The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network

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
Email: {arjav, gb, lauria}@cis.ohio-state.edu

Technical Report: OSU-CISRC-10/03-TR55

October 24, 2003

Abstract

Desktop grids have already been used to perform some of the largest computations in the world and have the potential to grow by several more orders of magnitude. However current approaches to utilizing desktop resources require either centralized servers or extensive knowledge of the underlying system, limiting their scalability.

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. In a radical departure from current models, we envision large scale desktop grids in which agents autonomically organize themselves so as to maximize resource utilization.

We demonstrate this concept with a reduced scale proof-of-concept implementation that executes a data-intensive parameter sweep application on a set of heterogeneous geographically distributed machines. We present a detailed exploration of the design space of our system and a performance evaluation of our implementation using metrics appropriate for assessing self-organizing desktop grids.

1 Introduction

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.

While impressive, these efforts only use a tiny fraction of the desktops connected to the Internet. Order of magnitude improvements could be achieved if the design of existing systems could be scaled up to reach a proportionally larger share of machines.

The basic design constraints for these desktop grid systems can be summarized as follows: negligible impact to the primary user; efficient resource usage in the presence of wide variations in resource parameters such as computing power and communication bandwidth; adaptivity under extreme temporal variations in resource availability. Given these constraints, a software infrastructure with unprecedented levels of robustness, scalability, and adaptivity is required.

The goal of utilizing the CPU cycles of idle machines was first realized by the Worm project [1] at Xerox PARC. Further progress was made by academic projects such as Condor [2]. The growth of the

Internet made large-scale efforts like GIMPS [3], distributed.net [4], SETI@home [5] and folding@home [6] feasible. Recently, commercial solutions such as Entropia [7] and Grid MP [8] have also been developed.

Existing designs for desktop grid systems tend to follow two models. Systems like GIMPS, SETI@home, distributed.net, folding@home and Entropia are based on a master/worker model and rely on the on-demand distribution of tasks to provide an inherently decentralized and self-balancing form of scheduling. The main shortcoming of this solution is its reliance on a single master which severely constrains the computing-to-data ratio on large systems, represents a single point of failure, puts a constraint on the number of users that can use the system simultaneously, and limits the available resource pool to those that are within good reach of the server.

The other design philosophy is an extension of traditional Massively-Parallel Processing (MPP) and Grid scheduling models in that it assumes that sufficient information is available on the underlying system to precompute a resource-optimizing schedule. A number of meta-schedulers have been built that can compute optimal schedules under this assumption [9, 10, 11, 12]. Clearly, with desktop grids comprising hundreds of thousands of machines, it is unrealistic to assume a satisfactory knowledge about the network topology, the number of cycles available at each node, and the bandwidth of every network connection.

In this paper, we introduce a new approach to the utilization of desktop grids, which represents a radical departure from current models. Firstly, we do away with the notion of a central scheduling authority and the related assumptions about the amount of information available on the system. Instead, we propose a self-organizing approach to computation which is reminiscent of the way complex biological systems organize themselves. In our design, a large computation is broken down into autonomous units, each endowed with unrestricted mobility and a set of uniform behavioral rules; we then let these units arrange themselves over a network of machines, constrained only by resource availability. Our approach is inspired by models of organization in which complex patterns emerge from the interplay of a large number of agents exhibiting simple behavior. Examples of such patterns are observed in shell motifs, in neural structures, and in the behavior of social insects [13, 14].

The other distinctive aspect of our approach is that we pursue a true Peer-to-Peer approach. Our system is based on the voluntary sharing of resources; every node in the system has equal access and the same role as anyone else, including the possibility of submitting tasks. The advantages of this organization are: i) better utilization of every part of the system, ii) absence of centralized server bottlenecks, iii) better fault tolerance, iv) simultaneous access to the resources for a large number of users.

In summary, we suggest that this combination of task mobility and self-organizing behavior achieves an ”organic” organization of the computation, which is well suited for large and dynamic systems because of the completely decentralized nature of control. The success of this type of organization is evident in complex biological systems with which our organic grid shares its underlying design principles.

The main contributions of this paper are: i) the description of a new organization principle for desktop grids which combines autonomous scheduling with strongly mobile agents, ii) the demonstration of these principles into a working proof-of-concept prototype, iii) a detailed exploration of the design space of our system, and iv) the performance evaluation of our design using metrics appropriate for assessing self-organizing desktop grids.

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, evaluations of scalability, degree of tolerance to faults and adaptivity to rapidly changing systems, have been left for future studies.

2 Background and Related Work

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

2.1 Self-Organization of Complex Systems

The organization of many complex biological and social systems has often been explained in terms of aggregations of a large number of autonomous entities or agents 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 [13, 14]. The existence of this mechanism, often referred to as *emergence*, was proposed to explain complex patterns such as, among others, shell motifs, animal coats, several types of social insect behaviors, or neural structures.

Recently, a particular version of this principle called Local Activation, Long-range Inhibition (LALI) was for the first time formally proven to be responsible for a complex pattern through a clever experiment on ants [15]. The LALI rule is based on two types of interaction: a positive, reinforcing interaction that works over a short range, and a negative, destructive one that works over longer distances. In the ant experiment this rule has been shown to fully account for the formation of cemeteries in ant colonies. An application of ant behavior to peer-to-peer systems has been proposed [16]. In our project, we are interested in a more general approach where the aim is maximizing system utilization rather than the accurate reproduction of animal behavior. We retain the LALI principle but in a different form; we use a notion of distance based on throughput rather than physical location, and we encourage the propagation of computation among well-connected nodes while discouraging the inclusion of "distant" agents.

2.2 Peer-to-Peer Computing

Peer-to-peer computing [17, 18] adopts a highly decentralized approach to resource sharing and enables the building of flexible systems. Current peer-to-peer systems rely on highly available central servers, however, and are mainly used for high profile, large scale computational projects such as SETI@home, or for popular data-centric applications like file-sharing [19].

Applications that leverage the capabilities of these systems may make use of data that is distributed across a network. Program code may also be stored at multiple locations, and subprograms may communicate with one another during execution, i.e., applications need not necessarily be massively parallel. Decentralized scheduling is important to avoid bottlenecks, to keep applications scalable, and to dynamically utilize resources. In this paper, we propose a peer-to-peer framework that differs from both current-generation desktop grid systems and Grid computing approaches.

2.3 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. Checkpointing is a time-honored technique, but its storage and computational overheads put additional constraints on the design of the system, particularly on the amount of state that can be handled and the length of the minimum useful stretch of continuous availability of a computing resource. To avoid this drawback, desktop grids need to support the asynchronous and transparent migration of processes across machine boundaries.

Most distributed applications are currently built with distributed object technologies, such as Java RMI. RPC-based approaches do not consider the execution state of their arguments. If a thread is active within one of the arguments passed to a remote procedure, it does not travel along with the argument.

Mobile agents [20, 21] 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, the scheduling of other agents, and communication with other agents on a network.

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* [22]. Weak mobility forces programmers to use a difficult programming style.

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 to solve a problem, 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.

Strong mobility is added to Java by using modified or custom VMs [23, 24, 25, 26, 27, 28, 29], or by changing the compilation model [30, 31, 32, 33]. The first approach suffers from a lack of portability, while implementations that use the second approach are either unable to migrate multi-threaded agents, or do not provide support for forced mobility.

We have chosen to provide strong mobility for Java by using a preprocessor that translates strongly mobile source code into weakly mobile source code [34, 35, 36]. The generated weakly mobile code maintains a movable execution state for each thread at all times. Our implementation is designed for the full Java programming language.

2.4 Scheduling

The scheduling of decomposable applications has been studied extensively in the context of heterogeneous processing resources [37, 38, 39, 40, 41, 42]. All of these approaches have some resemblance of centralized scheduling, making them inappropriate for large desktop grids. Master/worker schemes have been explored in [43] and [44]. Both suffer from a lack of scalability and the presence of a single point of failure. In addition, some of these results are restricted to homogeneous worker resources.

Some research into decentralized scheduling has been conducted as well. Two-level scheduling schemes have been considered [45, 46], but these are not scalable enough for the Internet. In the scheduling heuristic described by Leangsuksun et al. [47], 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 the authors' scheme is practical in large-scale and highly dynamic environments such as a desktop grid system.

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 [48]. 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.

Truly distributed and autonomous/autonomic scheduling protocols have been developed for peer-to-peer computing. A biological system is the basis of the novel scheduling scheme discussed in [16]. This scheme is designed to disperse tasks uniformly over a network, which is not optimal in the context of a computational grid where tasks should ideally be assigned in proportion to resource availability.

A bandwidth-centric approach has been proposed that assumes the presence of an overlay network with a tree structure — the tasks originate at the root [49]. Each node in the system sends tasks to and receives results from all of its children. The children are prioritised on the basis of the bandwidth of their connection to their parent. This minimizes communication time and increases processor utilization. However, the performance of the system is entirely dependent on the initial selection of the tree overlay network. The lack of dynamism is bound to affect the performance of the scheme. The authors also do not elaborate on the issue of choosing such an optimal tree; this appears to require accurate information about system parameters such as the bandwidth of the connections and the computational capacity of the nodes.

The unique constraints of a desktop grid system were kept in mind when designing and implementing the scheduling algorithm, particularly the absence of any knowledge of the network topology, machine configurations, or connection bandwidths. In particular, from the point of view of network topology our system starts with a minimal amount of knowledge (in the form of the "friend-list"), and then build its own overlay network on the fly. The construction of the overlay network reflects the availability of computational and communication bandwidth resources. We only assume that a "friend-list" is available initially on each node to prime the system; the construction of such lists has been discussed in the context of peer-to-peer file-sharing [19, 50, 51].

3 Autonomic Scheduling

3.1 Agent Behavior Design

We have developed an autonomic, self-reliant scheduling scheme based on the behavior of mobile agents that organize themselves on a peer-to-peer network. Although our scheme is general enough to accommodate several different classes of applications, we focus on the solution to one particular problem in this paper: the scheduling of the independent, identical subtasks of a parameter sweep-type application 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 [52].

In designing the behavior of the mobile agents, we faced the classic issues of any distributed computation in a dynamic environment: distribution of the data, discovery of new nodes, load balancing, collection of the results, tolerance to faults, detection of task completion. The solutions for all these issues had to be cast in terms of direct interaction between agents. We proceeded by taking some early design decisions based on our intuition, and then we refined them with an iterative process of implementation, testing on our testbed, performance analysis, and new implementation. A modular design in which different aspects of the behavior were implemented as distinct routines allowed us to deal with the above issues separately.

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

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

The following sections will describe the various aspects of agent behavior in detail, as well as their relation to the global progress of the computation.

3.2 Basic Scheme

Since we currently use the `Aglets` weak mobility library, an `Aglets` environment gets set up when a machine becomes available (for example when the machine has been idle for some time; in our experiments we assumed the machines to be available at all times).

Every machine has a list of the `URL`s of other machines that it could ask for work. This list is known as the *friend-list*. It is used for constructing the initial overlay network. The problem of how to form this initial list is not addressed in this paper. For now, we assume that every agent has such a list.

The environment creates a stationary agent, which asks the friends for work by sending them messages. If a request arrives at a machine that has no computation running on it, the request is ignored. Nothing is known about the configurations of the machines on the friend-list, or of the bandwidths or latencies of the links to them, i.e., the algorithm is zero-knowledge and appropriate for dynamic, large-scale systems.

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

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

When the subtasks on a machine have been completely executed, the agent on that machine requests its parent for more subtasks to work on. The parent attempts to comply. Even if the parent does not have the requested number of subtasks, it will respond and send its child what it can. The parent keeps a record of the number of subtasks that remain to be sent, and sends a request for those tasks to its own parent. This can be seen in Figures 1 and 2.

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

3.3 Maintenance of Child-lists

A node cannot have an arbitrarily large number of children because this will adversely affect the synchronization delay at that node. Since the data transfer times of the independent subtasks are large, a node might have to wait for a very long time for its request to be satisfied. Therefore, each node has a fixed number of children, c. The number of children also should not be too small so as to avoid deep trees which will lead to long delays in propagating the data from the root to the leaf nodes. These children are ranked by the rate at which they send in results. When a child sends in r results and the time that was required to obtain them, its ranking is updated. This ranking is a reflection of the performance of not just a child node, but of the entire subtree with the child node as its root. This factors in the scenario of there being a cluster

```
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)
```

Figure 1: Behavior of Node on Receiving Request

```
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>
// this may include subtasks for
// node itself
```

Figure 2: Behavior of Node on Receiving Subtasks

```
receive result-burst 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)
    sc.send_ancestor_list(child_list)
```

Figure 3: Behavior of Parent Node on Receiving Result-Burst

```
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)
```

Figure 4: 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
```

Figure 5: Behavior of Child Node on Receiving Positive Response to Propagation of Grandchild

of high-performance compute nodes available to do work, but with only a slow front end that is publicly accessible. The front end would be at the root of a subtree of compute nodes. The poor performance of the root would not penalize the entire cluster.

In addition to *c* children, a node can also be the parent of *p* potential children. These are nodes that have yet to send *r* results to the current node. The current node needs to receive *r* results from another node before it can accept it as its child. Potential children may request subtasks just like normal children. When a potential child sends its first batch of *r* results, along with the time required to obtain them, it is added to the list of the node's children. 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 set of alternative nodes to contact to try to 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 removed from this list once a user-defined time period elapses. If *sc* now attempts to make requests or send results to the current node within this time period, these messages will remain unacknowledged. The pseudo-code for the maintenance of child-lists has been presented in Figure 3.

3.4 Restructuring of the Overlay Network

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

A node periodically informs its parent about its best-performing child. The parent then checks whether its grandchild is present in its list of former children. If not, it adds the grandchild to its list of potential children and tells this node that it is willing to consider the grandchild. The node then informs the grandchild that it should now contact its grandparent directly. This results in fast nodes percolating towards the root of the tree and has been detailed in Figures 4 and 5.

When a node updates its child-list and decides to remove its slowest child, *sc*, it does not simply discard the child. It sends *sc* a list of its other children, which *sc* attempts to contact in turn. If *sc* had earlier been propagated to this node, a check is made as to whether *sc*'s original parent is still a child of this node. In that case, *sc*'s original parent, *op*, is placed first in the list of nodes being sent for *sc* to attempt to contact. Since *sc* was *op*'s fastest child at some point, there is a good chance that it will be accepted by *op* again. The pseudo-code for this algorithm is in Figure 3. The actions of a node on receipt of a new list of ancestors are in Figure 6.

3.5 Size of Result Burst

Each node ranks its children on the basis of the time that it took the children to send *r* results to this node. The child propagation algorithm benefits from using multiple results, instead of setting *r* to one. Since the time required to obtain just one result may not be a good measure of the performance of a child, nodes may end up making poor decisions about which children to keep and which to discard. The topology of the overlay network would also change too rapidly and the best-performing nodes may not be near the root. We found that a better measure for the performance of a child is the time taken by a node to obtain a small number of results. Even though nodes will now be able to make better evaluations of their children, it will take more time for these children to send results to their parent. Updates to the child-list will now take more time. Therefore, *r* should not be set to a very large value because the overlay network would take too much time to take form and get updated.

3.6 Fault Tolerance

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

A child waits a certain user-defined time for a response after sending a message to its parent — the *a*-th node in its ancestor-list. If the parent is able to respond, it will, irrespective of whether it has any subtasks to send its child at this moment or not. The child will receive the response, check whether its request was satisfied with any subtasks, and begin waiting again if that is not the case. The pseudo-code for this algorithm is in Figure 7.

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

Figure 6: Behavior of Node on Receiving new Ancestor-List

```
while true
  send message to parent
  if <reply received within
        MAX_REPLY_TIME>
    break
  ancestor_list.remove(parent)
  if ancestor_list.size = 0
    <self-destruct>
  parent := ancestor_list.last
```

Figure 7: Fault Tolerance — Contacting Ancestors

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

If a node's ancestor-list does go down to size 0, the node has no means of obtaining any work to do. The mobile agent that computes subtasks informs the agent environment that no useful work is being done by this machine, and then self-destructs. Just as before, a stationary agent begins to send out requests for work to a list of friends.

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

3.7 Cycles in the Overlay Network

Even though the overlay network should be a tree, failures could cause the formation of a cycle of nodes. This cycle of nodes will eventually run out of subtasks to compute. However, this does not guarantee that the compute agents on the nodes in the cycle will die, or that the cycle will be broken. Every node in the cycle requests subtasks of its parent. The parent informs its child that it is still alive and, in turn, requests tasks from its own parent. Nodes may thus wait indefinitely.

This situation is avoided 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 and its computation agent self-destructs.

If the cycle involves a very large number of nodes, the ancestor-list may be too small to include the current node. A node also keeps track of the total time that has elapsed since it last received a subtask. If that time exceeds a user-defined limit, a cycle is assumed to have taken shape and the computation agent on the node destroys itself. Even if a cycle has not actually taken shape, the fact that this node has waited for an excessive amount of time means that subtasks from the root are taking too long to reach it and that this node will be unable to contribute effectively to the computation.

3.8 Termination

The root of the tree is the source of the subtasks that flow to all the other nodes. The root continues to send out subtasks unless it obtains all the results that it requires, i.e., subtasks may be computed redundantly by multiple machines. Once all the results have been computed, the computation needs to be terminated. The root sends a termination message to each of its children, potential children and former children. The computation agent on the root then self-destructs.

The other nodes to which the root sends the termination message do the same. In this way, termination messages spread from the root down to the leaves and all the computation agents on these nodes destroy themselves.

The overlay network is thus destroyed. The stationary agents on the independent machines now begin sending request messages to their lists of friends. This goes on until some user creates a new computation agent.

There are two scenarios in which termination could be incomplete:

- A termination message might not reach a node due to machine or link failures. In that case, the situation is the same as described in Subsection 3.6. The node will try to contact one of its ancestors, and when it fails to get a response from any of them, its computation agent will destroy itself.
- Consider the scenario in which a computation agent executes on a node *n1*, with one of its descendant agents on node *n2*. The agent on *n1* receives a termination message and destroys itself, but that on *n2* does not because of a failure. The stationary agent on *n1* now sends request messages to the machines in its friend-list. If one of these is *n2*, a clone of *n2*'s computation agent is sent to *n1*: the computation has not terminated. This is not a problem because every agent is required to check whether it has any uncomputed subtasks before sending out clones. All the subtasks of a computation come from the root, and the computation agent on the root has obviously destroyed itself. *n2* will eventually run out of subtasks and be unable to propagate any more computation agents. *n2* will unsuccessfully attempt to contact its ancestors and then perish; *n1* will eventually follow.

3.9 Self-adjustment of Task List Size

A node always requests a certain number of subtasks, *s*, in the scheme discussed earlier. It then obtains the results of those many subtask before requesting another subtask to work on. However, 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. If a machine offers good performance, i.e., if it is able to compute subtasks rapidly and/or has a fast network connection, the utilization of that 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 (initially *t=1*). 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 chooses to request *i(t), d(t)* or *t* subtasks for its next run, where *i* and *d* are increasing and decreasing functions, respectively.

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.

We found that it is beneficial to use prefetching for reducing the time that a node waits for subtasks. A node determines that it should request *t* subtasks from its parent. The node 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. *t+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.

Node Category	Delay per Computed Task (sec)	Delay per MB Communicated (sec)
fast	0	0
medium	22.5	1.5
slow	45	3

Table 1: Delays for different machine configurations

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

Table 2: Original parameters

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

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 machines at different locations around Ohio. The machines were of several different configurations and they ran the `Aglets` weakly mobile agent environment on top of either Linux or Solaris.

The scientific application we used to test our system was the gene sequence similarity search tool, NCBI's nucleotide-nucleotide BLAST [52]: a parameter sweep application. The task was to match the same 256KB sequence against 320 data chunks, each of size 512KB. Each subtask was to match the string against one chunk.

The time to compute one of these matchings was around 30 seconds on average. 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 the delays in Table 1.

As shown in Figure 8, 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`.

Our autonomic scheduling algorithm organizes the nodes into a tree such that the machine where the computation originates, i.e., the root, is connected to the subtrees that perform best. The rest of this section will illustrate how that impacts the total running time of the computation.

We ran the computation with the parameters described in Table 2. Linear self-adjustment means that the increasing and decreasing functions of the number of subtasks requested at each node are linear. The ramp-up time, i.e., the time required for the code and the first subtask to arrive at the different nodes can be seen in Figure 10.

We observed negligible variations in the time required for computation agents to reach all eighteen machines, across all our experiments. If the same experiments were run on a much larger number of machines, greater variations in code ramp-up times might be expected.

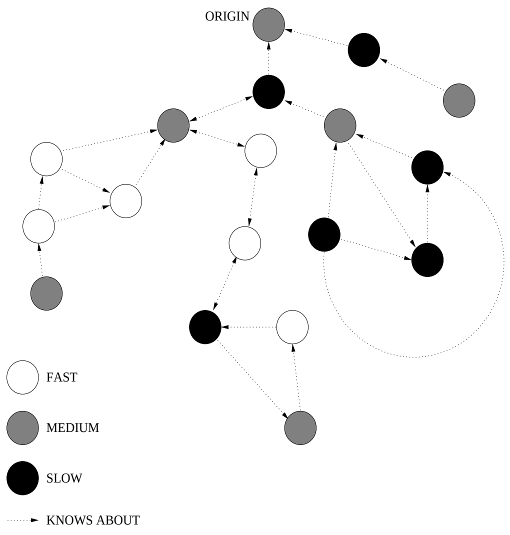

Figure 8: Original Configuration of Machines

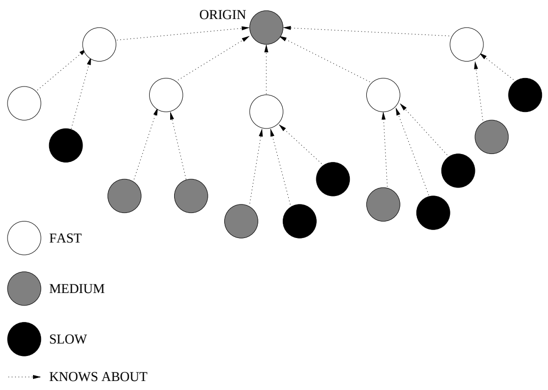

Figure 9: Good Configuration obtained with A Priori Knowledge

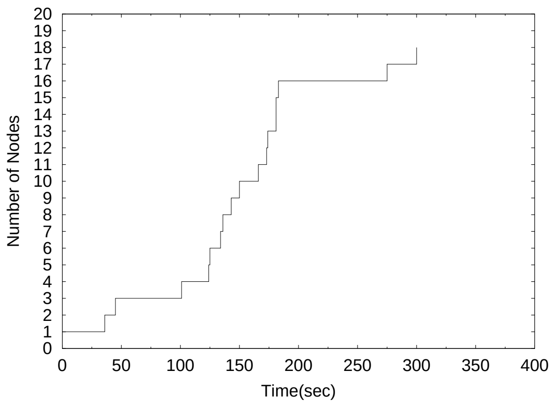

Figure 10: Code Ramp-up

4.1 Comparison with Knowledge-based Scheme

We ran an experiment using the original configuration and parameters in Figure 8 and Table 2, respectively.

With a priori knowledge of the performance of the different machines used in our experiments, and an assumption that they are dedicated resources, a centralized scheduler would be able to come up with a good initial configuration as in Figure 9.

We then organized the nodes as per this topology and ran the experiment using the same parameters as in Table 2. A comparison of the running times is in Table 3, and the scheme that uses a priori knowledge performs 22 % better than a purely zero-knowledge scheme.

It is infeasible, however, to expect a centralized scheduler to be able to work effectively in the large-scale and dynamic environment of a desktop grid.

Configuration	Running Time (sec)
original	2294
with a priori knowledge	1781

Table 3: Effect of A Priori Knowledge on Running Time

Scheme	Running Time (sec)
With child-propagation	2294
Without child-propagation	3035

Table 4: Effect of Child-propagation

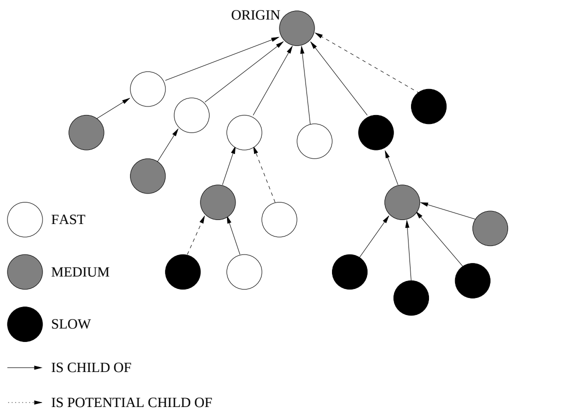

Figure 11: Organization of Nodes at End of Experiment, With Child Propagation

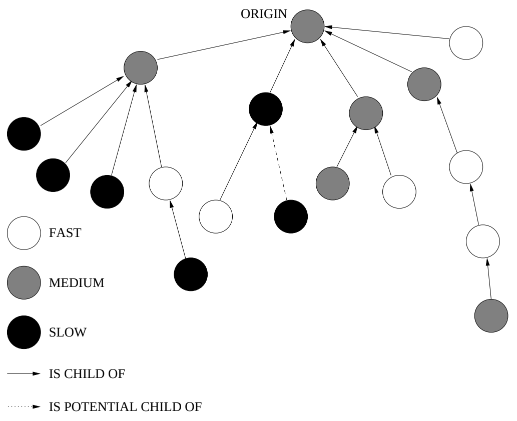

Figure 12: Organization of Nodes at End of Experiment, Without Child Propagation

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 4 and Figures 11 and 12. The solid arrows are from a child to its parent. 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.

It may be expected that the structure of the tree in the case without child-propagation be very similar to the original structure in Figure 8. Our fault-tolerance mechanism is the only reason that they are different. The slow nodes near the root are unable to provide subtasks to their descendents at a high enough rate. Some descendants wait for their timeout periods and then communicate directly with their ancestors.

4.3 Result-burst size

The experimental setup in Table 2 was again used. We then ran the experiment with different result-burst sizes. The running times have been tabulated in Table 5. The variations in the data ramp-up were small enough to be ignored.

When the result-burst is of size 1, nodes evaluate their children based on just one result. They are unable to make good judgments about the performance of their children. This is why the child-lists that they form change frequently and are far from ideal, as is apparent in Figure 13, where only one fast node is adjacent to the root.

As the result-burst size increases, there is a qualitative improvement in the child-lists. The structure of the resulting overlay networks for result-burst sizes 3 and 5 are in Figures 14 and 15. Both these tree structures have several fast nodes adjacent to and close to the root. This is why the running times of the

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

Table 5: Effect of Result-burst Size on Running Time

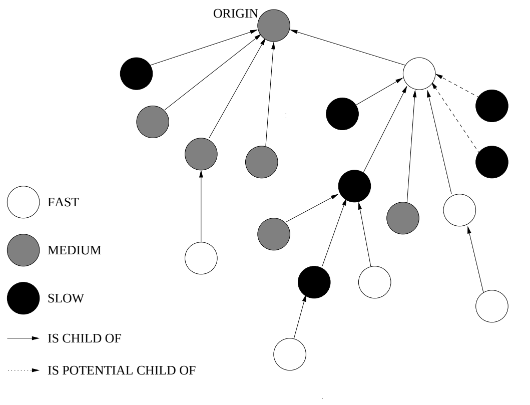

Figure 13: Organization of Nodes at End of Experiment, Result-burst size = 1

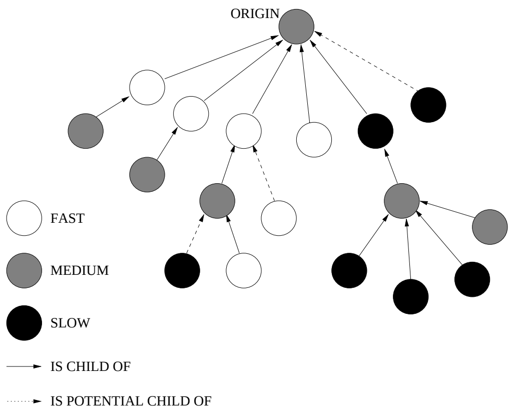

Figure 14: Organization of Nodes at End of Experiment, Result-burst size = 3

experiments decrease with an increase in the size of the result-burst.

However, with still larger child-lists, it takes longer for a node to accept another as its child. Consequently, child propagation takes more time and it takes longer for the overlay network to organize itself. This is why the experiment slows down when the result-burst size is made 8. Figure 16 reveals that the structure of the overlay network is very poor and that there are a large number of potential children that have never been able to send even one result-burst to their parents.

4.4 Prefetching and Initial Task Size

Prefetching is expected to reduce the overall running time of the experiment. This is because each node asks for more subtasks than it actually needs. If its children request some subtasks, it can satisfy those requests immediately.

The data ramp-up time, i.e., the time required for subtasks to reach every single node, is dependent on the minimum number of subtasks that each node requests. The greater this number, the greater the amount of data that needs to be sent to each node, and the slower the data ramp-up. This is true of experiments both with and without prefetching, as can be seen in Table 6 and Figures 17 and 18.

For a given minimum number of subtasks, the time for data ramp-up reduces with prefetching. The effect of prefetching on the ramp-up is shown in Table 6 and Figures 19, 20 and 21.

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 7 and Figure 22 summarize these

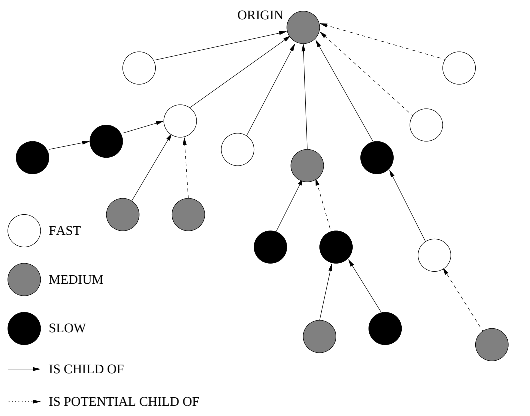

Figure 15: Organization of Nodes at End of Experiment, Result-burst size = 5

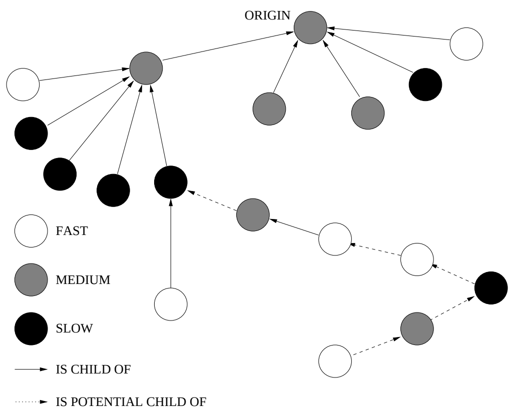

Figure 16: Organization of Nodes at End of Experiment, Result-burst size = 8

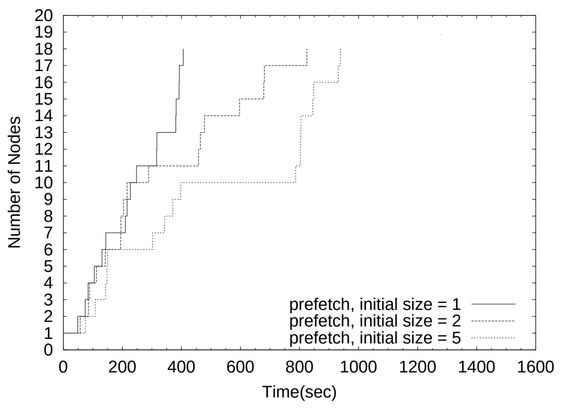

Figure 17: Effect of Min. Number of Subtasks on Data Ramp-up with Prefetching

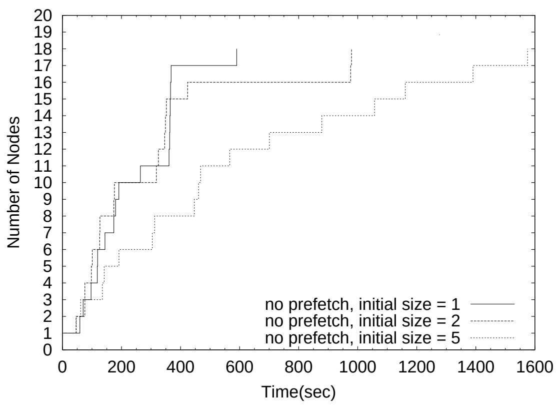

Figure 18: Effect of Min. Number of Subtasks on Data Ramp-up without Prefetching

results.

4.5 Self-Adjustment

The original configuration in Table 2 has a linear self-adjustment function. We used that for one experiment, and then ran the same experiment for two different self-adjustment functions, constant and exponential.

The data ramp-ups have been compared in Table 8 and Figure 23. 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.

The ramp-up was faster with the aggressive approach because once nodes began to self-adjust and increase the number of subtasks that they needed, they also prefetch more subtasks from their parents. More prefetching by a node results in more subtasks being available for its children: subtasks quickly reach the nodes farthest from the root, thus improving ramp-up.

We also compared the running times of the three runs which are in Table 8. Interestingly, the run with the exponential self-adjustment performed poorly with respect to runs with linear or constant self-adjustment; this is due to nodes prefetching extremely large numbers of subtasks. Nodes now spend more time waiting

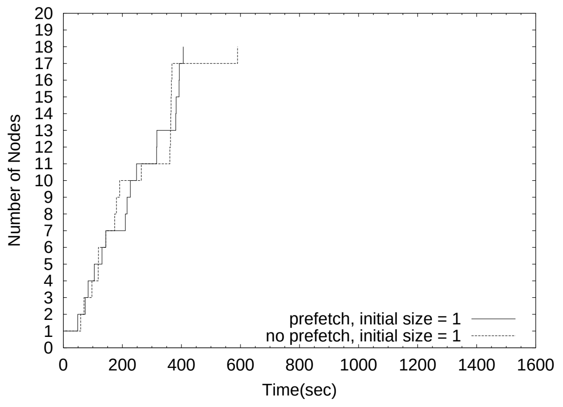

Figure 19: Effect of Prefetching on Data Ramp-up, Min. Number of Subtasks = 1

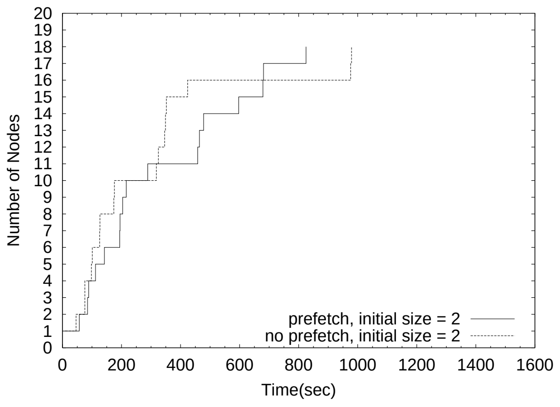

Figure 20: Effect of Prefetching on Data Ramp-up, Min. Number of Subtasks = 2

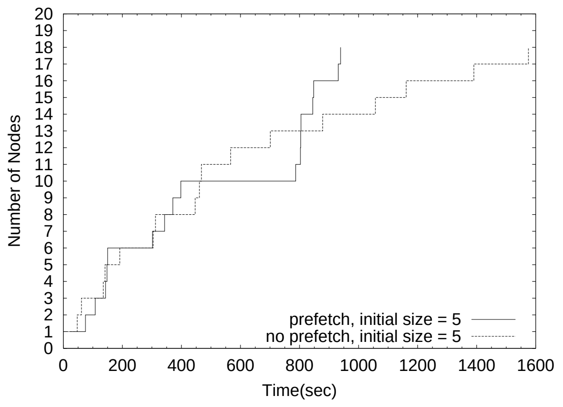

Figure 21: Effect of Prefetching on Data Ramp-up, Min. Number of Subtasks = 5

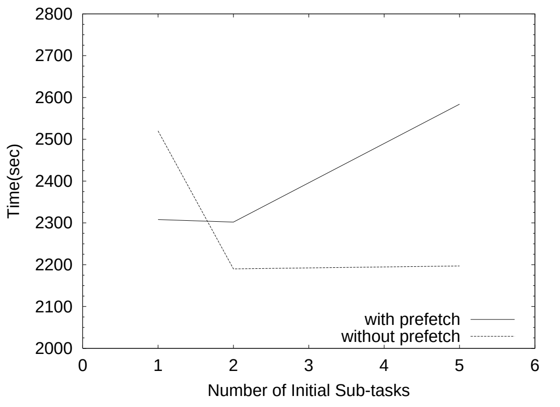

Figure 22: Effect of Prefetching and Min. Number of Subtasks on Running Time

for their requests to be satisfied, resulting in a degradation in the throughput at that node.

However, the difference in overall running time between the runs with linear and constant self-adjustment, is not very significant. We expect better performance in the linear case and it is our opinion that this difference would be more pronounced if the total number of subtasks to be computed were to be increased and the experiment run for a longer period of time.

4.6 Number of children

We experimented with different children-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 9 and Figure 24. 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 the synchronization delay at each node increases with the number of requesting children. Children have to wait for a longer time for their requests to be satisfied.

We actually expected a marked degradation in performance as the size of the children-list increases.

Initial Number of Subtasks	Time (sec)	Time (sec)
	Prefetching	No prefetching
1	406	590
3	825	979
5	939	1575

Table 6: Effect of Prefetching and Min. Number of Subtasks on Data Ramp-up

Initial Number of Subtasks	Time (sec)	Time (sec)
	Prefetching	No prefetching
1	2308	2520
2	2302	2190
5	2584	2197

Table 7: Effect of Prefetching and Min. Number of Subtasks on Running Time

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

Table 8: Effect of Self-adjustment function on Data Ramp-up Time and Running Time

Maximum Number of Children	Time (sec)
5	1068
10	760
20	778

Table 9: Effect of Max. Number of Children on Data Ramp-up

However, our current computational task does not run for enough time to be able to test this effect properly. The topology of the overlay network changes frequently before the nodes organize themselves well.

In order to obtain an idea of the severity of the synchronization delay, we ran two experiments: one with the good initial configuration of Figure 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 no change in their topology as the computation progressed, there was no impact of these changes on the overall running time. The effect of the synchronization delay was then clearly observed as in Table 10.

As expected, the root needed to satisfy a large number of requests in the latter case, which is why it took more time to fulfill any single request. This resulted in the running time with maximum child-list size set to 20 being 15 % slower than with the size set to 5. Similar results were observed even when the child-propagation mechanisms were turned off.

5 Conclusions and Future Work

Strong mobility is an important abstraction for developing desktop grid applications. We translate strongly mobile source code into weakly mobile source code by using a preprocessor. This allows us to build flexible, large-scale applications where the subtasks of a computation, in the form of multi-threaded mobile agents, utilize unused desktop resources. Different threads within these agents perform distributed computation and communication, and decentralized scheduling.

We have designed an autonomic scheduling algorithm for massively parallel applications where the data initially resided at one location, and whose subtasks have considerable data transfer times. The agents build a tree-structured overlay network with the data source at the root. 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 system performance. Our scheduling scheme has a number of mechanisms. We have conducted experiments on a set of machines distributed across Ohio, and have performed extensive analyses of the performance of the scheme's different mechanisms. These show that our approach is feasible in practice.

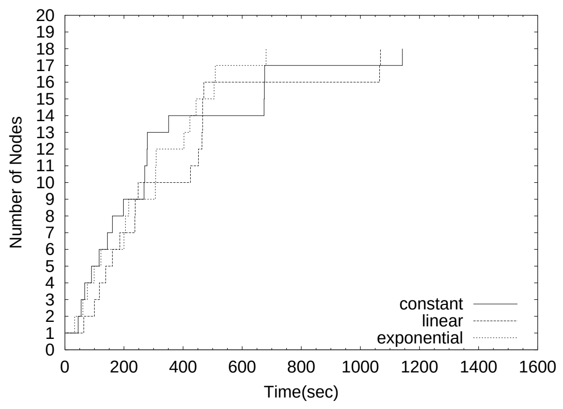

Figure 23: Effect of Self-adjustment function on Data Ramp-up Time

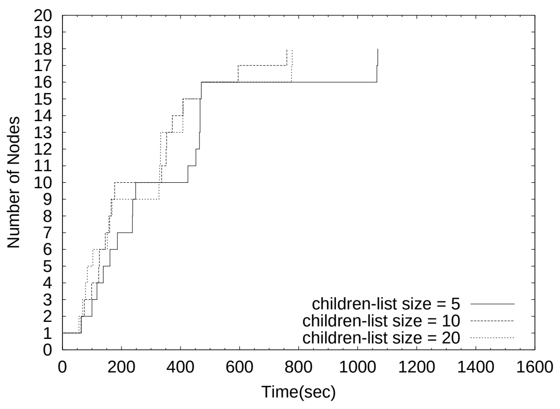

Figure 24: Effect of Max. Number of Children on Data Ramp-up

Maximum Number of Children	Time (sec)
5	1781
20	2041

Table 10: Effect of Max. Number of Children on Running Time, No Topological Variation

An important problem that we will address in future is the initial assignment of the friend-list. There has been some research in this direction [19, 50, 51], and we will consider how best to apply this to our own work.

An interesting aspect of the algorithm in [49] is that a node can prioritize its children according to their ability to communicate. A communication from a higher-priority child can interrupt that from one of lower priority. We will experiment with incorporating a similar mechanism into our scheme.

While this paper concentrated on a scheduling scheme for massively parallel applications where the data was initially all in one repository, we will experiment with autonomic schemes for a wide class of applications in future. It is our intention to present a desktop grid user with a simple software interface that will allow him/her to use one of a set of scheduling schemes, depending on the application characteristics.

The experiments that we ran for this paper were run on a set of 18 heterogeneous machines. We plan to harness the computing power of idle machines across the Internet — at the Ohio State University in particular — to create a desktop grid of a scale of the tens or hundreds of thousands. We will then allow researchers to deploy scientific applications on this system.

6 Acknowledgements

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

References

[1] J. A. H. John F. Shoch, “The ”worm” programs — early experience with a distributed computation,” *Communications of the ACM*, vol. 25, no. 3, Mar. 1982.

[2] M. Litzkow, M. Livny, and M. Mutka, “Condor — a hunter of idle workstations,” in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[3] G. Woltman. [Online]. Available: http://www.mersenne.org/prime.htm

[4] distributed.net. [Online]. Available: http://www.distributed.net

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

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

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

[8] grid.org. [Online]. Available: http://www.grid.org

[9] A. S. Grimshaw and W. A. Wulf, “The legion vision of a worldwide virtual computer,” *Communications of the ACM*, vol. 40, no. 1, Jan. 1997.

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

[11] D. Abramson, J. Giddy, and L. Kotler, “High performance parametric modeling with nimrod/g: Killer application for the global grid?” in *Proceedings of the International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000, pp. 520–528.

[12] H. Casanova and J. Dongarra, “Netsolve: A network enabled server, examples and users,” in *Proceedings of the Heterogeneous Computing Workshop*, Orlando, FL, 1998.

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

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

[15] G. T. et al., “Spatial patterns in ant colonies,” in *PNAS*, vol. 99, no. 15, 2002, pp. 9645–9649.

[16] 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*, Bologna, Italy, July 2002.

[17] A. Oram, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O’Reilly, 2001.

[18] C. Shirky, “What is P2P . . . and what isn’t?” *O’Reilly Network*, November 2000. [Online]. Available: http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html

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

[20] D. Kotz, R. Gray, and D. Rus, “Future directions for mobile-agent research,” *IEEE Distributed Systems Online*, vol. 3, no. 8, August 2002. [Online]. Available: http://dsonline.computer.org/0208/f/kot.htm

[21] D. B. Lange and M. Oshima, “Seven good reasons for mobile agents,” *Communications of the ACM*, vol. 42, no. 3, Mar. 1999.

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

[23] S. Bouchenak, D. Hagimont, S. Krakowiak, N. D. Palma, and F. Boyer, “Experiences implementing efficient Java thread serialization, mobility and persistence,” INRIA, Tech. Rep. RR-4662, December 2002.

[24] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, and R. Jeffers, “Strong mobility and fine-grained resource control in NOMADS,” in *Proceedings of the Second International Symposium on Agent Systems and Applications / Fourth International Symposium on Mobile Agents*, Zurich, Sept. 2000.

[25] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus, “D’Agents: Applications and performance of a mobile-agent system,” *Software—Practice and Experience*, vol. 32, no. 6, May 2002.

[26] A. Acharya, M. Ranganathan, and J. Saltz, “Sumatra: A language for resource-aware mobile programs,” in *Mobile Object Systems: Towards the Programmable Internet*, ser. Lecture Notes in Computer Science, J. Vitek, Ed., no. 1222. Springer-Verlag, 1996, pp. 111–130. [Online]. Available: http://www.cs.umd.edu/~acha/papers/lncs97-1.html

[27] T. Suezawa, “Persistent execution state of a Java virtual machine,” in *Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[28] H. Peine and T. Stolpmann, “The architecture of the Ara platform for mobile agents,” in *First International Workshop on Mobile Agents*, Berlin, Germany, Apr. 1997.

[29] T. Illmann, T. Krüger, F. Kargl, and M. Weber, “Transparent migration of mobile agents using the Java platform debugger architecture,” in *Proceedings of the 5th International Conference on Mobile Agents*, Atlanta, GA, December 2001.

[30] S. Fünfrocken, “Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs,” in *Proceedings of the Second International Workshop on Mobile Agents*, Stuttgart, Germany, September 1998.

[31] T. Sekiguchi, H. Masuhara, and A. Yonezawa, “A simple extension of Java language for controllable transparent migration and its portable implementation,” in *Coordination Models and Languages*, 1999.

[32] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, “Portable support for transparent thread migration in Java,” in *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents*, Zurich, Switzerland, September 2000.

[33] T. Sakamoto, T. Sekiguchi, and A. Yonezawa, “Bytecode transformation for portable thread migration in Java,” in *Proceedings of Agent Systems, Mobile Agents, and Applications*, 2000.

[34] A. J. Chakravarti, X. Wang, J. O. Hallstrom, and G. Baumgartner, “Implementation of strong mobility for multi-threaded agents in Java,” in *Proceedings fot 2003 International Conference on Parallel Processing*. IEEE Computer Society, Oct. 2003.

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

[36] X. Wang, “Translation from strong mobility to weak mobility for Java,” Master’s thesis, The Ohio State University, 2001.

[37] S. F. Hummel, J. P. Schmidt, R. N. Uma, and J. Wein, “Load-sharing in heterogeneous systems via weighted factoring,” in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 1996, pp. 318–328.

[38] T. Hagerup, “Allocating independent tasks to parallel processors: An experimental study,” *Journal of Parallel and Distributed Computing*, vol. 47, pp. 185–197, 1997.

[39] O. H. Ibarra and C. E. Kim, “Heuristic algorithms for scheduling independent tasks on non-identical processors,” *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[40] C. Kruskal and A. Weiss, “Allocating independent subtasks on parallel processors,” *IEEE Transactions on Software Engineering*, vol. 11, pp. 1001–1016, 1984.

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

[42] T. H. Tzen and L. M. Ni, “Trapezoidal self-scheduling: A practical scheme for parallel compilers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87–98, Jan. 1993.

[43] E. Heymann, M. A. Senar, E. Luque, and M. Livny, “Adaptive scheduling for master-worker applications on the computational grid,” in *Proceedings of the First International Workshop on Grid Computing*, 2000, pp. 214–227.

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

[45] H. James, K. Hawick, and P. Coddington, “Scheduling independent tasks on metacomputing systems,” in *Proceedings of Parallel and Distributed Computing Systems*, Fort Lauderdale, FL, Aug. 1999.

[46] 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*, June 2000, pp. 99–105.

[47] C. Leangsuksun, J. Potter, and S. Scott, “Dynamic task mapping algorithms for a distributed heterogeneous computing environment,” in *Proceedings of the Heterogeneous Computing Workshop*, Apr. 1995, pp. 30–34.

[48] R. Wolski, J. Plank, J. Brevik, and T. Bryan, “Analyzing market-based resource allocation strategies for the computational grid,” *International Journal of High-performance Computing Applications*, vol. 15, no. 3, 2001.

[49] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante, “Autonomous protocols for bandwidth-centric scheduling of independent-task applications,” in *17th International Parallel and Distributed Processing Symposium*, Nice, France, Apr. 2003, pp. 23–25.

[50] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content addressable network,” in *Proceedings of ACM SIGCOMM‘01*, 2001.

[51] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, San Diego, CA, 2001, pp. 149–160.

[52] B. L. A. S. Tool. [Online]. Available: http://www.ncbi.nlm.nih.gov/BLAST/