# Self-Organizing Scheduling on the Organic Grid

Arjav J. Chakravarti*
The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760–2098, USA
Arjav.Chakravarti@mathworks.com

Gerald Baumgartner
Dept. of Computer Science
Louisiana State University
298 Coates Hall
Baton Rouge, LA 70803, USA
gb@csc.lsu.edu

Mario Lauria
Dept. of Computer Sci. and Eng.
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210–1277, USA
lauria@cse.ohio-state.edu

***Abstract*— 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.**

**In this paper 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 the Cannon's algorithm for matrix multiplication. The first is an example of 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.**

## I. 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 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 systems state is available to a single entity (the metascheduler) [1]–[4]. 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 [5]–[16]. 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, 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

*The work was done while pursuing a Ph.D. at The Ohio State University.

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 method 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. [17] 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 [18] and peer-to-peer file sharing networks [19], [20].

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 colonizes 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 that is the object of this paper is provided as a library that a developer may adapt for his/her purposes.

Computation organizes itself on the available nodes according to a pattern that emerges from agent-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 NCBI BLAST code for sequence alignment, and Cannon's algorithm for matrix multiplication.

The first is an example of 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.

The main contribution of this paper is the demonstration of 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.

## II. Background and Related Work

### A. Peer-to-Peer and Internet Computing

The goal of utilizing the CPU cycles of idle machines was first realized by the Worm project [21] at Xerox PARC. Further progress was made by academic projects such as Condor [11]. The growth of the Internet made large-scale efforts like GIMPS [6], SETI@home [7] and folding@home [9] feasible. Recently, commercial solutions such as Entropia [10] and United Devices [22] 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.

### B. Scheduling

Decentralized scheduling has recently attracted considerable attention. Two-level scheduling schemes have been considered [23], [24], but these are not scalable enough for the Internet. In the scheduling heuristic described by Leangsuksun et al. [25], 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 [26]. 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 specifically 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 [18]. 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.

### C. 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 [27], [28]. 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, certain complex behaviors of social insects such as ants and bees have been studied in detail, and their applications to the solution of specific computer science problems has been proposed [17], [29]. 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 was inspired by a particular version of the emergence principle called Local Activation, Long-range Inhibition (LALI) [30]. 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 same probability as close ones. During the course of the computation agents behavior encourages the propagation of computation among well-connected nodes while discouraging the inclusion of distant (i.e. less responsive) agents.

## III. Applications

The first demonstration of our decentralized approach was done using a class of applications that is commonly used in grid scheduling research, namely an *independent task application* (or ITA) [31], The specific application we used was BLAST, a popular sequence alignment tool.

In order to demonstrate the generality of the autonomic approach and the flexibility of the Organic Grid scheduling framework, for our second set of experiments we then selected an application at the opposite end of the spectrum, characterized by a highly regular and synchronous pattern of communication — Cannon's matrix multiplication algorithm [32]. In the following section we will emphasize the differences between these two applications and how the Organic Grid framework can be specialized to accommodate them.

For an ITA, the computation 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 used as the overlay network for making scheduling decisions. The tree is continuously restructured during the execution of the application, such that high-throughput nodes are always near the root.

In general, there could be separate overlay networks: for data distribution, for scheduling, and for communication between subtasks. In the case of an ITA, there is no communication between subtasks while the overlay trees for data distribution and scheduling overlap.

The data distribution and communication overlay networks are entirely application specific. On the other hand, the mechanisms for restructuring the scheduling overlay tree can be adapted to a wide variety of applications. There are two key aspects that determine the scheduling behavior: The cost metric used for measuring the performance of individual nodes determines which nodes are moved up or down the tree, whereas the width of the tree is constrained by resource availability. Both of these aspects are again specific to the application.

We have factored out the scheduling mechanism into an object-oriented framework, which an application can extend by providing application-specific metrics and resource constraints.

Cannon's matrix multiplication algorithm is characterized by a highly regular and synchronous pattern of communication. 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 the Cannon application it was the number of machines that belong to the torus.

## IV. Independent Task Application: Scheduling

### A. Overview

One of the works that inspired our project was the bandwidth-centric protocol proposed by Kreaseck et al. [18], in which a grid computation is organized around a tree-structured overlay network with the origin of the tasks at the root. A tree overlay network represents a natural and intuitive way of distributing tasks and collecting results. The drawback of the original scheme is that the performance and the degree of utilization of the system depend entirely on the initial assignment of the overlay network.

In contrast, we have developed our systems to be adaptive in the absence of any knowledge about machine configurations, connection bandwidths, network topology, and assuming only a minimal amount of initial information. While our scheme is also based on a tree, our overlay network keeps changing to adapt to system conditions. Our tree adaptation mechanism is driven by the perceived performance of a node's children, measured passively as part of the ongoing computation [33]. From the point of view of network topology, our system starts with a small amount of knowledge in the form of a *friends list*, and then keeps building its own overlay network on the fly. Information from each node's friends list is shared with other nodes so the initial configuration of the lists is not critical. The only assumption we rely upon is that a friends list is available initially on each node to prime the system; solutions for the construction of such lists have been developed in the context of peer-to-peer file-sharing [19], [34] and will not be addressed in this paper.

The Local Activation, Long-range Inhibition (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. In our experiment, distance is based on the perceived throughput which is some function of communication bandwidth and computational throughput. Nodes are initially recruited using the friends list 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 agents' behavior encourages the propagation of computation among well-connected nodes while discouraging the inclusion of distant (i.e., less responsive) agents.

The methodology we followed to design the agent behavior is as follows. We selected a tree-structured overlay network as the desirable pattern of execution. We then empirically determined the simplest behavior that would organize the communication and task distribution among mobile agents according to that pattern. We then augmented the basic behavior in a way that introduced other desirable properties. With the total computation time as the performance metric, every addition to the basic scheme was separately evaluated and its contribution to the total performance quantitatively assessed.

One such property is the continuous monitoring of the performance of the child nodes. We assumed that no knowledge is initially available on the system, instead passive feedback from child nodes is used to measure their effective performance, e.g., the product of computational speed and communication bandwidth.

Another property is continuous, on-the-fly adaptation using the restructuring algorithm presented in Section IV-D. Basically, the overlay tree is incrementally restructured while the computation is in progress by pushing fast nodes up towards the root of the tree. Other functions that were found to be critical for performance were the automatic determination of parameters such as prefetching and task size, the detection of cycles, the detection of dead nodes and the end of the computation.

### *B. Basic Agent Design*

A large computational task is encapsulated in a strongly mobile agent [35]. This task should be divisible into a number of independent subtasks. A user starts the computation agent on his/her machine. One thread of the agent begins executing subtasks sequentially. The agent is also prepared to receive requests for work from other machines. If the machine has any uncomputed subtasks, and receives a request for work from another machine, it sends a clone of itself to the requesting machine. The requester is now this machine's *child*.

The clone asks its parent for a certain number of subtasks to work on, $s$. A thread begins to compute the subtasks. Other threads are created — when required — to communicate with the parent or other machines. When work requests are received, the agent 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.

An agent requests its parent for more work when it has executed its own subtasks. 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 to its own parent. Every time a node of the tree obtains $r$ results, either computed by itself or obtained from a child, it sends them to its parent. This message includes a request for all pending subtasks.

### *C. Maintenance of Child-lists*

Each node has up to $c$ active children, and up to $p$ potential children. Ideally, $c+p$ is chosen so as to strike a balance between a tree that is too deep (long delays in data propagation) and one that is too wide (inefficient distribution of data).

The active children are ranked on the basis of their performance. The performance metric is application-dependent. For an ITA, a child is evaluated on the basis of the rate at which it sends in results. When a child sends $r$ results, the node measures the time-interval since the last time it sent $r$ results. The final result-rate of this child is calculated as an average of the last $R$ such time-intervals. This ranking is a reflection of the performance of not just a child, but of the entire subtree with the child node at its root.

Potential children are the ones which the current node has not yet been able to evaluate. A potential child is added to the active child-list once it has sent enough results to the current node. If the node now has more than $c$ children, the slowest child, $sc$, is removed from the child-list. As described below, $sc$ is then given a list of other nodes, which it can contact to try and get back into the tree. The current node keeps a record of the last $o$ former children, and $sc$ is now placed in this list. Nodes are purged from this list once a sufficient, user-defined time period elapses. During that interval of time, messages from $sc$ will be ignored. This avoids thrashing and excessive dynamism in the tree.

### *D. Restructuring of the Overlay Network*

The topology of the overlay network is a tree, and it is desirable for the best-performing nodes to be close to the root. In the case of an ITA, both computational speed and link bandwidth contribute to a node's effective performance. Having well connected nodes close to the top enhances the extraction of subtasks from the root and minimizes the communication delay between the root and the best nodes. Therefore the overlay network is constantly being restructured so that the nodes with the highest throughput migrate toward the root, pushing those with low throughput towards the leaves [33].

A node periodically informs its parent about its best-performing child. The parent 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 instructs its child to contact its grandparent directly. If the contact ends in a promotion, the entire subtree rooted at the child node will move one level higher in the tree. This constant restructuring results in fast nodes percolating towards the root of the tree. The checking of a promising child against a list of former children prevents the occurrence of thrashing due

to consecutive promotions and demotions of the same node.

When a node updates its child-list and decides to remove its slowest child, $sc$, it does not simply discard the child. It prepares a list of its children in descending order of performance, i.e., slowest node first. The list is sent to $sc$, which attempts to contact those nodes in turn. Since the first nodes that are contacted are the slower ones, the tree is sought to be kept balanced.

### E. Fault Tolerance

If the parent of a node were to become inaccessible due to machine or link failures, the node and its own descendants would be disconnected from the tree. The application might require that a node remain in the tree at all times. In this scenario, the node must be able to contact its parent's ancestors. Every node keeps a (constant size) list of $a$ of its ancestors. This list is updated every time its 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.

In case of failure (detected by a simple time-out mechanism) a child sends a message to its parent — the $a$-th node in its ancestor-list. If it is unable to contact the parent, it sends a message to the $(a-1)$-st node in that list. This goes on until an ancestor responds to this node's request. The ancestor becomes the parent of the current node and normal operation resumes.

If a node's ancestor-list goes down to size 0, it attempts to obtain the address of some other agent by checking its data distribution and communication overlays. If these are the same as the scheduling tree, the node has no means of obtaining any more work to do. The mobile agent informs the agent environment that no useful work is being done by this machine, before self-destructing. The environment begins to send out requests for work to a list of friends.

In order to recover from the loss of tasks by failing nodes, every node keeps track of unfinished subtasks that were sent to children. If a child requests additional work and no new task can be obtained from the parent, unfinished tasks are handed out again.

## V. Independent Task Application: Measurements

We have conducted experiments to evaluate the performance of each aspect of our scheduling scheme. The experiments were performed on a cluster of eighteen heterogeneous machines at different locations around Ohio. The machines ran the `Aglets` weak mobility agent environment on top of either Linux or Solaris.

TABLE I
Original Parameters

| Parameter Name | Parameter Value |
|---|---|
| Maximum children | 5 |
| Maximum potential children | 5 |
| Result-burst size | 3 |
| Self-adjustment | linear |
| Number of subtasks initially requested | 1 |
| Child-propagation | On |

The application we used to test our system was the gene sequence similarity search tool, NCBI's nucleotide-nucleotide BLAST [36] — a representative independent-task application. The mobile agents started up a BLAST executable to perform the actual computation. The task was to match a 256KB sequence against 320 data chunks, each of size 512KB. Each subtask was to match the sequence against one chunk. Chunks flow down the overlay tree whereas results flow up to the root. An agent cannot migrate during the execution of the BLAST code; since our experiments do not require strong mobility, this limitation is irrelevant to our measurements.

All eighteen machines would have offered good performance as they all had fast connections to the Internet, high processor speeds and large memories. In order to obtain more heterogeneity in their performance, we introduced delays in the application code so that we could simulate the effect of slower machines and slower network connections. We divided the machines into fast, medium and slow categories by introducing delays in the application code.

As shown in Figure 1, the nodes were initially organized randomly. The dotted arrows indicate the directions in which request messages for work were sent to friends. The only thing a machine knew about a friend was its `URL`. We ran the computation with the parameters described in Table I. Linear self-adjustment means that the increasing and decreasing functions of the number of subtasks requested at each node are linear. The time required for the code and the first subtask to arrive at the different nodes can be seen in Figure 2. This is the same for all the experiments.

### A. Comparison with Knowledge-based Scheme

The purpose of these tests is to evaluate the quality of the configuration which is autonomously determined by our scheme for different initial conditions.

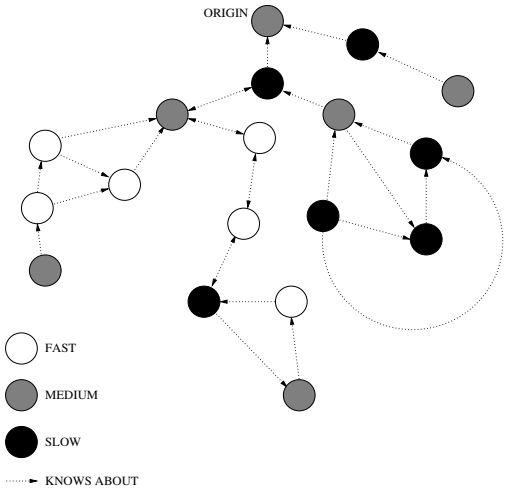


Fig. 1. Random Configuration of Machines

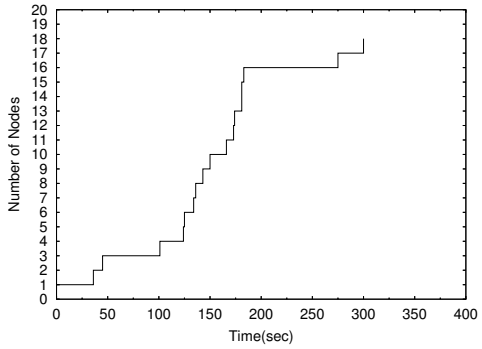


Fig. 2. Code Ramp-up

Two experiments were conducted using the parameters in Table I. In the first, we manually created a good initial configuration assuming a priori knowledge of system parameters. We then ran the application, and verified that the final configuration did not substantially depart from the initial one. We consider a good configuration to be one in which fast nodes are nearer the root. The final tree configuration shows that fast nodes are kept near the root and that the system is constantly re-evaluating every node for possible relocation (as shown by the three rightmost children which are under evaluation by the root).

We began the second experiment with the completely random configuration shown in Figure 1. The resulting configuration shown in Figure 3 is substantially similar to the good configurations of the previous experiment; if the execution time had been longer, the migration towards the root of the two fast nodes at depths 2 and 3 would have been complete.

TABLE II
EFFECT OF PRIOR KNOWLEDGE

| Configuration | Running Time (sec) |
|---|---|
| original | 2294 |
| good | 1781 |

TABLE III
EFFECT OF CHILD PROPAGATION

| Scheme | Running Time (sec) |
|---|---|
| With | 2294 |
| Without | 3035 |

### B. Effect of Child Propagation

We performed our computation with the child-propagation aspect of the scheduling scheme disabled. Comparisons of the running times and topologies are in Table III and Figures 3 and 4. The child-propagation mechanism results in a 32% improvement in the running time. The reason for this improvement is the difference in the topologies. With child-propagation turned on, the best-performing nodes are closer to the root. Subtasks and results travel to and from these nodes at a faster rate, thus improving system throughput and preventing the root from becoming a bottleneck. This mechanism is the most effective aspect of our scheduling scheme.

### C. Number of Children

We experimented with different child-list sizes and found that the data ramp-up time with the maximum number of children set to 5 was less than that with the maximum number of children set to 10 or 20. These results are in Table IV. The root is able to take on more children in the latter cases and the spread of subtasks to nodes that were originally far from the root takes less time.

Instead of exhibiting better performance, the runs where large numbers of children were allowed, had approximately the same total running time as the run with the maximum number of children set to 5. This is because children have to wait for a longer time for their requests to be satisfied.

In order to obtain a better idea of the effect of several children waiting for their requests to be satisfied, we ran two experiments: one with the good initial configuration and the other using a star topology — every non-root node was adjacent to the root at the beginning of the experiment itself. The maximum sizes of the child-lists

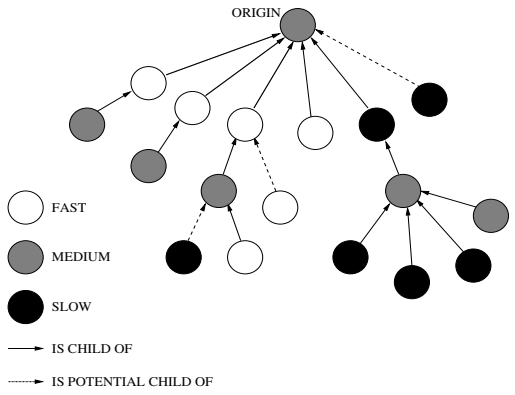


Fig. 3. Final Node Organization, Result-burst size=3, With Child Propagation

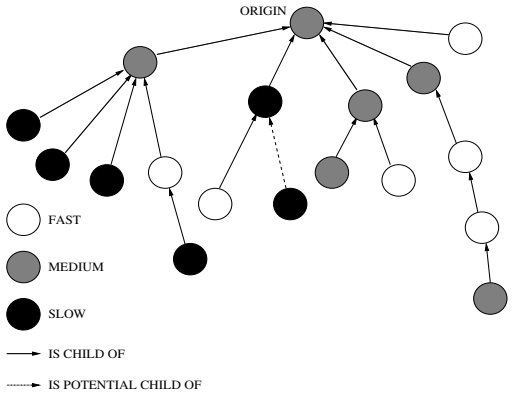


Fig. 4. Final Node Organization, Result-burst size=3, No Child Propagation

were set to 5 and 20, respectively. Since the overlay networks were already organized such that there would be little change in their topology as the computation progressed, there was minimal impact of these changes on the overall running time. The effect of the size of the child-list was then clearly observed as in Table V. Similar results were observed even when the child-propagation mechanisms were turned off.

## VI. Cannon's Algorithm: Scheduling

ITAs easily lend themselves to purely decentralized scheduling over the Organic Grid. However, running Cannon's matrix multiplication algorithm on a desktop grid reveals one point at which centralization is needed: the matrix multiplication stages should begin only after a grid of $k$ ($k = p \times p$) machines is available for computation; a central entity is necessary to count the number of nodes that have been recruited by the computation and to signal the start of the matrix multiplication.

TABLE IV
Effect of No. of Children on Data Ramp-up

| Max. No. of Children | Time (sec) |
|---|---|
| 5 | 1068 |
| 10 | 760 |
| 20 | 778 |

TABLE V
Effect of No. of Children on Running Time

| Max. No. of Children | Time (sec) |
|---|---|
| 5 | 1781 |
| 20 | 2041 |

### A. Overlay Networks

Our systems are designed to operate on large-scale unstructured networks, assuming no knowledge of machine configurations, connection bandwidths, network topology, etc. The only assumption we rely upon is that a friends list is available initially on each node to prime the system; research has been conducted on constructing such lists for peer-to-peer file-sharing [19], [34] and the problem will not be addressed in this paper.

We selected a tree-structured overlay network as the desirable pattern of execution in our previous work [31], [33]. Mobile agents spread out over a desktop grid and formed a tree overlay. The tree restructured itself continuously while computation was in progress, so as to adjust to the performance of the individual nodes and bring high-performance nodes close to the root.

Nodes involved in a Cannon matrix multiplication are organized as a torus. The behavior of the Organic Grid's agents was augmented with matrix multiplication logic, including that for torus formation and maintenance. Of the $n$ nodes recruited by the overlay tree, $k$ were involved in the matrix multiplication. The system thus contained two overlay networks: a tree of all $n$ nodes, and a torus composed of $k$ of these nodes [37].

### B. Basic Implementation

A user decides to use a desktop grid to multiply two matrices, $A$ and $B$, to produce a result matrix, $C$. These matrices may be located at a central location (forming a star), or distributed or replicated across several remote data servers. He/she also decides on the size of tiles the matrices will be divided into. Based on the size of the matrices and tiles, the user can determine the number of machines, $k$, required to multiply the matrices.

The user starts up an agent environment on his/her machine, and creates two agents: a *distribution agent*, which is also the central entity that will signal the beginning of the multiplication, and a computation agent. The computation agent registers with the distribution

agent and obtains a position on the agent grid, before reading one tile each of the $A$ and $B$ matrices from a data server.

Whenever other machines on the desktop grid become idle, they send requests to a list of URLs (friends), where a URL consists of an IP address and port number. If such a request arrives at a machine that is running the computation agent, the agent creates a clone of itself and dispatches the clone to the idle machine. On arrival, the clone also registers with the distribution agent, obtains a position on the agent grid, and reads its own $A$ and $B$ tiles. The topology of the resulting overlay network is a tree with the user's machine at the root node.

When the distribution agent has been contacted by $k$ computation agents, it forms a torus with $p$ machines along each dimension, where $p = \sqrt{k}$. Each computation agent is sent a *start* message to inform it of its left and upper neighbors. These connections to left and upper neighbors form the torus overlay network. Also included in the start messages are the addresses of the nodes to which tiles should be sent during the initialization phase of the algorithm.

Phase one of the algorithm is the initialization phase, where nodes send and receive $A$ and $B$ tiles to and from each other. Different threads within each agent are started up to carry out these operations. As soon as a node has obtained the $A$ and $B$ tiles, it begins phase two to actually multiply the matrices.

### C. Adaptive Tree

Unlike most dedicated clusters, desktop grids could contain a set of heterogeneous machines of varying configurations and performance. The distribution agent will create a torus overlay network of the first $k$ machines it finds. The tree overlay network may spread out to cover a much larger number of nodes, $n$, but only $k$ of them will be part of the torus. The $(n-k)$ extra nodes might include faster machines than those in the torus. The application will benefit from a selection of the $k$ best machines.

Each node has some active children, and some potential children. The active children are ranked on the basis of an application-specific performance metric. The ranking is a reflection of the performance of the entire subtree with the child node at its root. Potential children are those that have not sent any results. If one of them does and performs better than an active child, it replaces that child in the list of active children.

A node periodically informs its parent about its best-performing child. The parent checks whether the grandchild was its child in the recent past. If not, it is willing to consider the grandchild and makes it a potential child instead. The node then instructs its child to contact its grandparent directly.

In this manner, the tree overlay network dynamically adjusts to changing conditions so as to maximize application performance. Each node continuously receives feedback from its children and attempts to propagate its fastest child up the tree. Slow children, on the other hand, are demoted towards the leaves.

In the case of the Cannon application, tree nodes rank their children on the basis of the time required for the last $t$ tile multiplications. This is a a reasonable metric because we assume that computation dominates communication. The $(n-k)$ tree nodes that are not in the torus carry out dummy tile multiplications so that they can be evaluated by their parents.

### D. Role Reversal

The overlay tree contains $k$ regular nodes that are in the torus, and $(n-k)$ extra nodes. As the tree structure changes dynamically, fast, extra nodes get pushed up the tree. When one of these becomes the parent of a slow, regular node, it recognizes that it should be in the torus instead of its slow child. The parent, $f$, initiates a role reversal with the child, $c$.

At the end of its current tile multiplication stage, $c$ informs the nodes to its right and bottom on the torus that they should contact $f$ in future. $c$ transfers its own tiles to $f$, so that $f$ seamlessly replaces it in the torus. $c$ is now an extra node. Thus, application performance is maximized by including the fastest tree nodes in the torus.

### E. Fault Tolerance

A desktop grid is more prone to failure than a reliable dedicated cluster. We focus on the problem of crash faults in this paper. As mentioned previously, a tree overlay network of $n$ nodes is constructed. $k$ of these are part of a torus, and the remaining $(n-k)$ nodes function as spares.

*1) Fault Tolerance on Tree:* If the parent of a node were to become inaccessible due to machine or link failures, the node and its own descendants would be disconnected from the tree. A node must be able to contact its parent's ancestors if necessary. Every node keeps a list of $a$ of its ancestors. This list is updated every time its parent sends it a message.

A child sends a message to its parent — the $a$-th node in its ancestor-list. If it unable to contact the parent, it

sends a message to the $(a-1)$-th node in that list. This goes on until an ancestor responds to this node's request. The ancestor becomes the parent of the current node and normal operation resumes. If a node's ancestor-list goes down to size 0, the computation agent on that node self-destructs and a stationary agent begins to send out requests for work to a list of friends.

*2) Fault Tolerance on Torus:* Fault tolerance is a much more difficult problem for the torus because a failure will cause the entire distributed computation to stall. We define the requirements of a failure detection and recovery mechanism as: i) detect failure, ii) find replacement node, iii) insert replacement at correct position in torus, iv) provide replacement with the state necessary to continue predecessor's computation, v) provide replacement with the information needed to recompute tiles lost to the crash.

A fundamental aspect of our fault tolerance algorithm for the torus is that every torus node knows who its left neighbor is at all times. Nodes take responsibility for detecting crashes to their immediate left and for replacing the crashed nodes. A crash is detected when one node, $r$, attempts to contact the node to its left and finds that it is unable to do so.

The rest of the system system does not stall while failed nodes are being replaced. Instead, a node will timeout if it has not received $A$ or $B$ tiles from its right or bottom neighbors, and the node will read the necessary tiles directly from a data server.

Spare nodes periodically publish their availability to information servers. $r$ queries one of these servers which responds with the URL of machine $l$. $r$ contacts $l$ and gives it three necessary pieces of information: $l$'s position on the torus, the matrix multiplication stage that $r$ — and hence $l$ — is on, and the URL of $l$'s neighbors. $l$ then reads its $A$ and $B$ tiles from the data servers.

The matrix multiplication stages then proceed as described before. When the stages have been completed, the nodes write their $C$ tiles to the appropriate data server. Nodes that were inserted as replacements now need to compute the state that was lost due to their predecessor's crash. They do this by reading the necessary $A$ and $B$ tiles from the data servers, multiplying the tiles, and writing the complete $C$ tiles back to the repositories.

The failure detection and recovery algorithm makes two assumptions:

- Enough extra nodes are present to act as spares throughout the duration of the computation. The number of failures that the application can tolerate is the same as the number of extra machines: $(n-k)$. Since it is the distribution agent that signals the start of the computation, it is easy for it to postpone this signaling until a large number of extra machines have been recruited by the application. The overlay tree can also keep growing, even after the matrix multiplication has begun. This increases the number of failures that can be tolerated, as well as the probability of the application finding high-performance machines.
- Five of the replacement's new neighbors — to its right, top-right, top, top-left and left — are running when the replacement node is inserted, so that the new node can discover its top and left neighbors before computation proceeds. This restriction may be removed by requiring that each node periodically publish its torus position and URL. This information could be published to multiple servers and even the distribution agent itself. New additions to the torus can query these servers and discover their left and top neighbors. The interval at which this publishing occurs needs to be set carefully so that the time for which the computation stalls is minimized.

## VII. Cannon's Algorithm: Measurements

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, and with a Myrinet interconnect. 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.

### A. Scalability

We performed a scalability evaluation by running the application on various sizes of tori and matrices. The tree adaptation mechanism was disabled in order to eliminate its effect on the experiments. The agent behavior has been described in Table VI.

Tables VIII and IX, and Figure 6 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

TABLE VI

AGENT BEHAVIOR, WITHOUT ADAPTATION

| Parameter Name | Parameter Value |
|---|---|
| Maximum children | 2 |
| Maximum potential children | 2 |
| Feedback from children | Off |
| Child-propagation | Off |

TABLE VII

AGENT BEHAVIOR, WITH ADAPTATION

| Parameter Name | Parameter Value |
|---|---|
| Maximum children | 2 |
| Maximum potential children | 2 |
| Result-burst | Average of last 2 tile multiplications |
| Number of subtasks requested | 0 |
| Child-propagation | On |

TABLE VIII

RUNNING TIME ON 1 AND 4 MACHINES, 16 ROUNDS

| Matrix | Single Agent | | $2 \times 2$ Agent Grid | | |
|---|---|---|---|---|---|
| Size (MB) | Tile (MB) | Time (sec) | Tile (MB) | Time (sec) | Speedup |
| 1 | 1 | 75 | 0.25 | 22 | 3.4 |
| 4 | 4 | 846 | 1 | 225 | 3.8 |
| 16 | 16 | 14029 | 4 | 2535 | 5.5 |

TABLE IX

RUNNING TIME ON 1 AND 16 MACHINES, 16 ROUNDS

| Matrix | Single Agent | | $4 \times 4$ 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 |

with a decrease in the size of the tiles stored at each machine. A better scalability evaluation was carried out by using tiling on single agents as well.

### *B. 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 behavior of each agent was as in Table VII. 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 5. 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 7 and 8 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 children was slower than itself and part of the torus, it initiated a swap of roles. Figure 9 shows the tree before the first swap, with the nodes to be swapped having been circled. The effect of this swap on the torus is shown in Figure 10.

Similarly, the topology of the tree and the torus before and after the last swap are in Figures 11 and 12.

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 XII 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 X.

While the adaptive tree mechanism undoubtedly results in a performance improvement in the presence of high-performance extra nodes, it also introduces some overhead when no such extra nodes are present. Nodes still 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 experiments. The overhead of this mechanism was negligible, as can be seen in Table XIII.

### *C. 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.

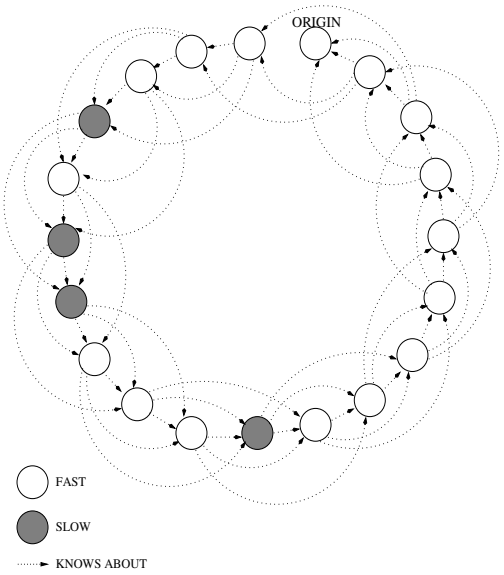


Fig. 5. Original Configuration of Machines

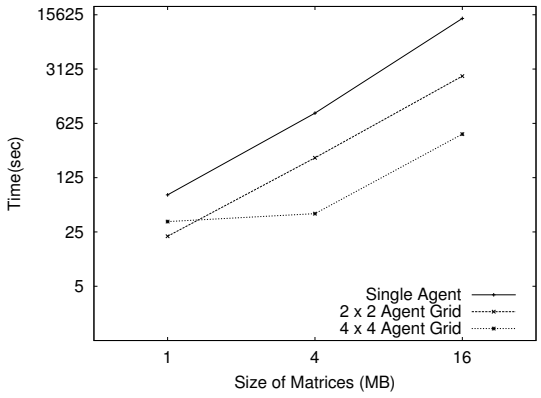


Fig. 6. Running Time on 1, 4 and 16 Machines, 16 Rounds

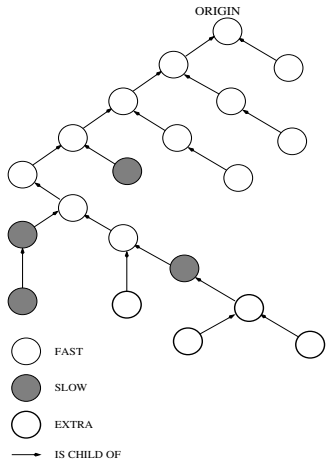


Fig. 7. Original Tree Overlay

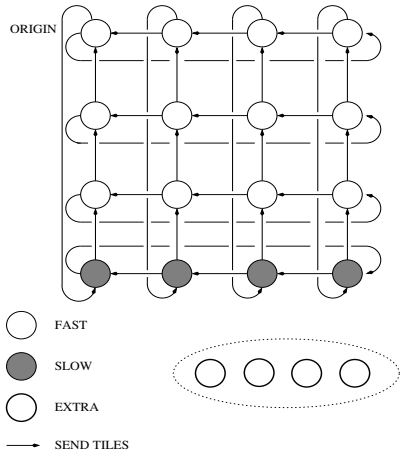


Fig. 8. Original Torus Overlay

The system recovers rapidly from failures on the same column and diagonal, as can be seen in Table XIV. 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.

TABLE X
RUNNING TIME OF 16 ROUNDS ON 4X4 GRID, 16MB MATRIX, 1MB TILES, ADAPTIVE TREE

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

## VIII. 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 applications in which individual nodes need to communicate, a Cannon-style matrix mul-

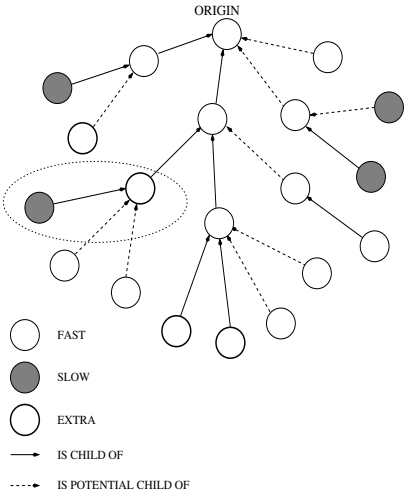


Fig. 9. Tree Overlay Before First Swap

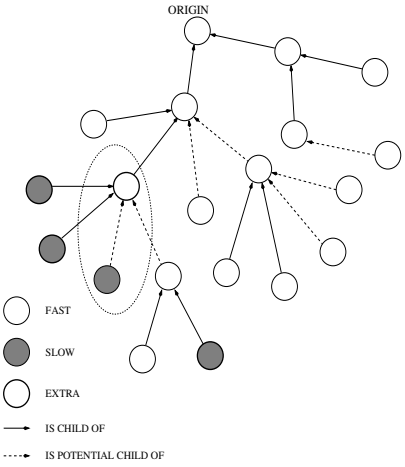


Fig. 11. Tree Overlay Before Fourth Swap

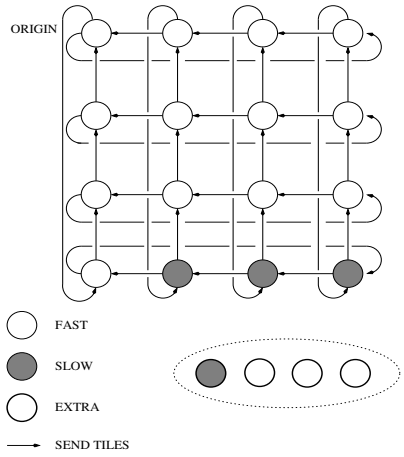


Fig. 10. Torus Overlay After First Swap

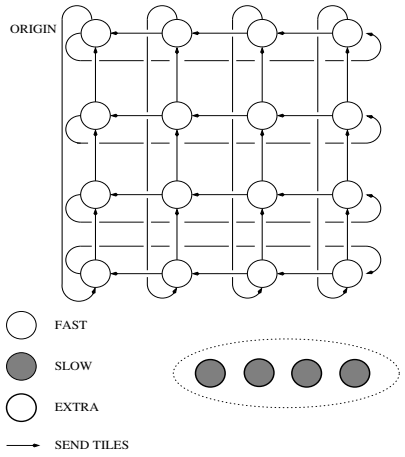


Fig. 12. Torus Overlay After Fourth Swap

TABLE XI
RUNNING TIME OF 16 ROUNDS ON 2X2 GRID, 4MB MATRIX, 1MB TILES, ADAPTIVE TREE

| Slow Nodes | Extra Nodes | Time (sec) |
|---|---|---|
| 2 | 0 | 417 |
| 0 | 0 | 226 |
| 2 | 2 | 343 |

tiplication 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

TABLE XII
PERFORMANCE AT DIFFERENT STAGES OF EXPERIMENT, 4X4 AGENT GRID

| 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 | 14 | 14 |
| 44–47 | - | 13 |
| 48-64 | - | 7 |

TABLE XIII
OVERHEAD OF ADAPTIVE TREE, 16 ROUNDS, 4X4 GRID, 16MB MATRIX, 1MB TILES

| 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 |

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

TABLE XIV
RUNNING TIME OF 16 ROUNDS ON 4X4 GRID, 16MB MATRIX, 1MB TILES

| 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 |

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 centralized schedulers would be used for, say, a Globus host.

The system described in this paper is a small scale proof-of-concept implementation. Clearly our results need to be validated on full scale system. In addition to a screensaver-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 the management of the friends lists.

## ACKNOWLEDGMENTS

This work was partially supported by the Ohio Supercomputer Center grants PAS0036-1 and PAS0121-1.

## REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

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

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

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

[28] A. Gierer and H. Meinhardt, "A theory of biological pattern formation," in *Kybernetik*, no. 12, 1972, pp. 30–39.

[29] 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.

[30] 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," in *PNAS*, vol. 99, no. 15, 2002, pp. 9645–9649.

[31] A. J. Chakravarti, G. Baumgartner, and M. Lauria, "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.

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

[33] A. J. Chakravarti, G. Baumgartner, and M. Lauria, "The Organic Grid: Self-organizing computation on a peer-to-peer network," Dept. of Computer and Information Science, The Ohio State University, Tech. Rep. OSU-CISRC-10/03-TR55, Oct. 2003.

[34] 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.

[35] 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.

[36] Basic Local Alignment Search Tool. [Online]. Available: http://www.ncbi.nlm.nih.gov/BLAST/

[37] A. J. Chakravarti, G. Baumgartner, and M. Lauria, "Application-specific scheduling for the Organic Grid," Dept. of Computer and Information Science, The Ohio State University, Tech. Rep. OSU-CISRC-4/04-TR23, April 2004.