources effectively. We envision a system where every node is capable of contributing resources for ongoing computations, and starting its own arbitrarily large computation. Once an application is started at a node, e.g., the user's laptop, other nodes are called in to contribute resources. New mobile agents are created that, under their autonomous control, readily colonize the available resources and start computing.

Only minimal support software is required on each node, since most of the scheduling infrastructure is encapsulated along with the application code inside an agent. In our experiments we only deployed a JVM and a mobile agent environment on each node. The scheduling framework described in this chapter is being implemented as a library that a developer will be able to adapt for his or her purposes.

Computation organizes itself on the available nodes according to a pattern that emerges from agent-to-agent interaction. In the simplest case, this pattern is an overlay tree rooted at the starting node; in the case of a data intensive application, the tree can be rooted at one or more separate, presumably well-connected machines at a supercomputer center. More complex patterns can be developed as required by the applications requirements, either by using different topologies than the tree, and/or by having multiple overlay networks each specialized for a different task.

In our system, the only knowledge each agent relies upon is what it can derive from its interaction with its neighbor and with the environment, plus an initial *friends list* needed to bootstrap the system. The nature of the information required for successful operation is application dependent and can be customized. E.g., for our first (data-intensive) application, both neighbor computing rate and communication bandwidth of the intervening link were important; this information was obtained using feedback from the ongoing computation.

Agent behavior completely determines the way computation is organized. In order to demonstrate the feasibility and generality of this approach, we report our experience in designing agent behavior for running two representative applications on an Organic Grid.

The first, the NCBI BLAST code for sequence alignment, is an example of an independent task application. This type of application is commonly used for grid scheduling research because of its easily decomposable nature and absence of intra-node communication. The second, Cannon's algorithm for matrix multiplication, is a block algorithm for parallel matrix multiplication that interleaves communication with computation. Because of its highly structured and synchronous communication pattern it is a challenging application for grid platforms.

The most important contribution of the experiments described here is to demon-

strate how the very different requirements — in terms of communication topology, resource use, and response to faults — of each of these two applications at the opposite ends of the distributed computing spectrum can be satisfied by the careful design of agent behavior in an Organic Grid context.

1.2 Background and Related Work

This section contains a brief introduction to the critical concepts and technologies used in our work on autonomic scheduling, as well as the related work in these areas. These include: Peer-to-Peer and Internet computing, self-organizing systems and the concept of emergence, and strongly mobile agents.

1.2.1 Peer-to-Peer and Internet Computing

The goal of utilizing the CPU cycles of idle machines was first realized by the Worm project [23] at Xerox PARC. Further progress was made by academic projects such as Condor [30]. The growth of the Internet made large-scale efforts like GIMPS [46], SETI@home [39] and folding@home [15] feasible. Recently, commercial solutions such as Entropia [13] and United Devices [44] have also been developed.

The idea of combining Internet and peer-to-peer computing is attractive because of the potential for almost unlimited computational power, low cost, ease and universality of access — the dream of a true Computational Grid. Among the technical challenges posed by such an architecture, scheduling is one of the most formidable — how to organize computation on a highly dynamic system at a planetary scale while relying on a negligible amount of knowledge about its state.

1.2.2 Decentralized Scheduling

Decentralized scheduling has recently attracted considerable attention. Two-level scheduling schemes have been considered [22, 38], but these are not scalable enough for the Internet. In the scheduling heuristic described by Leangsuksun et al. [29], every machine attempts to map tasks on to itself as well as its K best neighbors. This appears to require that each machine have an estimate of the execution time of subtasks on each of its neighbors, as well as of the bandwidth of the links to these other machines. It is not clear that their scheme is practical in large-scale and dynamic environments.

G-Commerce was a study of dynamic resource allocation on the Grid in terms of computational market economies in which applications must buy resources at a market price influenced by demand [45]. While conceptually decentralized, if implemented this scheme would require the equivalent of centralized commodity markets (or banks, auction houses, etc.) where offer and demand meet, and commodity prices

can be determined.

Recently, a new autonomous and decentralized approach to scheduling has been proposed to address the needs of large grid and peer-to-peer platforms. In this bandwidth-centric protocol, the computation is organized around a tree-structured overlay network with the origin of the tasks at the root [26]. Each node sends tasks to and receives results from its K best neighbors, according to bandwidth constraints. One shortcoming of this scheme is that the structure of the tree, and consequently the performance of the system, depends completely on the initial structure of the overlay network. This lack of dynamism is bound to affect the performance of the scheme and might also limit the number of machines that can participate in a computation.

1.2.3 Self-Organization of Complex Systems

The organization of many complex biological and social systems has been explained in terms of the aggregations of a large number of autonomous entities that behave according to simple rules. According to this theory, complicated patterns can emerge from the interplay of many agents — despite the simplicity of the rules [18, 43]. The existence of this mechanism, often referred to as *emergence*, has been proposed to explain patterns such as shell motifs, animal coats, neural structures, and social behavior. In particular, complex behaviors of colonial organisms such as social insects (e.g., ants or bees) have been studied in detail, and their applications to the solution of classic computer science problems such as task scheduling and TSP has been proposed [4, 33].

The dynamic nature and complexity of mobile ad-hoc networks (MANETs) has motivated some research in self-organization as an approach to reducing the complexity of systems installation, maintenance, and management. Self-organizing algorithms for several network functions of MANETs have been proposed, including topology control and broadcast [8, 47]. Recently, the network research community has even tried to formalize the concept of self-organization; the four design paradigms proposed by Prehofer and Bettstetter represent a first attempt to provide guidelines for developing a self-organized network function [35].

In a departure from the methodological approach followed in previous projects, we did not try to accurately reproduce a naturally occurring behavior. Rather, we started with a problem and then designed a completely artificial behavior that would result in a satisfactory solution to it.

Our work is somewhat closer to the self-organizing computation concept explored in the Co-Fields project [32]. The idea behind Co-Fields is to drive the organization of autonomous agents through artificial potential fields.

Our work was inspired by a particular version of the emergence principle called Local Activation, Long-range Inhibition (LALI) [42]. The LALI rule is based on two types of interactions: a positive, reinforcing one that works over a short range, and a negative, destructive one that works over longer distances. We retain the LALI principle but in a different form: we use a definition of distance which is based on a performance-based metric. Nodes are initially recruited using a friends list (a list of some other peers on the network) in a way that is completely oblivious of distance,

therefore propagating computation on distant nodes with the same probability as close ones. During the course of the computation the agent behavior encourages the propagation of computation among well-connected nodes while discouraging the inclusion of distant (i.e., less responsive) agents.

1.2.4 Strongly Mobile Agents

To make progress in the presence of frequent reclamations of desktop machines, current systems rely on different forms of checkpointing: automatic, e.g., SETI@home, or voluntary, e.g., Legion. The storage and computational overheads of checkpointing put constraints on the design of a system. To avoid this drawback, desktop grids need to support the asynchronous and transparent migration of processes across machine boundaries.

Mobile agents [28] have relocation autonomy. These agents offer a flexible means of distributing data and code around a network, of dynamically moving between hosts as resource availability varies, and of carrying multiple threads of execution to simultaneously perform computation, decentralized scheduling, and communication with other agents. There have been some previous attempts to use mobile agents for grid computing or distributed computing [5, 17, 34, 36].

The majority of the mobile agent systems that have been developed until now are Java-based. However, the execution model of the Java Virtual Machine does not permit an agent to access its execution state, which is why Java-based mobility libraries can only provide *weak mobility* [14]. Weakly mobile agent systems, such as IBM's Aglets framework [27] do not migrate the execution state of methods. The `go()` method, used to move an agent from one virtual machine to another, simply does not return. When an agent moves to a new location, the threads currently executing in it are killed without saving their state. The lifeless agent is then shipped to its destination and restarted there. Weak mobility forces programmers to use a difficult programming style, i.e., the use of callback methods, to account for the absence of migration transparency.

By contrast, agent systems with *strong mobility* provide the abstraction that the execution of the agent is uninterrupted, even as its location changes. Applications where agents migrate from host to host while communicating with one another, are severely restricted by the absence of strong mobility. Strong mobility also allows programmers to use a far more natural programming style.

The ability of a system to support the migration of an agent at any time by an external thread, is termed *forced mobility*. This is essential in desktop grid systems, because owners need to be able to reclaim their resources. Forced mobility is difficult to implement without strong mobility.

We provide strong and forced mobility for the full Java programming language by using a preprocessor that translates an extension of Java with strong mobility into weakly mobile Java code that explicitly maintains the execution state for all threads as a mobile data structure [11, 12]. For the target weakly mobile code we currently use IBM's Aglets framework [27]. The generated weakly mobile code maintains a movable execution state for each thread at all times.

1.3 Design of an Organic Grid

The purpose of this section is to describe the architecture of the proof-of-concept, small-scale prototypes of the Organic Grid we have built so far.

1.3.1 Agent Behavior

In designing the behavior of the mobile agents, we faced the classic issues of performing a distributed computation in a dynamic environment: distribution of the data, discovery of new nodes, load balancing, collection of the results, tolerance to faults, detection of task completion. The solutions for each of these issues had to be cast in terms of one-to-one interactions between pairs of agents and embedded in the agent behavior. Using an empirical approach, we developed some initial design decisions and we then refined them through an iterative process of implementation, testing on our experimental testbed, performance analysis, redesign and new implementation. To facilitate the process we adopted a modular design in which different aspects of the behavior were implemented as separate and well identified routines.

As a starting point in our design process, we decided to organize the computation around a tree-based overlay network that would simplify load balancing and the collection of results. Since such a network does not exist at the beginning of the computation, it has to be built on the fly as part of the agents' take-over of the system.

In our system, a computational task represented by an agent is initially submitted to an arbitrary node in the overlay network. If the task is too large to be executed by a single agent in a reasonable amount of time, agents will clone themselves and migrate to other nodes; the clones will be assigned a small section of the task by the initiating agent. The new agents will complete the subtasks that they were assigned and return the results to their parent. They will also, in turn, clone and send agents to available nodes and distribute subtasks to them. The overlay network is constituted by the connections that are created between agents as the computation spreads out.

For our preliminary work we used the Java-based Aglets weak mobility library, on top of which we added our own strong mobility library. An Aglets environment is set up when a machine becomes available (for example when the machine has been idle for some time; in our experiments we assumed the machines to be available at all times).

Every machine has a list of the URLs of other machines that it could ask for work. This list is known as the *friend-list*. It is used for constructing the initial overlay network. The problem of how to generate this initial list was not addressed in our work; one could use one of the mechanisms used to create similar lists in tools such as Gnutella, CAN [37], and Chord [40].

The environment creates a stationary agent, which asks the friends for work by sending them messages. If a request arrives at a machine that has no computation running on it, the request is ignored. Nothing is known about the configurations

of the machines on the friend-list, or of the bandwidths or latencies of the links to them, i.e., the algorithm is zero-knowledge and appropriate for dynamic, large-scale systems.

A large computational task is written as a strongly mobile agent. This task should be divisible into a number of independent and identical subtasks by simply dividing the input data. A user sets up the agent environment on his/her machine and starts up the computation agent. One thread of the agent begins executing subtasks sequentially. This agent is now also prepared to receive requests for work from other machines. On receiving such a request, it checks whether it has any uncomputed subtasks, and if it does, it creates a clone of itself and sends that clone to the requesting machine. The requester is now this machine's *child*.

A clone is ready to do useful work as soon as it reaches a new location. It asks its parent for a certain number of subtasks s to work on. When the parent sends the subtasks, one of this agent's threads begins to compute them. Other threads are created as needed to communicate with the parent or to respond to requests from other machines. When such a request is received, the agent clones itself and dispatches its own clone to the requester. The computation spreads in this manner. The topology of the resulting overlay network is a tree with the originating machine at the root node.

When the subtasks on a machine have been completely executed, the agent on that machine requests more subtasks to work an from its parent. The parent attempts to comply. Even if the parent does not have the requested number of subtasks, it will respond and send its child what it can. The parent keeps a record of the number of subtasks that remain to be sent, and sends a request for those tasks to its own parent.

Every time a node of the tree obtains some r results, either computed by itself or obtained from a child, it needs to send the results to its parent. It also sends along a measurement of the time that has elapsed since the last time it computed r results. The results and the timing measurement are packaged into a single message. At this point, the node also checks whether its own — or any of its children's — requests were not fully satisfied. If that is the case, a request for the remaining number of subtasks is added to the message and the entire message is sent to the node's parent. The parent then uses the timing measurements to compare the performance of its children and to restructure the overlay network. The timing measurement was built into the agent behavior in order to provide some feedback on its own performance (in terms of both computational power and communication bandwidth).

1.3.2 Details of the Agent Behavior

Maintenance of Child-lists A node cannot have an arbitrarily large number of children because this will adversely affect the synchronization delay at that node. Since the data transfer times of the independent subtasks are large, a node might have to wait for a very long time for its request to be satisfied. Therefore, each node has a fixed number of children, c. The number of children also should not be too small so as to avoid deep trees which will lead to long delays in propagating the data from the root to the leaf nodes. These children are ranked by the rate at which they send in results. When a child sends in r results with the time that was required to

obtain them, its ranking is updated. This ranking is a reflection of the performance of not just a child node, but of the entire subtree with the child node as its root. This ranking is used in the restructuring of the tree as described below.

Restructuring of the Overlay Network The topology of a typical overlay network is a tree with the root being the node where the original computation was injected. It is desirable for the best-performing nodes to be close to the root. This minimizes the communication delay between the root and the best nodes, and the time that these nodes need to wait for their requests to be handled by the root. This principle to improve system throughput is applicable down the tree, i.e., a mechanism is required to structure the overlay network such that the nodes with the highest throughput are closer to the root, while those with low throughput are near the leaves.

A node periodically informs its parent about its best-performing child. The parent then checks whether its grandchild is present in its list of former children. If not, it adds the grandchild to its list of potential children and tells this node that it is willing to consider the grandchild. The node then informs the grandchild that it should now contact its grandparent directly. This results in fast nodes percolating towards the root of the tree.

When a node updates its child-list and decides to remove its slowest child, *sc*, it does not simply discard the child. It sends *sc* a list of its other children, which *sc* attempts to contact in turn. If *sc* had earlier been propagated to this node, a check is made as to whether *sc*'s original parent is still a child of this node. In that case, *sc*'s original parent, *op*, is placed first in the list of nodes being sent for *sc* to attempt to contact. Since *sc* was *op*'s fastest child at some point, there is a good chance that it will be accepted by *op* again.

Fault Tolerance A node depends on its parent to supply it with new subtasks to work on. However, if the parent were to become inaccessible due to machine or link failures, the node and its own descendents would be unable to do any useful work. A node must be able to change its parent if necessary; every node keeps a list of a of its ancestors in order to accomplish this. A node obtains this list from its parent every time the parent sends it a message. The updates to the ancestor-list take into account the possibility of the topology of the overlay network changing frequently.

A child waits a certain user-defined time for a response after sending a message to its parent — the a-th node in its ancestor-list. If the parent is able to respond, it will, irrespective of whether it has any subtasks to send its child at this moment or not. The child will receive the response, check whether its request was satisfied with any subtasks, and begin waiting again if that is not the case.

If no response is obtained within the timeout period, the child removes the current parent from its ancestor-list and sends a message to the $(a - 1)$-st node in that list. This goes on until either the size of the list becomes 0, or an ancestor responds to this node's request.

If a node's ancestor-list does go down to size 0, the node has no means of obtaining any work to do. The mobile agent that computes subtasks informs the agent environ-

ment that no useful work is being done by this machine, and then self-destructs. Just as before, a stationary agent begins to send out requests for work to a list of friends.

However, if an ancestor does respond to a request, it becomes the parent of the current node and sends a new ancestor-list of size *a* to this node. Normal operation resumes with the parent sending subtasks to this node and this node sending requests and results to its parent.

Prefetching A potential cause of slowdown in the basic scheduling scheme described earlier is the delay at each node due to its waiting for new subtasks. This is because it needs to wait while its requests propagate up the tree to the root and subtasks propagate down the tree to the node.

We found that it is beneficial to use prefetching for reducing the time that a node waits for subtasks. A node determines that it should request *t* subtasks from its parent. The node then makes an optimistic prediction of how many subtasks it might require in the future and requests $t + i(t)$ subtasks from its parent. When a node finishes computing one set of subtasks, more subtasks are readily available for it to work on, even as a request is submitted to the parent. This interleaving of computation and communication reduces the time for which a node is idle.

While prefetching will reduce the delay in obtaining new subtasks to work on, it also increases the amount of data that needs to be transferred at a time from the root to the current node, thus increasing the synchronization delay and data transfer time. This is why excessively aggressive prefetching will end up performing worse than a scheduling scheme with no prefetching.

1.4 Measurements

We have demonstrated the applicability of our scheduling approach using two very different types of applications, the National Center for Biotechnology Information (NCBI) basic local alignment search tool (BLAST) code for sequence alignment [10], and Cannon's algorithm for parallel matrix multiplication [9]. In this section, we summarize the results of these experiments, with emphasis on Cannon's algorithm.

1.4.1 Independent Task Application: BLAST

For our initial experiments, we used BLAST, an application that is representative of a class of applications commonly used in grid scheduling research called an *independent task application* (or ITA) [10]. The lack of communication between the tasks of an ITA simplifies scheduling, because there are no constraints on the order of evaluation of the tasks.

The application consisted of 320 tasks, each matching a given 256KB sequence agains a 512KB chunk of a data base. When arriving at a node, a mobile agent in-

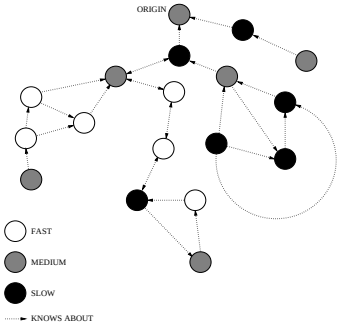

FIGURE 1.1
BLAST: Original Configuration of Machines

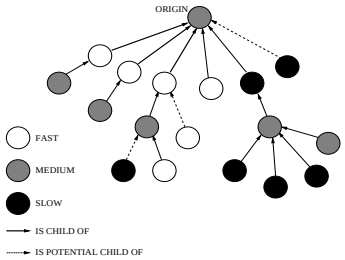

FIGURE 1.2
Final Node Organization, Result-burst size=3, With Child Propagation

stalls the BLAST executable and then repeatedly requests new tasks from its parent and returns the results to its parent until no more tasks are available. If the agent receives requests for work from an idle machine, it sends a clone of itself to the idle machine. The computation thus spreads out from its source in the form of a tree. The source distributes the data in the form of computational subtasks that flow down the tree; results flow towards the root. This same tree structure was also used as the overlay network for making scheduling decisions. In general, there could be separate overlay networks: for data distribution, for scheduling, and for communication between subtasks. For this application, there is no communication between subtasks while the overlay trees for data distribution and scheduling overlap.

We ran the experiments with an arbitrary initial configuration of the overlay network as shown in Figure 1.1. To simulate the effect of heterogeneity, we introduced delays in the application code resulting in fast, medium, and slow nodes. We performed a variety of experiments with different parameters of our scheduling algorithm, such as the width of the overlay tree or the number of results over which to average the performance of a node, and measured the running time and the time needed for the computation to reach all nodes. The parameters that resulted in the best performance were a maximum tree width of 5 and a result burst size of 3. Figure 1.2 shows the resulting overlay tree at the end of the computation, in which most of the fast nodes had been propagated closer to the root.

1.4.2 Communicating Tasks: Cannon's Matrix-Matrix Multiplication

For demonstrating the generality of the self-organizing approach and the flexibility of the Organic Grid scheduling framework, we selected a second application at the opposite end of the spectrum, characterized by a highly regular and synchronous pattern of communication — Cannon's matrix multiplication algorithm [7]. Cannon's algorithm employs a square processor grid of size $k = p \times p$ in which computation is

alternated (and can be interleaved) with communication. The initial node waits until k machines are available for the computation. Each processor in the grid then gets one tile of each of the argument matrices. After multiplying these tiles, one of the argument matrices is rotated along the first dimension of the processor grid, the other argument matrix is rotated along the second dimension of the processor grid. Each processor gets new tiles of the argument matrices and adds the result of multiplying these tiles to its tile of the result matrix. The algorithm terminates after p of these tile multiplications.

This application employs three different overlay networks: a star topology for data distribution, a torus for the communication between subtasks, and the tree overlay of the scheduling framework. The metric used for restructuring the tree was the time to multiply two matrix tiles. While for the ITA the resource constraint was the communication bandwidth of the root, for Cannon's algorithm it was the number of machines that belong to the torus. Below we report a subset of the results of our experiment; more results are available in [9]

Three aspects of the Organic Grid implementation of Cannon's matrix multiplication were sought to be evaluated: i) performance and scalability, ii) fault-tolerance and iii) decentralized selection of compute nodes. A good evaluation of this application required tight control over the experimental parameters. The experiments were therefore performed on a Beowulf cluster of homogeneous Linux machines, each with dual AMD Athlon MP processors (1.533 GHz) and 2 GB of memory. When necessary, artificial delays were introduced to simulate a heterogeneous environment. The accuracy of the experiments was improved by multiplying the matrices 16 times instead of just once.

1.4.3 Scalability

We performed a scalability evaluation by running the application on various sizes of tori and matrices. The tree adaptation mechanism was temporarily disabled in order to eliminate its effect on the experiments.

Table 1.1, and Figure 1.4 present a comparison of the running times of 16 rounds of matrix multiplications on tori with 1, 2 and 4 agents along each dimension. Superlinear speedups are observed with larger numbers of nodes because of the reduction in cache effects with a decrease in the size of the tiles stored at each machine.

Matrix	Single Agent		4×4 Agent Grid		
Size (MB)	Tile (MB)	Time (sec)	Tile (MB)	Time (sec)	Speedup
1	1	75	0.0625	34	2.2
4	4	846	0.25	43	19.7
16	16	14029	1	454	30.9

Table 1.1: Running Time on 1 and 16 Machines, 16 Rounds

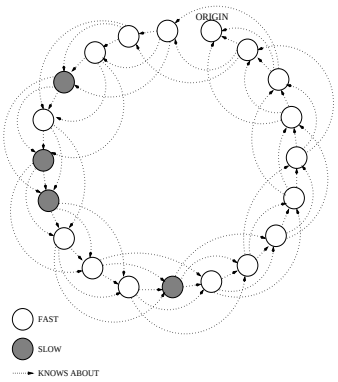

FIGURE 1.3
Cannon: Original Configuration of Machines

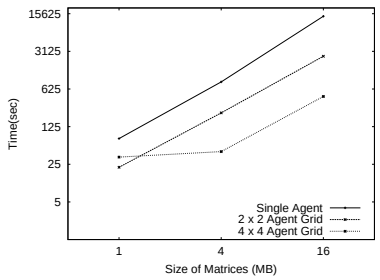

FIGURE 1.4
Running Time on 1, 4 and 16 Machines, 16 Rounds

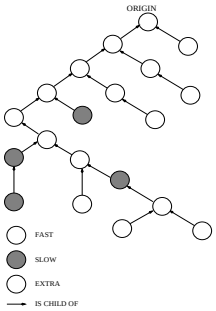

FIGURE 1.5
Original Tree Overlay

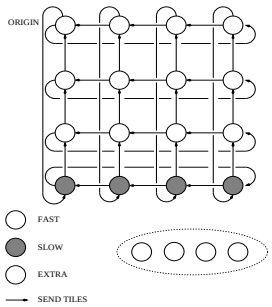

FIGURE 1.6
Original Torus Overlay

1.4.4 Adaptive Tree Mechanism

We then made use of the adaptive tree mechanism to select the best available machines for the torus in a decentralized manner. The feedback sent by each child to its parent was the time taken by the child to complete its two previous tile multiplications.

We experimented with a desktop grid of 20 agents in Figure 1.3. These 20 agents then formed a tree overlay network, of which the first 16 to contact the distribution agent were included in a torus with 4 agents along each dimension; the remaining agents acted as extras in case any faults occurred. The initial tree and torus can be seen in Figures 1.5 and 1.6 with 4 slow nodes in the torus and 4 extra, fast nodes.

The structure of the tree continually changed and the high-performance nodes were pushed up towards the root. When a fast, extra node found that one of its

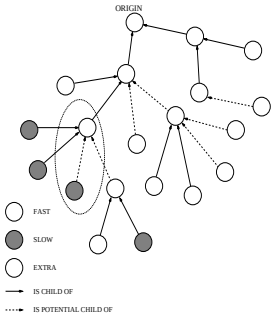

FIGURE 1.7
Tree Overlay Before Fourth Swap

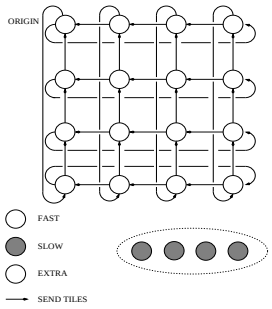

FIGURE 1.8
Torus Overlay After Fourth Swap

Slow Nodes	Extra Nodes	Time (sec)
4	0	898
0	0	462
4	4	759

Table 1.2: Running Time of 16 Rounds on 4x4 Grid, 16MB Matrix, 1MB Tiles, Adaptive Tree

Stage	Swap position on Torus	Avg. Tile Mult. Time (sec)
1–3	-	10
4	12	13
5	-	15
6	13,15	15
7–42	-	14
43	12	14
44–47	12	13
48-64	-	7

Table 1.3: Performance at Different Stages of Experiment, 4x4 Agent Grid

children was slower than itself and part of the torus, it initiated a swap of roles. The topology of the tree and the torus before and after the fourth swap are shown in Figures 1.7 and 1.8.

Each matrix multiplication on the $4x4$ agent grid had 4 tile multiplication stages; our experiment consisted of 16 rounds — 64 stages. A tile multiplication took 7 sec. on a fast node and 14 sec. on a slow one. Table 1.3 presents the average execution time of these stages. This began at 10 sec., then increased to 13 sec. before the first swap took place. The fast nodes were inserted into the torus on stages 4, 6 and 43. Once the slow nodes had been swapped out, the system required 4 rounds until all the 16 agents sped up and reached high steady-state performance. The effect of this on overall running time can be seen in Table 1.2.

While the adaptive tree mechanism undoubtedly results in a performance improvement in the presence of high-performance extra nodes, it also introduces some overhead. Nodes provide feedback to their parents who, in turn, rank their children and propagate the best ones. We first ran the Cannon application without any extra nodes present, and then disabled the adaptive tree mechanism for a second set of experi-

No Adaptation			Adaptation		
Slow Nodes	Extra Nodes	Time (sec)	Slow Nodes	Extra Nodes	Time (sec)
4	0	898	4	0	899
0	0	454	0	0	462

Table 1.4: Overhead of Adaptive Tree, 16 Rounds, 4x4 Grid, 16MB Matrix, 1MB Tiles

No. of Failures	Failures on Column		Failures on Diagonal	
	Positions	Time (sec)	Positions	Time (sec)
0	-	454	-	454
1	5	466	5	466
2	5, 9	479	6, 9	464
3	5, 9, 13	486	6, 9, 12	540

Table 1.5: Running Time of 16 Rounds on 4x4 Grid, 16MB Matrix, 1MB Tiles

ments. The overhead of this mechanism was negligible, as can be seen in Table 1.4.

1.4.5 Fault-Tolerance

We introduce crash failures by bringing down some machines during application execution. We were interested in observing the amount of time that the system would stall in the presence of failures. Different numbers of failures were introduced at different positions on the torus. When multiple nodes on the same column crash, they are replaced in parallel. The replacements for crashes on a diagonal occur sequentially.

The system recovers rapidly from failures on the same column and diagonal, as can be seen in Table 1.5. For a small number of crashes (1 or 2), there is little difference in the penalty of crashes on columns or diagonals. This difference increases for 3 crashes, and we expect it to increase further for larger numbers of crashes on larger tori.

1.5 Conclusions and Future Work

We have designed a desktop grid in which mobile agents are used to deliver applications to idle machines. The agents also contain a scheduling algorithm that decides which task to run on which machine. Using simple scheduling rules in each agent, a tree-structured overlay network is formed and restructured dynamically, such that well performing nodes are brought closer to important resources, thus improving the performance of the overall system.

We have demonstrated the applicability of our scheduling scheme with two very different styles of applications, an independent task application, a BLAST executable,

and an application in which individual nodes need to communicate, a Cannon-style matrix multiplication application.

Because of the unpredictability of a desktop grid, the scheduler does not have any a priori knowledge of the capabilities of the machines or the network connections. For restructuring the overlay network, the scheduler relies on measurements of the performance of the individual nodes and makes scheduling decisions using application-specific cost functions. In the case of BLAST, where the data was propagated along the same overlay tree, nodes with higher throughput were moved closer to the root to minimize congestion. In the case of Cannon's algorithm, where the data came from a separate data center, the fastest nodes were moved closer to the root, to prevent individual slow nodes from slowing down the entire application.

The common aspect in scheduling the tasks for these very different applications is that access to a resource needs to be managed. In the case of BLAST, the critical resource is the available communication bandwidth at the root and at intermediate nodes in the tree. If a node has too many children, communication becomes a bottleneck. Conversely, if a node has too few children, the tree becomes too deep and the communication delay between the root and the leaves too long. The goal for BLAST was, therefore, to limit the width of the tree and to propagate high-throughput nodes closer to the root. In the case of Cannon's algorithm, the critical resource is the communication torus. Since any slow node participating in the torus would slow down the entire application, the goal is to propagate the fast nodes closer to the root and to keep the slower nodes further from the root.

By selecting the appropriate parameters to our scheduling algorithm, an application developer can tune the scheduling algorithm to the characteristics of an individual application. This choice of parameters includes constraints on how the overlay tree should be formed, e.g., the maximum width of the tree, and a metric with which the performance of individual nodes can be compared to decide which nodes to propagate up in the tree. Our scheduling scheme is inherently fault tolerant. If a node in the overlay tree fails, the tree will be restructured to allow other nodes to continue participating in the application. If a task is lost because of a failing node, it will eventually be assigned to another node. However, in the case of communication between tasks, such as in Cannon's algorithm, it is necessary for the application developer to write application-specific code to recover from a failed node and to reestablish the communication overlay network.

In the near future we plan to harness the computing power of idle machines by running the agent platform inside a screen saver. Since computing resources can become unavailable (e.g., if a user wiggles the mouse to terminate the screen saver), we are planning to extend our scheduling cost functions appropriately to allow agents to migrate a running computation, while continuing the communication with other agents.

We are also planning to investigate combinations of distributed, zero-knowledge scheduling with more centralized scheduling schemes to improve the performance for parts of the grid with known machine characteristics. Similar as in networking, where decentralized routing table update protocols such as RIP coexist with more centralized protocols such as OSPF, we envision a grid in which a decentralized

scheduler would be used for unpredictable desktop machines, while a centralized scheduler would be used for, say, a Globus host.

The system described here is a reduced scale proof-of-concept implementation. Clearly, our results need to be validated on a large scale system. In addition to a screen saver-based implementation, we are planning the construction of a simulator. Some important aspects of the Organic Grid approach that remain to be investigated are more advanced forms of fault detection and recovery, the dynamic behavior of the system in relation to changes in the underlying system, and the management of the friends lists.

Acknowledgments

This research was done when all authors were at The Ohio State University. It was partially supported by the Ohio Supercomputer Center grants PAS0036-1 and PAS0121-1.

References

[1] D. Abramson, J. Giddy, and L. Kotler, "High performance parametric modeling with Nimrod/G: Killer application for the global grid?" in *Proc. Intl. Parallel and Distributed Processing Symp.*, May 2000, pp. 520–528.

[2] Berkeley Open Infrastructure for Network Computing (BOINC). [Online]. Available: http://boinc.berkeley.edu/

[3] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using AppLeS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 369–382, 2003.

[4] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, Santa Fe Institute Studies in the Sciences of Complexity, 1999.

[5] J. Bradshaw, N. Suri, A. J. Cañas, R. Davis, K. M. Ford, R. R. Hoffman, R. Jeffers, and T. Reichherzer, "Terraforming cyberspace," *Computer*, vol. 34, no. 7, pp. 48–56, July 2001.

[6] D. Buaklee, G. Tracy, M. K. Vernon, and S. Wright, "Near-optimal adaptive control of a large grid application," in *Proceedings of the International Conference on Supercomputing*, June 2002, pp. 315–326.

[7] L. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University, 1969.

[8] A. Cerpa and D. Estrin, "ASCENT: Adaptive self-configuring sEnsor networks topologies," *IEEE Transactions on Mobile Computing*, vol. 3, no. 3, pp. 272–285, 2004.

[9] A. J. Chakravarti, G. Baumgartner, and M. Lauria, "Application-specific scheduling for the Organic Grid," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2 004)*, Pittsburgh, November 2004, pp. 146–155.

[10] ——, "The Organic Grid: Self-organizing computation on a peer-to-peer network," in *Proceedings of the International Conference on Autonomic Computing*. IEEE Computer Society, May 2004, pp. 96–103.

[11] A. J. Chakravarti, X. Wang, J. O. Hallstrom, and G. Baumgartner, "Implementation of strong mobility for multi-threaded agents in Java," in *Proceedings of the International Conference on Parallel Processing*. IEEE Computer Society, Oct. 2003, pp. 321–330.

[12] ——, "Implementation of strong mobility for multi-threaded agents in Java," Dept. of Computer and Information Science, The Ohio State University, Tech. Rep. OSU-CISRC-2/03-TR06, Feb. 2003.

[13] A. A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: architecture and performance of an enterprise desktop grid system," *J. Parallel and Distributed Computing*, vol. 63, no. 5, pp. 597–610, 2003.

[14] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna, "Analyzing mobile code languages," in *Mobile Object Systems: Towards the Programmable Internet*, ser. Lecture Notes in Computer Science, J. Vitek, Ed., no. 1222. Springer-Verlag, 1996, pp. 93–110. [Online]. Available: http://www.polito.it/~picco/papers/ecoop96.ps.gz

[15] folding@home. [Online]. Available: http://folding.stanford.edu

[16] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," in *Proc. IEEE Symp. on High Performance Distributed Computing (HPDC)*, San Francisco, CA, August 2001, pp. 7–9.

[17] R. Ghanea-Hercock, J. Collis, and D. Ndumu, "Co-operating mobile agents for distributed parallel processing," in *Third International Conference on Autonomous Agents AA99*. Mineapolis, MN: ACM Press, May 1999.

[18] A. Gierer and H. Meinhardt, "A theory of biological pattern formation," *Kybernetik*, vol. 12, pp. 30–39, 1972.

[19] Gnutella. [Online]. Available: http://www.gnutella.com

[20] A. S. Grimshaw and W. A. Wulf, "The Legion vision of a worldwide virtual computer," *Comm. of the ACM*, vol. 40, no. 1, pp. 39–45, Jan. 1997.

[21] E. Heymann, M. A. Senar, E. Luque, and M. Livny, "Adaptive scheduling for master-worker applications on the computational grid," in *Proc. of the First Intl. Workshop on Grid Computing*, 2000, pp. 214–227.

[22] H. James, K. Hawick, and P. Coddington, "Scheduling independent tasks on metacomputing systems," in *Proceedings of Parallel and Distributed Computing Systems*, Aug. 1999.

[23] J. A. H. John F. Shoch, "The "Worm" programs — early experience with a distributed computation," *Comm. of the ACM*, vol. 25, no. 3, pp. 172–180, Mar. 1982.

[24] N. T. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A grid-enabled implementation of the message passing interface," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551–563, 2003.

[25] T. Kindberg, A. Sahiner, and Y. Paker, "Adaptive Parallelism under Equus," in *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, Mar. 1994, pp. 172–184.

[26] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante, "Autonomous protocols for bandwidth-centric scheduling of independent-task applications," in *Proceedings of the International Parallel and Distributed Processing Symposium*, Apr. 2003, pp. 23–25.

[27] D. B. Lange and M. Oshima, *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley, 1998.

[28] ——, "Seven good reasons for mobile agents," *Communications of the ACM*, vol. 42, no. 3, pp. 88–89, Mar. 1999.

[29] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environment," in *Proc. Heterogeneous Computing Workshop*, Apr. 1995, pp. 30–34.

[30] M. Litzkow, M. Livny, and M. Mutka, "Condor — a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988, pp. 104–111.

[31] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Proceedings of the 8th Heterogeneous Computing Workshop*, Apr. 1999, pp. 30–44.

[32] M. Mamei and F. Zambonelli, "Co-Fields: a Physically Inspired Approach to Distributed Motion Coordination," *IEEE Pervasive Computing*, vol. 3, no. 2, April 2004.

[33] A. Montresor, H. Meling, and O. Babaoglu, "Messor: Load-balancing through a swarm of autonomous agents," in *Proceedings of 1st Workshop on Agent and Peer-to-Peer Systems*, ser. Lecture Notes in Artificial Intelligence, no. 2530. Springer-Verlag, July 2002, pp. 125–137.

[34] B. Overeinder, N. Wijngaards, M. van Steen, and F. Brazier, "Multi-agent support for Internet-scale Grid management," in *AISB'02 Symposium on AI and Grid Computing*, O. Rana and M. Schroeder, Eds., April 2002, pp. 18–22.

[35] C. Prehofer and C. Bettstetter, "Self-Organization in Communication Networks: Principles and Design Paradigms," *IEEE Communications Magazine*, vol. 43, no. 7, pp. 78–85, July 2005.

[36] O. Rana and D. Walker, "The Agent Grid: Agent-based resource integration in PSEs," in *16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, Switzerland, August 2000.

[37] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proceedings of ACM SIGCOMM'01*, 2001, pp. 161–172.

[38] J. Santoso, G. D. van Albada, B. A. A. Nazief, and P. M. A. Sloot, "Hierarchical job scheduling for clusters of workstations," in *Proc. Conf. Advanced School for Computing and Imaging*, June 2000, pp. 99–105.

[39] SETI@home. [Online]. Available: http://setiathome.ssl.berkeley.edu

[40] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, San Diego, CA, 2001, pp. 149–160.

[41] I. Taylor, M. Shields, and I. Wang, *Grid Resource Management*. Kluwer, June 2003, ch. 1 - Resource Management of Triana P2P Services.

[42] G. Theraulaz, E. Bonabeau, S. C. Nicolis, R. V. Sol, V. Fourcassi, S. Blanco, R. Fournier, J.-L. Joly, P. Fernndez, A. Grimal, P. Dalle, and J.-L. Deneubourg, "Spatial patterns in ant colonies," *PNAS*, vol. 99, no. 15, pp. 9645–9649, 2002.

[43] A. Turing, "The chemical basis of morphogenesis," *Philos. Trans. R. Soc. London*, vol. 237, no. B, pp. 37–72, 1952.

[44] United Devices, "Grid computing solutions." [Online]. Available: http://www.ud.com

[45] R. Wolski, J. Plank, J. Brevik, and T. Bryan, "Analyzing market-based resource allocation strategies for the computational grid," *Intl. J. of High-performance Computing Applications*, vol. 15, no. 3, pp. 258–281, 2001.

[46] G. Woltman, "GIMPS: The great internet mersenne prime search." [Online]. Available: http://www.mersenne.org/prime.htm

[47] J. Wu and I. Stojmenovic, "Ad Hoc Networks," *IEEE Computer*, vol. 37, no. 2, pp. 29–13, February 2004.