Application-Specific Scheduling for the Organic Grid[*]

Arjav J. Chakravarti Gerald Baumgartner Mario Lauria
Dept. of Computer and Information Science
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210–1277, USA
{arjav,gb,lauria}@cis.ohio-state.edu

Technical Report: OSU-CISRC-4/04-TR23

Abstract

We propose 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. Our approach achieves the following design objectives: near-zero knowledge of network topology, zero knowledge of system status, autonomous scheduling, distributed computation, lack of specialized nodes. Every node is equally responsible for scheduling and computation, both of which are performed with practically no information about the system.

We believe that this model is ideally suited for large-scale unstructured grids such as desktop grids. This model avoids the extensive system knowledge requirements of traditional Grid scheduling approaches. Contrary to the popular master/worker organization of current desktop grids, our approach does not rely on specialized super-servers or on application-specific clients. By encapsulating computation and scheduling behavior into mobile agents, we decouple both application code and scheduling functionality from the underlying infrastructure. The resulting system is one where every node can start a large grid job, and where the computation naturally organizes itself around available resources.

Through the careful design of agent behavior, the resulting global organization of the computation can be customized for different classes of applications. In a previous paper, we described a proof-of-concept prototype for an independent task application. In this paper, we generalize the scheduling framework and demonstrate that our approach is applicable to a computation with a highly synchronous communication pattern, namely Cannon's matrix multiplication.

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

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 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) [14, 2, 1, 29]. 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 [11, 33, 27, 10, 8, 22, 23, 15, 19, 4, 18]. 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 easily replicated in a way that scales with the size of the system.

Current approaches to organizing computation on large systems can be traced to techniques that were first developed in the context of parallel computing using traditional supercomputers. We propose a completely new approach to large scale computations. Our approach is conceptually different in that instead of starting with legacy models of computation and trying to adapt them to large scale systems (bottom-up approach), we propose a computational model designed to work with an arbitrarily large number of entities, and we work our way down (top-bottom approach).

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 task of organizing computation over large complex systems, a large computation must be broken into small self-contained chunks, each capable of expressing autonomous behavior in its interaction with other chunks.

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. The centrality of local behavior in our systems means that only a minimal infrastructure is needed, providing very little functionality beyond sandboxing and detection of idle cycles, contrarily to prevailing current approaches [11].

The notion that complex systems can be organized according to local rules is not new. Montresor et al. [24] 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 [20] and peer-to-peer file sharing networks [13, 28].

1.1 The Big Picture

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, for example the user's laptop, other nodes are called in to contribute resources. Computation organizes itself on the available nodes according to a pattern that emerges from their point-to-point interactions.

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. The important point is that this flexibility is achieved because these patterns are not built into the system, but they emerge from the autonomous behavior of its parts.

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 is provided as a library that is adapted to the application using application-specific scheduling rules.

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 and the way to get it is application dependent and can be personalized. For example for our first (data-intensive) application, both neighbor computing rate and communication bandwidth of the intervening link were important, and this information was obtained using feedback from the ongoing computation.

In summary, our approach relies on the flexibility of designing a behavior that implicitly defines many of the global aspects of a distributed computation, and explicitly defines local tasks. Almost all of the relevant design issue of the computations are solved in code that is part of the application, requiring a very thin, uniform infrastructure. The reliance of the computation on only local interactions defines a system which is inherently adaptive and scalable.

The main contributions of this paper are a generalization of the scheduling mechanisms of the Organic Grid. While previously we studied the scheduling aspects for an independent task application, in this paper we demonstrate that our framework is flexible enough to be applied to an application with a synchronous, mesh-like communication pattern, such as Cannon's matrix multiplication — an application usually considered challenging for running on a grid.

2 Background and Related Work

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 [17] at Xerox PARC. Further progress was made by academic projects such as Condor [22]. The growth of the

Internet made large-scale efforts like GIMPS [33], SETI@home [27] and folding@home [10] feasible. Recently, commercial solutions such as Entropia [8] and United Devices [9] 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.

2.2 Scheduling

Decentralized scheduling is a field that has recently attracted considerable attention. Two-level scheduling schemes have been considered [16, 26], but these are not scalable enough for the Internet. In the scheduling heuristic described by Leangsuksun et al. [21], 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 [32]. 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 [20]. Each node in the system 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.

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 [31, 12]. 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 [24, 3]. 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 the “friends list” 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.

3 Applications

In previous research we have demonstrated how to apply our decentralized approach to organizing computation to a class of applications that is commonly used in grid scheduling research, namely an *independent task application* (or ITA) [7], The specific application we used was BLAST, a popular sequence alignment tool.

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.

In order to demonstrate the generality of the autonomic approach and the flexibility of the Organic Grid scheduling framework, we 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 [5]. 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.

4 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.

4.1 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 [13, 25] 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 [7, 6]. 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.

4.2 Basic Implementation

A user decides to use a desktop grid to multiply two matrices, A and B, to produce a result tile, C. These matrices’ may be distributed or replicated across several remote data servers, or be located on the user’s computer. 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). 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.

As described in Figure 1, a node needs five buffers to carry out its operations. $currentATile$ and $currentBTile$, hold the tiles that are to be multiplied during the current computation stage, while $nextATile$ and $nextBTile$ hold the tiles that were prefetched and will be used during the next stage. The result is stored in $resultCTile$.

```
for all $i = 1 : p - 1$
    $currentATile = nextATile$
    $currentBTile = nextBTile$

    /* five concurrent operations */
    (send $currentATile$ to $leftNeighbour$) || (send $currentBTile$ to $upperNeighbour$) ||
    (receive $nextATile$ from $rightNeighbour$) || (receive $nextBTile$ from $lowerNeighbour$) ||
    (multiply $currentATile$ and $currentBTile$, store result in $resultCTile$)

multiply $currentATile$ and $currentBTile$, store result in $resultCTile$
```

Figure 1. Matrix Multiplication Phase

4.3 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.

4.4 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.

4.5 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.

4.5.1 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.

4.5.2 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.

5 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. 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.

5.1 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 1.

Tables 3 and 4, and Figure 3 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. A better scalability evaluation was

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

Table 1. Agent Behavior, Without 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 2. Agent Behavior, With Adaptation

Matrix	Single Agent		2×2 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 3. Running Time on 1 and 4 Machines, 16 Rounds

carried out by using tiling on single agents as well. These results have been summarized in Tables 5 and 6, and Figures 4 and 5.

5.2 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 2. 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 2. 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 6 and 7 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 8 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 9.

Similarly, the topology of the tree and the torus before and after the remaining swaps are in Figures 10, 11, 12 and 13.

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 9 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

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

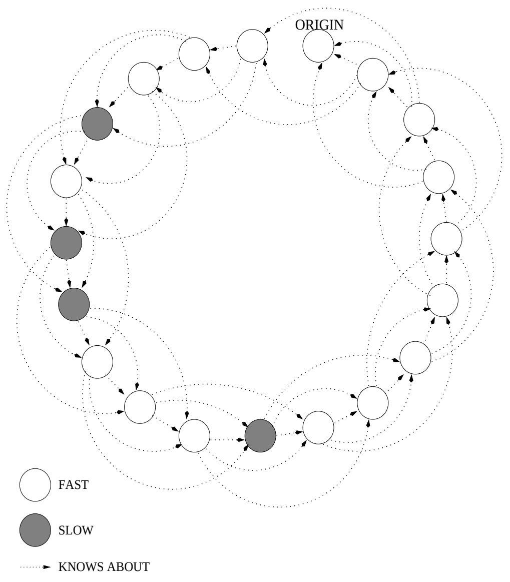

Figure 2. Original Configuration of Machines

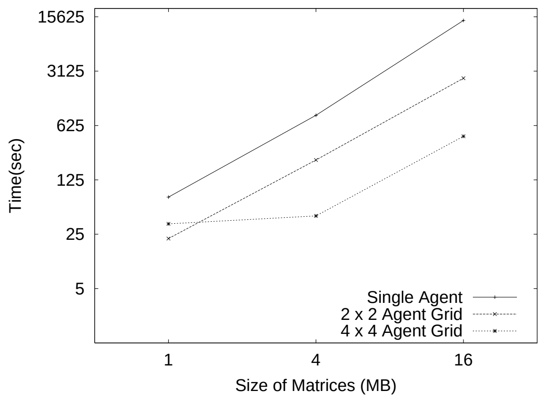

Figure 3. Running Time on 1, 4 and 16 Machines, 16 Rounds

the 16 agents sped up and reached high steady-state performance. The effect of this on overall running time can be seen in Table 7.

Similar results were obtained for an experiment that used a torus with 2 nodes along each dimension. The results are in Table 10.

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 experiments. The overhead of this mechanism was negligible, as can be seen in Tables 11 and 12.

5.3 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.

Matrix	Single Agent		2×2 Agent Grid		
Size (MB)	Tile (MB)	Time (sec)	Tile (MB)	Time (sec)	Speedup
1	0.25	52	0.25	22	2.4
4	1	708	1	225	3.2
16	4	8039	4	2535	3.2

Table 5. Comparison of Running Time on 1 and 4 Machines, 16 Rounds, Tiling for 1 Machine

Matrix	Single Agent		4×4 Agent Grid		
Size (MB)	Tile (MB)	Time (sec)	Tile (MB)	Time (sec)	Speedup
1	0.0625	45	0.0625	34	1.3
4	0.25	425	0.25	43	9.8
16	1	7409	1	454	16.3

Table 6. Comparison of Running Time on 1 and 16 Machines, 16 Rounds, Tiling for 1 Machine

The system recovers rapidly from failures on the same column and diagonal, as can be seen in Table 13. 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.

6 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.

Previously, we had experimented with scheduling a massively parallel application (BLAST) on the Organic Grid [7]. We have demonstrated that our scheduling scheme is also applicable to applications in which individual nodes need to communicate by scheduling 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 the Cannon 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 experimental platform we used was a set of 20 heterogeneous machines across Ohio. In the near future we plan to harness the computing power of idle machines by running the agent platforms inside

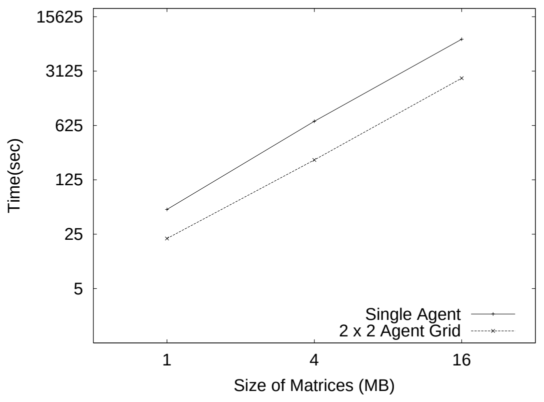

Figure 4. Running Time on 1 and 4 Machines, 16 Rounds, Tiling for 1 Machine

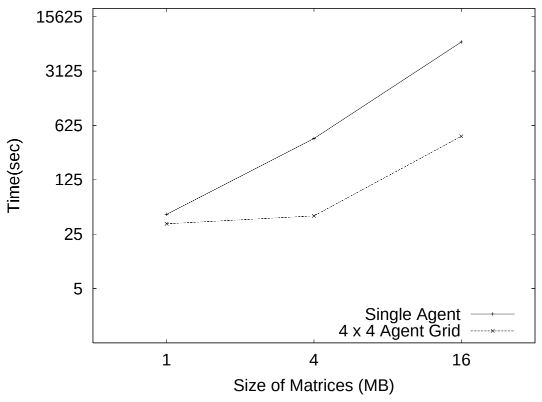

Figure 5. Running Time on 1 and 16 Machines, 16 Rounds, Tiling for 1 Machine

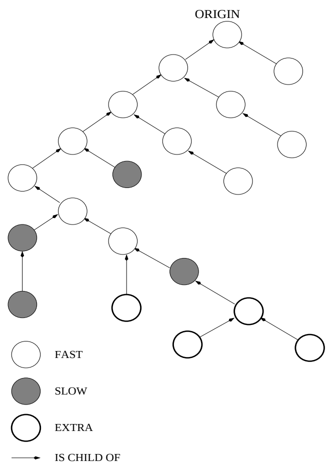

Figure 6. Original Tree Overlay

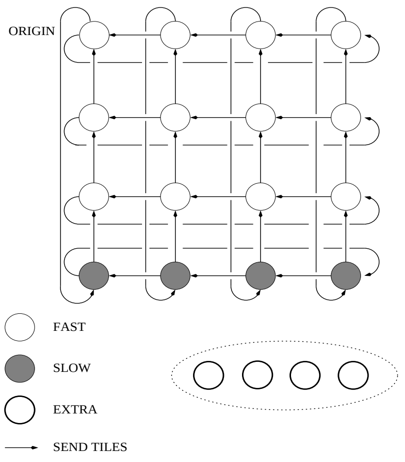

Figure 7. Original Torus Overlay

screen savers. 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.

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.

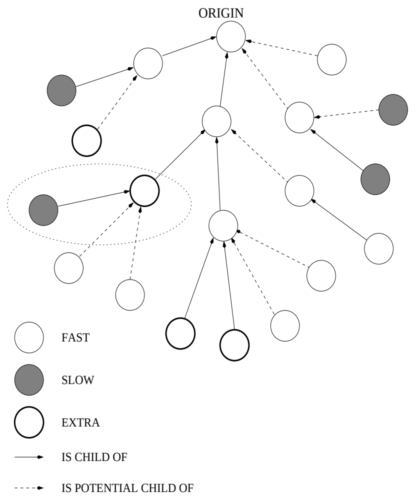

Figure 8. Tree Overlay Before First Swap

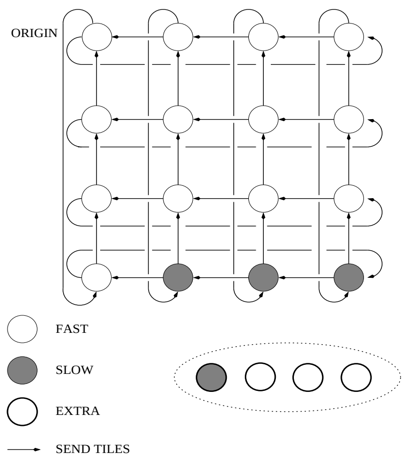

Figure 9. Torus Overlay After First Swap

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

Table 7. Running Time of 16 Rounds on 4x4 Grid, 16MB Matrix, 1MB Tiles, Adaptive Tree

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

Table 8. Running Time of 16 Rounds on 2x2 Grid, 4MB Matrix, 1MB Tiles, Adaptive Tree

References

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *Proceedings of International Parallel and Distributed Processing Symposium*, pages 520–528, May 2000.

[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*, 14(4):369–382, 2003.

[3] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, Santa Fe Institute Studies in the Sciences of Complexity, 1999.

[4] 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.

[5] L. Cannon. *A Cellular Computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[6] Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. The Organic Grid: Self-organizing computation on a peer-to-peer network. Technical Report OSU-CISRC-10/03-TR55, Dept. of Computer and Information Science, The Ohio State University, October 2003.

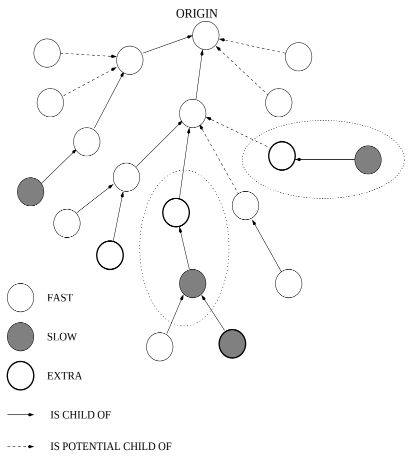

Figure 10. Tree Overlay Before Second And Third Swaps

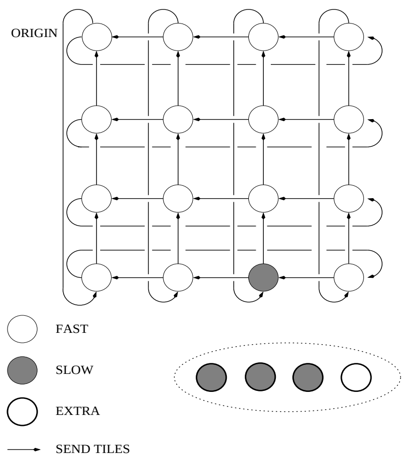

Figure 11. Torus Overlay After Second And Third Swaps

[7] Arjav J. Chakravarti, Gerald Baumgartner, and Mario 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.

[8] Andrew A. Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.

[9] United Devices. http://www.ud.com.

[10] folding@home. http://folding.stanford.edu.

[11] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7–9, San Francisco, California, August 2001.

[12] A. Gierer and H. Meinhardt. A theory of biological pattern formation. In *Kybernetik*, number 12, pages 30–39, 1972.

[13] Gnutella. http://www.gnutella.com.

[14] Andrew S. Grimshaw and W. A. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[15] Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive scheduling for master-worker applications on the computational grid. In *Proceedings of the First International Workshop on Grid Computing*, pages 214–227, 2000.

[16] H. James, K. Hawick, and P. Coddington. Scheduling independent tasks on metacomputing systems. In *Proceedings of Parallel and Distributed Computing Systems*, August 1999.

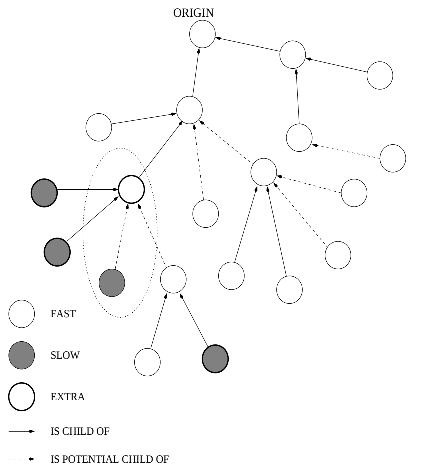

Figure 12. Tree Overlay Before Fourth Swap

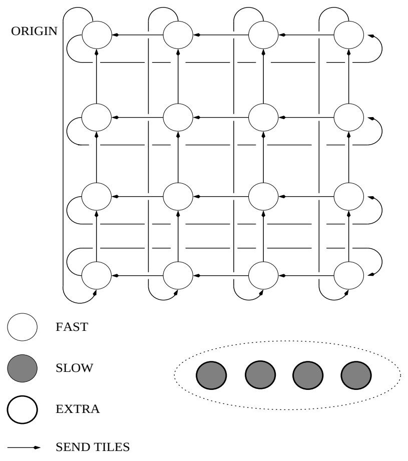

Figure 13. Torus Overlay After Fourth Swap

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 9. Performance at Different Stages of Experiment, 4x4 Agent Grid

Stage	Swap position on Torus	Avg. Tile Mult. Time (sec)
1–5	-	12
6	2	15
7–14	-	13
15	3	14
16–17	-	19
18-32	-	7

Table 10. Performance at Different Stages of Experiment, 2x2 Agent Grid

[17] Jon A. Hupp John F. Shoch. The ”Worm” programs — early experience with a distributed computation. *Communications of the ACM*, 25(3), March 1982.

[18] 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*, 63(5):551–563, 2003.

[19] T. Kindberg, A. Sahiner, and Y. Paker. Adaptive Parallelism under Equus. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, pages 172–184, March 1994.

[20] Barbara Kreaseck, Larry Carter, Henri Casanova, and Jeanne Ferrante. Autonomous protocols for bandwidth-centric scheduling of independent-task applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 23–25, April 2003.

[21] C. Leangsuksun, J. Potter, and S. Scott. Dynamic task mapping algorithms for a distributed heterogeneous computing environment. In *Proceedings of the Heterogeneous Computing Workshop*, pages 30–34, April 1995.

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 11. 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)
2	0	413	2	0	417
0	0	225	0	0	226

Table 12. Overhead of Adaptive Tree, 16 Rounds, 2x2 Grid, 4MB Matrix, 1MB Tiles

[22] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[23] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra A. Hensgen, and Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings of the 8th Heterogeneous Computing Workshop*, pages 30–44, April 1999.

[24] 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*, July 2002.

[25] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM'01*, 2001.

[26] J. Santoso, G. D. van Albada, B. A. A. Nazief, and P. M. A. Sloot. Hierarchical job scheduling for clusters of workstations. In *Proceedings of the 6th annual conference of the Advanced School for Computing and Imaging*, pages 99–105, June 2000.

[27] SETI@home. http://setiathome.ssl.berkeley.edu.

[28] Ion Stoica, Robert Morris, David Karger, M. Francs Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160, San Diego, CA, 2001.

[29] Ian Taylor, Matthew Shields, and Ian Wang. *Grid Resource Management*, chapter 1 - Resource Management of Triana P2P Services. Kluwer, June 2003.

[30] Guy Theraulaz, Eric Bonabeau, Stamatios C. Nicolis, Ricard V. Sol, Vincent Fourcassi, Stphane Blanco, Richard Fournier, Jean-Louis Joly, Pau Fernndez, Anne Grimal, Patrice Dalle, and Jean-Louis Deneubourg. Spatial patterns in ant colonies. In *PNAS*, volume 99, pages 9645–9649, 2002.

[31] A. Turing. The chemical basis of morphogenesis. In *Philos. Trans. R. Soc. London*, volume 237, pages 37–72, 1952.

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 13. Running Time of 16 Rounds on 4x4 Grid, 16MB Matrix, 1MB Tiles

[32] Rich Wolski, James Plank, John Brevik, and Todd Bryan. Analyzing market-based resource allocation strategies for the computational grid. *International Journal of High-performance Computing Applications*, 15(3), 2001.

[33] G. Woltman. http://www.mersenne.org/prime.htm.