1 The Organic Grid: self-organizing computational biology on desktop Grids

ARJAV J. CHAKRAVARTI[†] and GERALD BAUMGARTNER
and MARIO LAURIA

Dept. of Computer Science and Engineering
The Ohio State University
395 Dreese Laboratory,
2015 Neil Avenue
Columbus, OH 43210, USA

Machines take me by surprise with great frequency.

—A. Turing

I think there's a world market for about five computers.

—T. J. Watson

[†]Current address: The MathWorks, Inc., Natick, MA 01760, USA.

Contents

1.1 INTRODUCTION

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

Some of the largest computations in the world have been carried out on collections of PCs and workstations over the Internet. Tera-flop levels of computational power have been achieved by systems composed of heterogeneous computing resources that number in the hundreds-of-thousands to the millions. This extreme form of distributed computing is often called *internet computing*, and has allowed scientists to run applications at unprecedented scales at a comparably modest cost. The desktop-based platforms on which Internet-scale computations are carried out are often referred to as *desktop grids*. In analogy to computational grids [19, 18], these collections of distributed machines are glued together by a layer of middleware software that provides the illusion of a single system [38, 16, 12]. While impressive, these efforts only use a tiny fraction of the desktops connected to the Internet. Order of magnitude improvements could be achieved if novel systems of organization of the computation were to be introduced that overcome the limits of present systems.

A number of large-scale systems are based on variants of the master/workers model [17, 38, 50, 16, 12, 32, 33, 23, 26, 6, 25]. 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 master (especially for data-intensive applications) and by the difficulty of deploying the supporting software on a large number of workers. Since networks cannot be assumed to be reliable, large desktop grids are designed for independent task applications with relatively long-running individual tasks.

By contrast, research on traditional grid scheduling has focused on algorithms to determine an optimal computation schedule based on the assumption that sufficiently

Table 1.1 Classification of Approaches to Large-Scale Computation

	Large Desktop Grids (e.g., BOINC)	Small Desktop Grids (Condor)	Traditional Grids (e.g., Globus)	Organic Grid
Network	large, unreliable	small, reliable	small, reliable	large, unreliable
Task granularity	large	medium to large	medium to large	medium to large
Task model	independent task	any	any	any
Task scheduling	centralized	centralized	centralized	decentralized

detailed and up to date knowledge of the system state is available to a single entity (the metascheduler) [22, 3, 1, 45]. 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.

To summarize, the existing approaches to harnessing machine resources represent different design strategies, as shown in Table 1.1. Traditional grid approaches decide to limit the size of the system and assume a fairly reliable network in exchange for being able to run arbitrary tasks, such as MPI tasks. Desktop grid approaches restrict the type of the application to independent (or nearly independent) tasks of fairly large task granularity in exchange for being able to run on very large numbers of machines with potentially unreliable network connections. The best of both worlds, arbitrary tasks and large numbers of machines, is not possible because the central task scheduler would become a bottleneck.

We present a new approach to grid computing, called the Organic Grid, that does not have the restrictions of either of the existing approaches. By using a decentralized, adaptive scheduling scheme, we attempt to allow arbitrary tasks to be run on large numbers of machines or in conditions with unreliable networks. Our approach can be used to broaden the class of applications that can be run on a large desktop grid, or to extend a traditional grid computing approach to machines with unreliable connections. The tradeoff of our approach is that the distributed scheduling scheme may not result in as good resource usage as with a centralized scheduler.

The Organic Grid project is an effort to redesign from scratch the infrastructure for distributed computation on desktop grids. Our middleware represents a radical departure from current grid or Peer-to-Peer concepts, and does not rely on existing grid technology. In designing our Organic Grid infrastructure we have tried to address the following questions:

- What is the best model of utilization of a system based on the harvesting of idle cycles of hundreds-of-thousands to millions of PCs?
- How should the system be designed in order to make it consistent with the grid computing ideals of computation as a ubiquitous and easily accessible utility?

Nature provides numerous examples of complex systems comprising millions of organisms that organize themselves in an autonomous, adaptive way to produce complex patterns. In these systems, the emergence of complex patterns derives from the superposition of a large number of interactions between organisms that have relatively simple behavior. In order to apply this approach to the task of organizing computation over complex systems such as desktop grids, one would have to devise a way of breaking a large computation into small autonomous chunks, and then endowing each chunk with the appropriate behavior.

Our approach is to encapsulate computation and behavior into mobile agents. A similar concept was first explored by Montresor et al. [34] in a project showing how an ant algorithm could be used to solve the problem of dispersing tasks uniformly over a network. In our approach, the behavior is designed to produce desirable patterns of execution according to current grid engineering principles. More specifically, the pattern of computation resulting from the synthetic behavior of our agents reflects some general concepts about autonomous grid scheduling protocols studied by Kreaseck et al. [27]. Our approach extends previous results by showing i) how the basic concepts can be extended to accommodate highly dynamic systems, and ii) a practical implementation of these concepts.

One consequence of the encapsulation of behavior and computation into agents is that they can be easily customized for different classes of applications. Another desirable consequence is that the underlying support infrastructure for our system is extremely simple. Therefore, our approach naturally lends itself to a true peer-to-peer implementation, where each node can be at the same time provider and user of the computing utility infrastructure. Our scheme can be easily adapted to the case where the source of computation (the node initiating a computing job) is different from the source of the data.

The purpose of this work is the initial exploration of a novel concept, and as such it is not intended to give a quantitative assessment of all aspects and implications of our new approach. In particular, detailed evaluations of scalability, degree of tolerance to faults, adaptivity to rapidly changing systems, or security issues have been left for future studies.

1.2 BACKGROUND AND RELATED WORK

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

1.2.1 Grid Computing

Grid computing is a term often used to indicate some form of distributed computation on geographically distributd resources. We use the term not as referring to a particular technology, which is consistent with the following definition of grid computing given by Ian Foster and Carl Kesselmann:

> *"coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations"*

where virtual organizations (VOs) are defined as "dynamic collections of individuals, institutions, and resources" [19]. Examples of VOs are members of an industrial consortium bidding on a new aircraft; participants of a large, international, multi-year high-energy physics collaboration; peer-to-peer computing (as implemented, for example, in the Napster, Gnutella, and Freenet file sharing systems) and Internet computing (as implemented, for example by the SETI@home, Parabon, and Entropia systems) [19].

As these examples show, VOs encompass entities with very different characteristics. Nevertheless, in order to achieve the grid computing goal the following set of common requirements can be identified [19]:

- mechanisms for sharing of varied resources, ranging from programs, files, and data to computers, sensors, and networks;
- a need for highly flexible sharing relationships, ranging from client-server to peer-to-peer;
- a need for sophisticated and precise levels of control over how shared resources are used, including fine-grained and multi-stakeholder access control, delegation, and application of local and global policies;
- mechanisms for diverse usage models, ranging from single user to multi-user and from performance sensitive to cost-sensitive and hence embracing issues of quality of service, scheduling, co-allocation, and accounting.

Currently, the Globus toolkit [37] (both in its initial version and in its recent redesign following the web services architecture known as OGSA [18]) represents the de facto standard model for grid applications development and deployment. The Globus toolkit is a collection of tools for grid application development, each targeted to a particular area of Grid computing (application scheduling, security management, resource monitoring, etc.). An unintended consequence of its popularity is the common misconception that grid technology and Globus (or OGSA) are one and the same thing. In reality the grid concept as defined above is much more general, and a lot of research opportunities exist on the study of novel forms of grid computing.

Peer-to-peer and Internet computing are examples of the more general "beyond client-server" sharing modalities and computational structures that we referred to in our characterization of VOs. As such, they have much in common with grid technologies. In practice, however, so far there has been very little work done at the intersections of these domains.

The Organic Grid is a novel type of grid that is based on a peer-to-peer model of interaction, where the code (in addition to the data) is exchanged between peers. It therefore requires some type of mobility of the code. We use mobile agents as the underlying technology for building our grid infrastructure. In peer-to-peer systems like Napster or Kazaa, the peers exchange files. We demonstrate that by connecting agent platforms into a peer-to-peer network and by exchanging computation in the form of agents, it is possible to build a novel grid computing infrastructure.

At a very high level of abstraction, the Organic Grid is similar to Globus in that a user can submit an application to the system and the application will be executed somewhere on the grid. The innovative approach employed by the Organic Grid is that the user's application is encapsulated in a mobile agent together with scheduling code; the mobile agent then carries the application to a machine that has declared it has available computing resources. Instead of a centralized scheduler as in existing Grid approaches, agents make simple decentralized scheduling decisions on their own.

A grid architecture like ours with decentralized scheduling could have been built without employing mobile agents, using, for example, a service architecture instead (after all, mobile agents are implemented on top of remote method invocation, which is implemented on top of client-server communication). However, mobile agents provide a particularly convenient abstraction that make building such a system, and experimenting with different scheduling strategies, much easier and more flexible. In addition, the use of strong mobility will enable future implementations of the Organic Grid to transparently checkpoint and migrate user applications.

1.2.2 Peer-to-Peer and Internet Computing

The goal of utilizing the CPU cycles of idle machines was first realized by the Worm project [44] at Xerox PARC. Further progress was made by academic projects such as Condor [32]. The growth of the Internet made large-scale efforts like GIMPS [50], SETI@home [38] and folding@home [16] feasible. Recently, commercial solutions such as Entropia [12] and United Devices [14] have also been developed.

Peer-to-peer computing adopts a highly decentralized approach to resource sharing in which every node in the system can assume the role of client or server [35, 43].

Current peer-to-peer systems often rely on highly available central servers and are mainly used for high profile, large scale computational projects such as SETI@home, or for popular data-centric applications like file-sharing [15].

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

1.2.3 Scheduling

Decentralized scheduling is a field that has recently attracted considerable attention. Two-level scheduling schemes have been considered [24, 42], but these are not scalable enough for the Internet. In the scheduling heuristic described by Leangsuksun et al. [30], 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 this information will be available in large-scale and dynamic environments.

G-Commerce was a study of dynamic resource allocation on a grid in terms of computational market economies in which applications must buy resources at a market price influenced by demand [49]. 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 [27]. 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.

1.2.4 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 [47, 21]. 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 [34, 4]. 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), which was recently shown to be responsible for the formation of a complex pattern using a clever experiment on ants [46].

1.2.5 Strongly Mobile Agents

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

Mobile agents [29] have relocation autonomy. These agents offer a flexible means of distributing data and code around a network, of dynamically moving between hosts as resource availability varies, and of carrying multiple threads of execution to simultaneously perform computation, decentralized scheduling, and communication with other agents. There have been some previous attempts to use mobile agents for grid computing or distributed computing [5, 39, 36, 20].

The majority of the mobile agent systems that have been developed until now are Java-based. However, the execution model of the Java Virtual Machine does not permit an agent to access its execution state, which is why Java-based mobility libraries can only provide *weak mobility* [13]. Weakly mobile agent systems, such as IBM's Aglets framework [28] do not migrate the execution state of methods. The `go()` method, used to move an agent from one virtual machine to another, simply does not return. When an agent moves to a new location, the threads currently executing in it are killed without saving their state. The lifeless agent is then shipped to its destination and restarted there. Weak mobility forces programmers to use a difficult programming style, i.e., the use of callback methods, to account for the absence of migration transparency.

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

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

We provide strong and forced mobility for the full Java programming language by using a preprocessor that translates an extension of Java with strong mobility into weakly mobile Java code that explicitly maintains the execution state for all threads as a mobile data structure [10, 11]. For the target weakly mobile code we currently use IBM's Aglets framework [28]. The generated weakly mobile code maintains a movable execution state for each thread at all times.

1.3 AUTONOMIC SCHEDULING

1.3.1 Overview

One of the works that inspired our project was the bandwidth-centric protocol proposed by Kreaseck et al. [27], 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 [7]. 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 [15, 40] and will not be addressed in this paper.

To make the tree adaptive we rely on a particular version of the emergence principle called Local Activation, Long-range Inhibition (LALI). 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 with the following modification: 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 a 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 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.

In the natural phenomena, the emergence principle is often invoked to explain the formation of patterns. In the Organic Grid it is possible to define the physical pattern of a computation as the physical allocation of the agents to computing resources such as communication links, computing nodes, data repositories. Two examples of computation patterns on a grid are illustrated in Figure 1.1, where nodes are represented as dots (double circle is the starting node), links as lines of thickness proportional to the bandwidth. Figure 1.1(a) represents a pattern that could be obtained by recruiting nodes at random. A more desirable pattern would be one where the availability of resources is taken into account in deciding where to run agents. For example in a data intensive, indipendent task application such as BLAST [2] a factor of merit should be not only the processing power of a node but also the available bandwidth of the intervening link. Assuming for semplicity that nodes are homogeneous, a desirable pattern of computation is one where nodes connect to high bandwidth links are eventually more likely to be retained in the computation, as in Figure 1.1(b).

The main challenge of programming the Organic Grid is designing an agent behavior that produces a desirable physical pattern of computation. The agent design we adopted for runnig BLAST promotes the formation of a physical pattern that approximates the one shown in Figure 1.1(b). It is important to note that in general the criteria defining what is a desirable pattern are specific for each class of application, and therefore the agent design must be taylored to the requirement of each class.

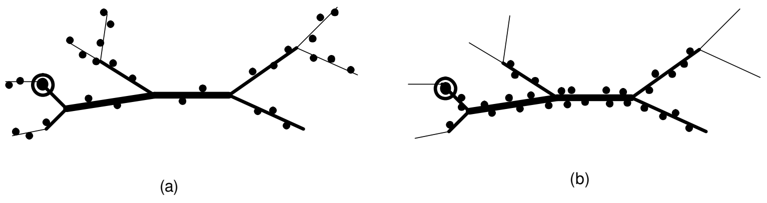


Fig. 1.1 Example of physical pattern of computation on a network; nodes are represented as dots (double circle is starting node), links as lines of thickness proportional to the bandwidth. (a) A configuration obtained by recruiting nodes at random, (b) a desirable pattern showing clustering of the computation around available resources by selective recruitment and dropping of nodes.

The methodology we followed to design the agent behavior is as follows. We selected a tree-structured overlay network as the fundamental logical pattern around which to organize the computation. We then empirically determined the simplest agent behavior that would i) generate the tree overlay, and ii) organize basic tasks such as agent-to-agent communication and task distribution according to such tree 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 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 1.3.4. 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.

In this paper we focus on the solution to one particular problem: the scheduling of the independent, identical subtasks of an independent-task application (ITA) whose data initially resides at one location. The size of individual subtasks and of their results is large, and so transfer times cannot be neglected. The application that we have used for our experiments is NCBI's nucleotide-nucleotide sequence comparison tool BLAST [2].

Our choice of using an ITA for our proof-of-concept implementation follows a common practice in grid scheduling research. However our scheme is general enough to accommodate other classes of applications. In a recent article we have demonstrated using a fault-tolerant implementation of Cannon's matrix multiplication algorithm that our scheduling scheme can be adapted to applications with communicating tasks [8, 9].

1.3.2 Basic Agent Design

A large computational task is encapsulated in a strongly mobile agent. 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.

```
receive request for s subtasks from node c
// c may be the node itself
if subtask_list.size>=s
  c.send_subtasks(s)
else
  c.send_subtasks(subtask_list.size)
  outstanding_subtask_queue.
    add(c,s--subtask_list.size)
  parent.
    send_request(outstanding_subtask_queue.
                 total_subtasks)
```

Fig. 1.2 Behavior of Node on Receiving Request

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. This can be seen in Figures 1.2 and 1.3.

1.3.3 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

```
receive t subtasks from parent
subtask_list.add(t)
if outstanding_subtask_queue.
     total_subtasks>=t
  <send t subtasks to nodes in
   outstanding_subtask_queue>
else
  <send outstanding_subtask_queue.
   total_subtasks subtasks to nodes in
   outstanding_subtask_queue>
// may include subtasks for node itself
```

Fig. 1.3 Behavior of Node on Receiving Subtasks

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. The pseudo-code for the maintenance of child-lists has been presented in Figure 1.4.

1.3.4 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

```
receive feedback from node c
if child_list.contains(c)
  child_list.update_rank(c)
else
  child_list.add(c)
  if child_list.size>MAX_CHILD_LIST_SIZE
    sc:=child_list.slowest
    child_list.remove(sc)
    old_child_list.add(sc)
    inverted_child_list:=inv(child_list)
    sc.send_ancestor_list(inverted_child_list)
```

Fig. 1.4 Behavior of Parent Node on Receiving Feedback

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.

A node periodically informs its parent about its best-performing child. The parent then checks whether its grandchild is present in its list of former children. If not, it adds the grandchild to its list of potential children and tells this node that it is willing to consider the grandchild. The node then instructs its child to contact its grandparent directly. If the contact ends in a promotion, the entire subtree with the child node at its root will move one level higher in the tree. This constant restructuring results in fast nodes percolating towards the root of the tree and has been detailed in Figures 1.5 and 1.6. The checking of a promising child against a list of former children prevents the occurrence of trashing 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

```
receive node b from node c
if old_child_list.not_contains(b)
  potential_child_list.add(b)
  c.send_accept_child(b)
else
  c.send_reject_child(b)
```

Fig. 1.5 Behavior of Parent Node on Receiving Propagated Child

```
receive accept_child(b) from parent
// a request was earlier made to parent
// about node b
b.send_ancestor_list(ancestor_list)
// b will now contact parent directly
```

Fig. 1.6 Behavior of Child Node on Receiving Positive Response

```
receive message from parent
ancestor_list := message.ancestor_list
if parent != ancestor_list.last
  parent:=ancestor_list.last
```

Fig. 1.7 Behavior of Node on Receiving new Ancestor-List

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. The actions of a node on receipt of a new list of ancestors are in Figure 1.7.

1.3.5 Size of Result Burst

Each agent of an ITA ranks its children on the basis of the time taken to send some results to this node. The time required to obtain just one result-burst, or a result-burst of size 1, might not be a good measure of the performance of a child. Nodes might make poor decisions about which children to keep and discard. The child propagation algorithm benefits from using the average of R result-burst intervals and from setting r, the result-burst burst size, to be greater than 1. A better measure for the performance of a child is the time taken by a node to obtain $r * (R + 1)$ results. However, r and R should not be set to very large values because the overlay network would take too much time to take form and to get updated.

1.3.6 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.

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 $(a - 1)$-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, 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. The pseudo-code for the fault tolerance algorithm is in Figure 1.8.

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

```
while true
  send message to parent
  if <unable to contact parent>
    ancestor_list.remove(parent)
    if ancestor_list.size = 0
      <find-new-parent or self-destruct>
    parent := ancestor_list.last
```

Fig. 1.8 Fault Tolerance — Contacting Ancestors

and no new task can be obtained from the parent, unfinished tasks are handed out again.

1.3.7 Cycles in the Overlay Network

Even though the scheduling overlay network should be a tree, failures could cause the formation of a cycle of nodes. The system recovers from this situation by having each node examine its ancestor list on receiving it from its parent. If a node finds itself in that list, it knows that a cycle has occurred. The node attempts to break the cycle by obtaining the address of some other agent on its data distribution or communication overlays. However, if these are identical to the scheduling overlay, the node will be starved of work. If the agent is starved of work for more than a specified time, it self-destructs.

1.3.8 Termination

The root of the tree is informed when the computational task has been completed. It sends a termination message to each of its actual, potential and former children. The computation agent on the root then self-destructs. The children of the root do the same. Termination messages spread down to the leaves and the computation terminates. There are two scenarios in which termination could be incomplete:

- A termination message might not reach a node. The situation is the same as that described in Subsection 1.3.6.

- Consider that computation agents are executing on nodes $n1$ and $n2$. $n1$ receives a termination message, but $n2$ does not because of a failure. The agent on $n1$ destroys itself. $n1$ now sends request messages to its friends. If one of these is $n2$, a clone of $n2$'s agent is sent to $n1$.

  An unchecked spread of computation will not occur because agents send out clones only if they do not have any uncomputed subtasks. $n1$ and $n2$ will eventually run out of subtasks and destroy themselves as explained in Subsection 1.3.6.

1.3.9 Self-adjustment of Task List Size

A node always requests a certain number of subtasks and obtains their results before requesting more subtasks to work on. The size of a subtask is simply an estimation of the smallest unit of work that every machine on the peer-to-peer network should be able to compute in a time that the user considers reasonable; scheduling should not be inordinately slow on account of subtasks that take a long time to compute. However, in an ITA-type application, the utilization of a high-performance machine may be poor because it is only requesting a fixed number of subtasks at a time.

A node may request more subtasks in order to increase the utilization of its resources and to improve the system computation-to-data ratio. A node requests a certain number of subtasks, t, that it will compute itself. Once it has finished computing the t subtasks, it compares the average time to compute a subtask on this run to that of the previous run. Depending on whether it performed better, worse or about the same, the node requests $i(t)$, $d(t)$ or t subtasks for its next run, where $i(t) > t$ and $d(t) < t$.

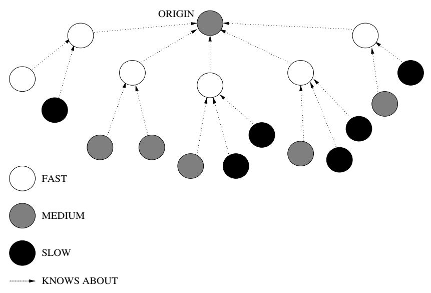


Fig. 1.9 Good Configuration with A Priori Knowledge

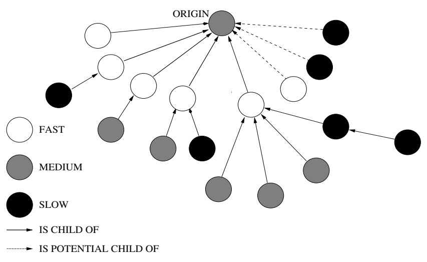


Fig. 1.10 Final Node Organization, Resultburst size=3, Good Initial Configuration

1.3.10 Prefetching

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

It might be beneficial to use prefetching to reduce the time that a node waits for subtasks. A node determines that it should request t subtasks from its parent. It also makes an optimistic prediction of how many subtasks it might require in future by using the i function that is used for self-adjustment. $i(t)$ subtasks are then requested from the parent. When a node finishes computing one set of subtasks, more subtasks are readily available for it to work on, even as a request is submitted to the parent. This interleaving of computation and communication reduces the time for which a node is idle.

While prefetching will reduce the delay in obtaining new subtasks to work on, it also increases the amount of data that needs to be transferred at a time from the root to the current node, thus increasing the synchronization delay and data transfer time. This is why excessively aggressive prefetching will result in a performance degradation.

Table 1.2 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

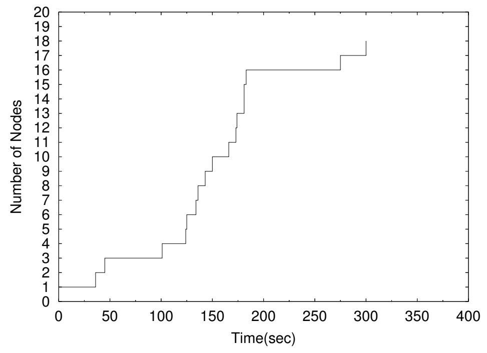


Fig. 1.11 Code Ramp-up

1.4 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.

The application we used to test our system was the gene sequence similarity search tool, NCBI's nucleotide-nucleotide BLAST [2] — 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 1.12, 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 1.2. 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 1.11. This is the same for all the experiments.

1.4.1 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.

Two experiments were conducted using the parameters in Table 1.2. 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. Figures 1.9 and 1.10 represent the start and end of this experiment. 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 1.12. The resulting configuration shown in Figure 1.13 is substantially

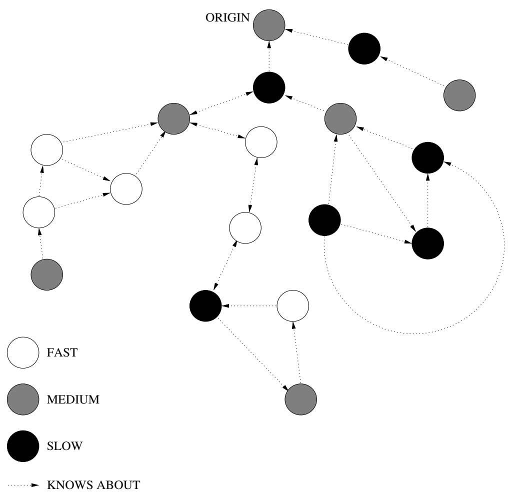


Fig. 1.12 Random Configuration of Machines

Table 1.3 Effect of Prior Knowledge

Configuration	Running Time (sec)
original	2294
good	1781

Table 1.4 Effect of Child Propagation

Scheme	Running Time (sec)
With	2294
Without	3035

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.

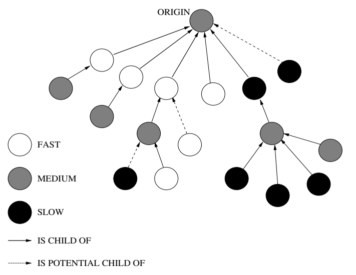


Fig. 1.13 Final Node Organization, Result-burst size=3, With Child Propagation

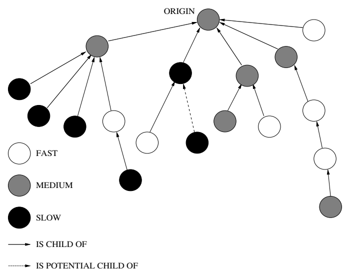


Fig. 1.14 Final Node Organization, Result-burst size=3, No Child Propagation

1.4.2 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 1.4 and Figures 1.13 and 1.14. 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.

1.4.3 Result-burst size

The experimental setup in Table 1.2 was again used. We then ran the experiment with different result-burst sizes. The running times have been tabulated in Table 1.5. The child evaluations that are made by nodes on the basis of one result are poor. The nodes' child-lists change frequently and are far from ideal, as in Figure 1.15.

There is a qualitative improvement in the child-lists as the result-burst size increases. The structure of the resulting overlay networks for result-burst sizes 3 and 5 are in Figures 1.16 and 1.17. However, with very large result-bursts, it takes longer for the tree overlay to form and adapt, thus slowing down the experiment. This can be seen in Figure 1.18.

Table 1.5 Effect of Result-burst Size

Result-burst Size	Running Time (sec)
1	3050
3	2294
5	2320
8	3020

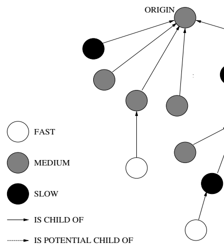


Fig. 1.15 Node Organization, Result-burst size=1

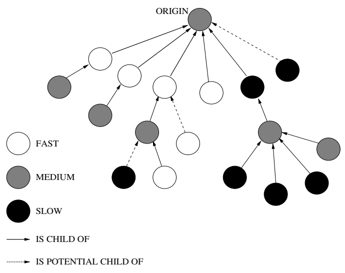


Fig. 1.16 Node Organization, Result-burst size = 3

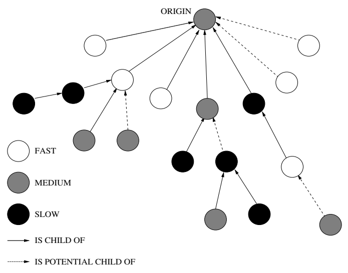


Fig. 1.17 Node Organization, Result-burst size=5

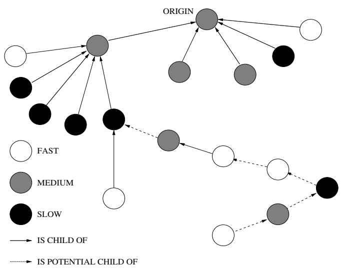


Fig. 1.18 Node Organization, Result-burst size=8

1.4.4 Prefetching and Initial Task Size

The data ramp-up time is the time required for subtasks to reach every single node. Prefetching has a positive effect on this. The minimum number of subtasks that each node requests also affects the data ramp-up. The greater this number, the greater the

Table 1.6 Effect of Prefetching and Minimum Number of Subtasks

No. of Subtasks	Ramp-up Time (sec)		Running Time (sec)	
	Prefetching	No prefetching	Prefetching	No prefetching
1	406	590	2308	2520
2	825	979	2302	2190
5	939	1575	2584	2197

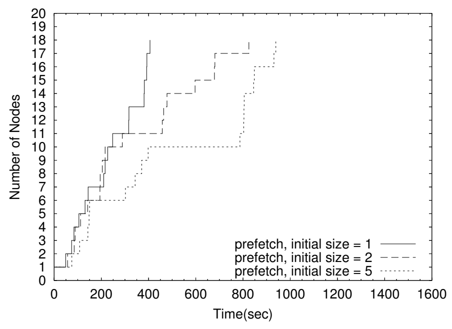


Fig. 1.19 Effect of Minimum Number of Subtasks on Data Ramp-up with Prefetching

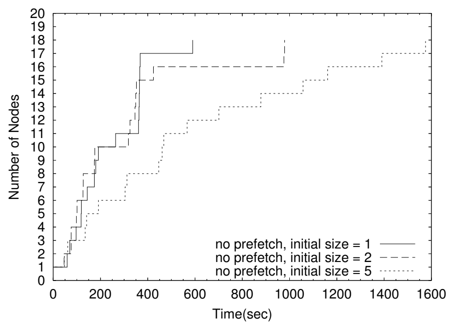
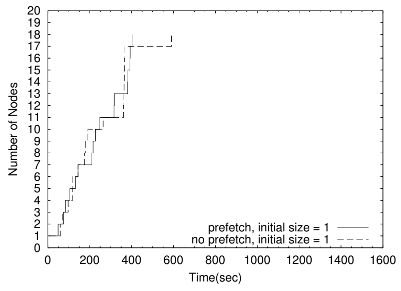


Fig. 1.20 Effect of Minimum Number of Subtasks on Data Ramp-up without Prefetching

amount of data that needs to be sent to each node, and the slower the data ramp-up. This can be seen in Table 1.6 and Figures 1.19, 1.20, 1.21, 1.22 and 1.23.

Prefetching does improves the ramp-up, but of paramount importance is its effect on the overall running time of an experiment. This is also closely related to the minimum number of subtasks requested by each node. Prefetching improves system throughput when the minimum number of subtasks requested is one. As the minimum number of subtasks requested by a node increases, more data needs to be transferred at a time from the root to this node, and the effect of prefetching becomes negligible. As this number increases further, prefetching actually causes a degradation in throughput. Table 1.6 and Figure 1.24 summarize these results.



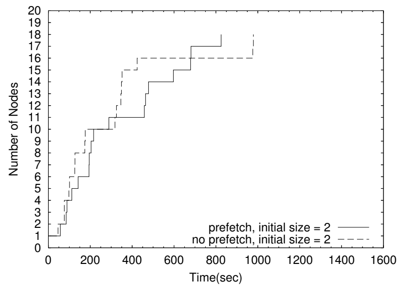
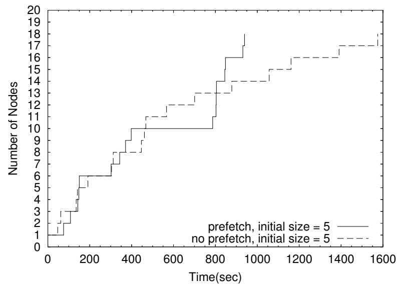


Fig. 1.21 Effect of Prefetching on Data Ramp-up with Minimum Number of Subtasks = 1

Fig. 1.22 Effect of Prefetching on Data Ramp-up with Minimum Number of Subtasks = 2



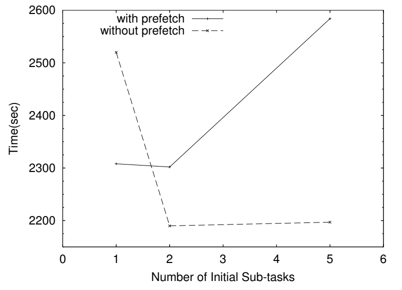


Fig. 1.23 Effect of Prefetching on Data Ramp-up with Minimum Number of Subtasks = 5

Fig. 1.24 Effect of Prefetching and Min. No. of Subtasks

1.4.5 Self-Adjustment

We ran an experiment using the configuration in Table 1.2 and then did the same using constant and exponential self-adjustment functions instead of the linear one. The data ramp-ups have been compared in Table 1.7 and Figure 1.25. The ramp-up with exponential self-adjustment is appreciably faster than that with linear or constant self-adjustment. The aggressive approach performs better because nodes prefetch a larger amount of subtasks, and subtasks quickly reach the nodes farthest from the root.

Table 1.7 **Effect of Self-adjustment function**

Self-adjustment Function	Ramp-up Time (sec)	Running Time (sec)
Linear	1068	2302
Constant	1142	2308
Exponential	681	2584

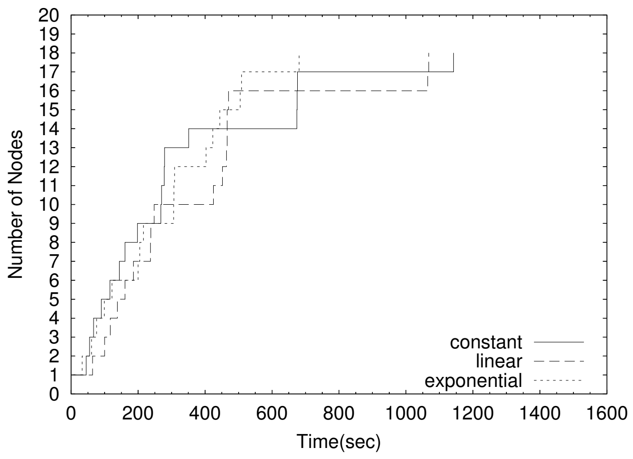


Fig. 1.25 Effect of Self-adjustment Function on Data Ramp-up Time

We also compared the running times of the three runs which are in Table 1.7. Interestingly, the run with the exponential self-adjustment performed poorly with respect to the other runs. This is due to nodes prefetching extremely large numbers of subtasks. Nodes now spend more time waiting for their requests to be satisfied, resulting in a degradation in the throughput at that node.

The linear case was expected to perform better than the constant one, but the observed difference was insignificant. We expect this difference to be more pronounced with longer experimental runs and a larger number of subtasks.

Table 1.8 Effect of No. of Children on Data Ramp-up

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

Table 1.9 Effect of No. of Children on Running Time

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

1.4.6 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 1.8. 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 of Figure 1.9, 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 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 1.9. Similar results were observed even when the child-propagation mechanisms were turned off.

1.5 CONCLUSIONS

We have designed an autonomic scheduling algorithm in which multi-threaded agents with strong mobility form a tree-structured overlay network. The structure of this tree is varied dynamically such that the nodes that currently exhibit good performance are brought closer to the root, thus improving the performance of the system.

We have described experiments with scheduling a representative computational biology application whose data initially resides at one location and whose subtasks have considerable data transfer times. The experiments were conducted on a set of machines distributed across Ohio. While this paper concentrated on a scheduling scheme for independent-task applications, we are experimenting with adapting the algorithm for a wide class of applications. Recent results show that our approach can be adapted to communicating applications, such as Cannon's algorithm for parallel matrix multiplication [8, 9].

It is our intention to present a desktop grid application developer with a simple application programming interface that will allow him/her to customize the scheduling scheme to the characteristics of an application. A prototype of this has already been implemented.

An important problem that we will address in future is the initial assignment of the friend-list. There has been some research on the problem of assigning friend-lists [15, 40], and we will consider how best to apply this to our own work.

The experimental platform was a set of 18 heterogeneous machines. In future, we plan to harness the computing power of idle machines across the Internet by running a mobile agent platform inside a screen saver in order to create a desktop grid of a scale of the tens or hundreds of thousands. Researchers will then be free to deploy scientific applications on this system.

1.6 FUTURE DIRECTIONS

Despite the obvious success of current master-workers internet computing systems such as SETI@home and folding@home, the range of applications that can be successfully employed with this approach is still limited. In this section we describe some applicative scenarios that showcase the Organic Grid advantage over current approaches, and how it could substantially expand the use of large scale Internet computing.

Ab initio prediction of protein folding involves the simulation of molecule folding using inter-atomic forces. This type of simulation requires the analysis of a very large number of atomic conformations over an energy landscape. The exploration of the conformational space can in principle be parceled to different nodes, but some kind of periodic data exchange is needed to ensure that only the simulation of the most promising conformations are carried forward.

An example of this approach is the "synchronize&exchange" algorithm called Replica Exchange Molecular Dynamics used in the Folding@home project [41]. This requires light communication that can go through the root but could be made more scalable with horizontal communication. Furthermore, to this date only simple proteins have been studied with the folding@home system. Complex molecular systems (i.e., protein-solvent interaction) could be studied by adopting more sophisticated molecular dyynamic simulation algorithms, which typically need more inter-node communication. One of these algorithms is GROMACS, which assumes a ring node topology and has been parallelized using PVM and MPI [31, 48].

Another promising applicative scenario for the Organic Grid is phylogenetic analysis. In order to make evolutionary and functional inferences, biologists use evolutionary trees to find correlated features in DNA and phenotypes and derive evolutionary or functional inferences. Evolutionary tree search is computationally intensive as the number of candidate trees is combinatorially explosive as more organisms are considered. Commonly used heuristics include the consideration of many candidate trees with randomization of the order in which organisms are added as trees are built.

Most advances in parallel computing for evolutionary trees follow a coarse grained approach in which candidate trees are evaluated independently in a one-replica-per-processor strategy. This strategy reduces overall search time but only allows the investigator to evaluate a number of trees concurrently rather than evaluating a single tree faster. An Organic Grid implementation could allow the parallelization of a candidate tree analysis by enabling horizontal communication between same level nodes in a tree overlay. Faster methods to evalutate single trees are crucial because with the advent of rapid whole genome sequencing technologies researchers are beginning to consider entire genomes for tens to hundreds of organisms.

In general, the Organic Grid support for arbitrary communication topologies could open the door to the use of algorithms previously unfit for implementation on a desktop grid. As demonstrated in our matrix multiplication experiment [8, 9], the agent behavior can be personalized to build an application-specific communication overlay in addition to the service tree overlay, with only a minimal amount of modifications required to the application itself.

Acknowledgments

This work was partially funded with Ohio Supercomputer Center grants PAS0036-1 and PAS0121-1.

REFERENCES

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

2. Basic Local Alignment Search Tool (BLAST). Web site at http://www.ncbi.nlm.nih.gov/BLAST/.

3. F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

4. 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.

5. Jeffrey Bradshaw, Niranjan Suri, Alberto J. Cañas, Robert Davis, Kenneth M. Ford, Robert R. Hoffman, Renia Jeffers, and Thomas Reichherzer. Terraforming cyberspace. In *Computer*, volume 34(7). IEEE, July 2001.

6. D. Buaklee, G. Tracy, M. K. Vernon, and S. Wright. Near-optimal adaptive control of a large grid application. In *Proceedings of the International Conference on Supercomputing*, June 2002.

7. 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.

8. Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. Application-specific scheduling for the Organic Grid. Technical Report OSU-CISRC-4/04-TR23, Dept. of Computer and Information Science, The Ohio State University, April 2004.

9. Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. Application-specific scheduling for the Organic Grid. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*, Pittsburgh, November 2004.

10. Arjav J. Chakravarti, Xiaojin Wang, Jason O. Hallstrom, and Gerald Baumgartner. Implementation of strong mobility for multi-threaded agents in Java. In *Proceedings of the International Conference on Parallel Processing*. IEEE Computer Society, October 2003.

11. Arjav J. Chakravarti, Xiaojin Wang, Jason O. Hallstrom, and Gerald Baumgartner. Implementation of strong mobility for multi-threaded agents in Java. Technical Report OSU-CISRC-2/03-TR06, Dept. of Computer and Information Science, The Ohio State University, February 2003.

12. 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.

13. Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In *Mobile Object Systems: Towards the Programmable Internet*, 1996.

14. United Devices. Web site at http://www.ud.com.

15. The Gnutella download. Web site at http://www.gnutelliums.com.

16. The folding@home project. Web site at http://folding.stanford.edu.

17. Berkeley Open Infrastructure for Network Computing (BOINC). http://boinc.berkeley.edu/.

18. I. Foster, C. Kesselman, J.Nick, and S. Tuecke. The physiology of the Grid: An open Grid services architecture for distributed systems integration, 2002. http://www.globus.org/research/papers.html.

19. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3), 2001.

20. R. Ghanea-Hercock, J.C. Collis, and D.T. Ndumu. Co-operating mobile agents for distributed parallel processing. In *Third International Conference on Autonomous Agents (AA '99)*, pages 398–399, Mineapolis, MN, May 1999. ACM Press.

21. A. Gierer and H. Meinhardt. A theory of biological pattern formation. *Kybernetik*, 12:30–39, 1972.

22. Andrew S. Grimshaw and W. A. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, January 1997.

23. 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*, 2000.

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

25. 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.

26. 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.

27. 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.

28. Danny B. Lange and Mitsuru Oshima. *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley, 1998.

29. Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, March 1999.

30. 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.

31. Hess B. Lindahl E. and van der Spoel D. Gromacs 3.0: A package for molecular simulation and trajectory analysis. *J. Mol. Mod.*, 7(8):306–317, 2001.

32. 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.

33. 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.

34. 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.

35. A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

36. B.J. Overeinder, N.J.E. Wijngaards, M. van Steen, and F.M.T. Brazier. Multi-agent support for Internet-scale Grid management. In O. Rana and M. Schroeder, editors, *AISB'02 Symposium on AI and Grid Computing*, pages 18–22, April 2002.

37. The Globus Project. Web site at http://www.globus.org.

38. The SETI@home project. Web site at http://setiathome.ssl.berkeley.edu.

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

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

41. Young Min Rhee and Vijay S. Pande. Multiplexed-replica exchange molecular dynamics method for protein folding simulation. *Biophysical Journal*, 84(2):775–786, 2003.

42. 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.

43. Clay Shirky. What is P2P ... and what isn't? *O'Reilly Network*, November 2000.

44. John F. Shoch and Jon A. Hupp. The "worm" programs — early experience with a distributed computation. *Communications of the ACM*, March 1982.

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

46. Guy Theraulaz, Eric Bonabeau, Stamatios C. Nicolis, Ricard V. Solé, Vincent Fourcassié, Stéphane Blanco, Richard Fournier, Jean-Louis Joly, Pau Fernández, Anne Grimal, Patrice Dalle, and Jean-Louis Deneubourg. Spatial patterns in ant colonies. *PNAS*, 99(15):9645–9649, 2002.

47. A. Turing. The chemical basis of morphogenesis. *Philos. Trans. R. Soc. London*, 237(B):37–72, 1952.

48. GROMACS web site. http://www.gromacs.org/.

49. 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.

50. G. Woltman. Web site for the Marsenne Prime project at http://www.mersenne.org/prime.htm.