Phoenix : a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources

Kenjiro Taura
University of Tokyo
7-3-1 Hongo Bunkyo-ku
Tokyo, 113-0033, Japan
tau@logos.t.u-tokyo.ac.jp

Kenji Kaneda
University of Tokyo
7-3-1 Hongo Bunkyo-ku
Tokyo, 113-0033 Japan
kaneda@is.s.u-tokyo.ac.jp

Toshio Endo
PRESTO, JST
4-1-8 Honcho, Kawaguchi
Saitama, 332-0012 Japan
endo@logos.t.u-tokyo.ac.jp

Akinori Yonezawa
University of Tokyo
7-3-1 Hongo Bunkyo-ku
Tokyo, 113-0033 Japan
yonezawa@is.s.u-tokyo.ac.jp

ABSTRACT

This paper proposes Phoenix, a programming model for writing parallel and distributed applications that accommodate dynamically joining/leaving compute resources. In the proposed model, nodes involved in an application see a large and fixed *virtual node name space*. They communicate via messages, whose destinations are specified by virtual node names, rather than names bound to a physical resource. We describe Phoenix API and show how it allows a transparent migration of application states, as well as dynamically joining/leaving nodes as its by-product. We also demonstrate through several application studies that Phoenix model is close enough to regular message passing, thus it is a general programming model that facilitates porting many parallel applications/algorithms to more dynamic environments. Experimental results indicate applications that have a small task migration cost can quickly take advantage of dynamically joining resources using Phoenix. Divide-and-conquer algorithms written in Phoenix achieved a good speedup with a large number of nodes across multiple LANs (120 times speedup using 169 CPUs across three LANs). We believe Phoenix provides a useful programming abstraction and platform for emerging parallel applications that must be deployed across multiple LANs and/or shared clusters having dynamically varying resource conditions.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*concurrent programming*

PPoPP'03, June 11–13, 2003, San Diego, California, USA.

General Terms

Performance

Keywords

Parallel programming, distributed programming, message passing, migration, resource reconfiguration

1. INTRODUCTION

Distributed computing and parallel computing are converging. Parallel computing, traditionally focusing on solving a single problem with a fixed number of homogeneous and reliable processors, is now involving resources distributed over the world [11]. Issues of primary importance for distributed applications, such as some degree of fault-tolerance and continuous 24x7 operations are now issues parallel applications must face as a fact of the world. Distributed computing, traditionally focusing on sharing and exchanging "information" between computers, is now extending its purpose to providing more compute-intensive services with clusters and even to harnessing compute power distributed over the world [6, 25, 38]. Issues traditionally studied in the context of parallel computing, such as scalability of applications and ease of writing applications that involve many nodes, will become a major issue in developing and deploying such applications. In summary, many applications are becoming both "parallel" and "distributed" in the traditional sense. They are parallel in the sense that they need coordinate a large number of resources with a simple programming interface. They are distributed in the sense that they must be able to serve continuously for a long time in the face of dynamically joining/leaving resources and occasional failures. Such applications include data-intensive computing that operates on a stream of data produced every day [35], information retrieval systems (e.g., web crawlers and indexers) that cooperatively gather and index web pages without duplications [14], many Internet-scale computing projects to support scientific discoveries by harnessing a large number of compute-resources on the Internet [9, 13, 31], and task scheduling systems that distribute a large number of batch jobs to available resources [21, 26]. These applica-

tions are currently implemented directly on top of a very primitive programming model (socket in most cases). With these applications becoming more complex and the number of available resources becoming larger, however, they demand a simpler programming model, with much of the complexity coming from the size and dynamism of the underlying resources (nodes and network) being masked by the middleware.

This paper proposes a programming model Phoenix for such emerging applications. We describe its API, programming methodology, our current implementation, and performance study. It specifically has the following features.

- It is similar to and subsumes regular message passing models with fixed resources, so it facilitates porting existing parallel code to environments where resources are dynamic. Thanks to its similarity to message passing, Phoenix can also take advantage of a large body of work on algorithms, libraries and higher-level abstractions built on top of message passing models such as distributed shared memory [18, 43] and distributed garbage collections [20, 36, 39].

- It allows nodes to join and leave an application while it is running without relying on non-scalable notifications. More specifically, it defines a simple message delivery semantics with which the programmer can safely migrate application states from one node to another without stopping the application. Supporting joining/leaving nodes is a by-product of the transparent migration.

- It allows applications to be deployed across multiple LANs without demanding network configurations to change for them. With a few pieces of network configuration information, Phoenix runtime automatically connects participating resources together, and it does not assume a connection between every pair of nodes is allowed. Our current implementation also supports tunneling through SSH [40] without requiring individual application users to manually establish them outside the application.

- It allows scalable implementation that does not have an obvious bottleneck. While scalability depends on particular implementation, of course, the model itself does not imply much serialization.

In Section 2, we state requirements for supporting "parallel and distributed" applications in depth, and review existing programming models. Section 3 describes Phoenix model and its API. Section 4 demonstrates how to write parallel programs in Phoenix model, focusing on ramifications necessary to port existing message passing code to Phoenix. Section 5 briefs our current implementation. Section 6 reports preliminary performance evaluation of our implementation. Section 7 mentions related work and Section 8 states conclusions.

2. REVIEW OF EXISTING MODELS

2.1 Requirements

In this section, we argue that popular models for parallel and distributed programming are inadequate for supporting the types of applications motivated in the previous section. A programming model and its implementation for such applications should have the following features.

1. **Simplify programming with a large number of nodes.** It should specifically support a simple node name space with which the application can easily exchange messages, partition application state, and distribute tasks among nodes.

2. **Support dynamic join/leave of nodes.** It should allow an application to start without knowing in advance all nodes that may be involved in the application, and continue its operation when some nodes leave computation.

3. **Work under today's typical network configurations.** It should allow an application to be deployed under typical network configurations of today, which include firewalls, DHCP, and NAT. It should provide the application with a flat communication space despite that the underlying communication layer does not necessarily allow any-to-any direct communication.

4. **Allow scalable implementation.** All these features must be implementable in a scalable fashion. Semantics should not imply too much serialization.

Table 1 summarizes several programming models with respect to the above requirements.

2.2 Message Passing Models

Message passing models simplify writing applications involving a large number of nodes by providing a simple flat name space with which nodes can communicate with each other. When an application runs with N nodes, they are given names $0, \cdots, N-1$. A node can send a message simply by specifying its destination node number; it does not have to manage connections, remember topologies of them, nor keep track of participating nodes (simply because they are fixed). It also suits for scalable implementation because its fundamental operation, point-to-point message passing, can be mapped to a similar operation of the underlying hardware in a relatively straightforward manner, and it is scalable.

On the other hand, this model is weak in supporting joining/leaving nodes, because of the very nature that made parallel programming simple—the simple name space. Let us assume we try to involve a new node into computation. We would immediately face problems such as how to assign it a node number uniquely and scalably and how to notify participating nodes of the new node. A more complex problem arises when a node leaves because it would leave a gap in the node name space. Both PVM [1] and MPI version 2 [23] allow nodes to join, and PVM also allows nodes to leave. Above problems are, however, largely unsolved, and complications that arise are simply exposed to the programmer.

Typical operating environments of parallel programming models have been MPPs, clusters, and LANs. Therefore, except for a few recent implementations targeting wide area operations, they typically assume that connections between any pair of nodes are allowed. That is, nodes simply establish all connections when an application starts and assume they are never broken. When one is ever broken, an unrecoverable error is signaled.

	ease of programming with many processors	flexibility to various network configurations	provision for dynamically changing resources	scalability
Message Passing	strong	weak*	weak	strong
Sockets	weak	fair	fair	strong
RPC/RMI	weak	fair	fair	strong
Shared Memory	strong	weak*	weak	weak

Table 1: Comparison of popular existing programming models regarding various aspects for supporting parallel applications with dynamic resources. *Strong/fair/weak* respectively mean the *model* (not a particular implementation) is strong/fair/weak in the specific aspect. *Weak means the model is not inherently weak, but typical implementations are because the main target environment has not been wide-area.**

In summary, message passing models and their typical implementations are strong in the items #1 and #4 of the requirements mentioned in Section 2.1 but not in others.

2.3 Distributed Programming Models

The most widely used programming models for distributed computation will be socket, RPC, and more recently, distributed objects such as CORBA and Java RMI (remote method invocations). Built on top of them, common programming idioms such as client-server and its multi-tier variants have been successful. These models are designed on the assumption that (1) nodes start independently and "meet" at runtime, and (2) neither nodes nor network are completely reliable.

The simplest of them is the client-server model, where only the server is initially running. Clients (nodes) join computation by connecting to servers. An obvious, but interesting aspect of this model is that clients do not have to know each other's name. They only have to know the server's name. In other words, clients "do not care" each other so they can join and leave computation merely by notifying the server (e.g., closing the connection). Coordination and communication among clients may be indirectly achieved through the server.

Although this model is not usually seen as a parallel programming model, we can see its simple adaptation to parallel applications in many "task-scheduling" systems. In cluster environments, they include PBS [26] and LSF [21]. In wide area, they include Nimrod/G [4] and many Internet-scale computing projects [9, 13, 31]. They have a basic architecture in common; there is a single or a few task servers that pool tasks. Compute resources occasionally get tasks from the server. Tasks do not (and cannot) directly talk to each other. A very primitive form of coordination is implicitly achieved through the server that dispatches tasks without duplication and detects the completion of all tasks.

While this programming model naturally supports dynamic resources, it has an obvious scalability limitation for more communication intensive programs. A scalable cooperation among a large number of compute nodes requires each node know each other's name (communication end point) and communicate without going through a central server. Building such facilities on top of low-level APIs such as socket is very complex, however. To achieve the same level of simplicity as message passing models, the programmer must implement such low level functions as managing participating node names, maintaining connections, and routing messages.

In short, client-server models widely used in distributed computation satisfy items #2 and #3 reasonably well. Many tools are available that have stronger supports in some but not all aspects of them [32, 41]. They are generally weak in #1 and do not provide solutions to achieving both #1 and #4 at the same time.

2.4 Shared Memory

A large body of work studied software shared memory or shared object abstraction implemented on distributed memory machines [18, 43]. Although shared memory models are not commonly used in wide area settings, it has many interesting aspects that naturally solve some of the problems associated with message passing and client-server models. First, it naturally allows new processes (or threads) to join and leave computation without notifications between each other. It does so because threads do not normally communicate with each other through their names. They instead indirectly communicate using shared memory as "communication media." The address space, the set of available "names" used for communication, stays the same regardless of the number of threads. Moreover, communication occurs without an obvious bottleneck in the programming model level. This is in contrast with the client-server computing where natural dynamic join/leave of clients are achieved by mediating all communication/coordination through *a single* communication medium (i.e., the server name). The fundamental difference between the two is that the number of server names is one or a few, whereas the number of locations in an address space can be very large, so we can in principle distribute communication traffic among nodes in a scalable manner, provided that locations are evenly distributed across participating nodes.

The Phoenix model can be viewed as message passing, combined with this idea of having a name space much larger than the number of compute resources.

3. Phoenix MODEL

3.1 Basic Concepts

Like traditional message passing models, Phoenix provides the application with a flat and per application node name space, which is a range of integers, say $[0, \cdots, L)$. A node name specifies a message destination. Unlike regular message passing, however, L can be much larger than the number of participating nodes and must be *constant* regardless of it. We shortly comment on how L can be chosen and assume for now a value of L has been chosen. We call the space $[0, \cdots, L)$ *the virtual node name space* or *the virtual processor name space* of the application. Since the number of participating nodes does not match L, each

physical node assumes, or is responsible for, *a set of* virtual node names. We hereafter always use a term "virtual node" to mean a virtual node and use a term "node" only to mean a *physical* node. Given a message destined for a virtual node, the runtime system routes the message to the physical node that currently assumes the specified virtual node (`ph_send`/`ph_recv`). Phoenix allows the mapping between physical nodes and virtual nodes to change at runtime (`ph_assume`/`ph_release`). The entire virtual node space nevertheless stays constant. This is the fundamental mechanism with which Phoenix supports parallel applications that change the number of participating nodes at runtime, while providing the programmer with the simpler view of a fixed node name space.

Remark: The name virtual "node/processor" might suggest a model similar to SIMD where there are as many threads of control as the number of virtual nodes and the program specifies the action of each virtual node. This is not the case in Phoenix, where each physical node only has as many threads of control as explicitly created (usually one). Virtual node names are just names given to physical resources as a means to specify message destinations (communication end points).

L can be chosen for the application's convenience, as long as all participating nodes can agree on the same value. As we explain in Section 4, the primary purpose of virtual node names is to associate each piece of application data with a virtual node name, so that virtual $\leftrightarrow$ physical node mapping will derive data distribution. So a reasonable choice is often determined by the size of the application data to be distributed over nodes. For example, if the only distributed data structure used by the application is a hash table with N (constant) keys, we may have $L = N$ and associate hash items of key x with virtual node x. If there are many distributed data structures of different sizes or even unknown sizes, one can simply choose an integer much larger than any conceivable number of data items, say 2^{62}.

Underneath, Phoenix runtime system routes messages through the underlying communication layer. The application specifies connections it would like to establish (`ph_add_port`), and the runtime system automatically builds a graph of nodes to route messages.

As will be clear from the above description, physical nodes participating in a Phoenix application should cooperatively cover the entire virtual node space. More specifically, Phoenix applications should maintain the following conditions.

- No two nodes assume the same virtual node at any instant.
- There may be an instant at which no node assumes a virtual node, but in such cases, one must eventually appear that assumes it.

We hereafter call the above conditions *the disjoint-cover property*. The intent is to always maintain the invariant that the entire virtual node name space is disjointly covered by participating nodes. We however slightly relax this condition (the second bullet), allowing finite periods of time in which no physical node assumes a virtual node. Messages to such a virtual node are queued until one appears that assumes it, rather than lost, bounced, or redirected to a node in a way the programmer cannot predict. This is important for supporting applications that migrate application-level states from one node to another, and/or applications that allow nodes to permanently leave. We detail in Section 3.5 how to write such applications in Phoenix framework, and why this semantics is important.

3.2 A Note on Reliability Assumptions

The present paper does not address the issue of making applications tolerant to node crashes, and does assume nodes are reliable. For network failures, our routing infrastructure transparently masks disconnections. The runtime system tries to re-establish broken connections. Our current implementation however assumes disconnections do not lead to message loss. That is, connections do not break when messages are in transit.

Technically, the disjoint-cover property introduced above cannot be maintained if crashes occur, so in practice, crash-robust stateful applications must rely on some sort of application-level rollback recovery (checkpointing and/or message logging). Details of such mechanisms are subject of our future research and beyond the scope of the present paper.

3.3 Phoenix API and its Semantics

A slightly simplified description of Phoenix API is as follows. The actual implementation is slightly more verbose. We also assume the size of virtual node name space, L, have been chosen and agreed by all nodes that participate.

Message send/receive functions for sending and receiving messages.

1. `ph_send`(`ph_vp_t` v, `ph_msg_t` m);
2. `ph_msg_t` m = `ph_recv`();

Node name mapping functions for dynamically changing the mapping between physical nodes and virtual nodes.

1. `ph_assume`(`ph_vps_t` s);
2. `ph_release`(`ph_vps_t` s);

Initialize and finalize functions for opening/closing the underlying communication layer on top of which Phoenix provides the simpler name space and message delivery semantics.

1. `ph_initialize`(`ph_port_t` p, `ph_path_t` f);
2. `ph_add_port`(`ph_port_t` p);
3. `int` e = `ph_finalize`(`ph_path_t` f);
4. `ph_vp_t` r = `ph_resource_name`();

In the above, `ph_vp_t` is a type for a single virtual node name, `ph_vps_t` for a set of virtual node names, `ph_msg_t` for a single message, `ph_port_t` for an end point of the underlying communication layer such as TCP, and `ph_path_t` a path name.

`ph_send` and `ph_recv` have the obvious semantics. `ph_send`(v, m) sends message m to a *virtual* node v, that is, a physical node that is currently assuming virtual node v. `ph_recv`() waits for a message to arrive at the caller node and returns the received message. Note that it receives a message destined for any of the virtual nodes the caller node assumes at the point of the call. In other words, each node has only a single queue for incoming messages.

Like other message passing systems, Phoenix also supports non-blocking receive and tags with which the receiver can selectively receive messages of particular types, but they are omitted in this paper for the sake of simplicity.

Mapping between virtual and physical nodes are determined through `ph_assume` and `ph_release`. When a physical node P calls `ph_assume`(s), the Phoenix runtime system starts delivering messages destined for virtual nodes in s to P. In other words, this is the caller's declaration that it is ready for receiving messages destined for virtual nodes in s. `ph_release`(s) has the reverse effect. The system no longer delivers messages destined for s to the caller node.

`ph_resource_name`() returns an opaque virtual node name *outside* the virtual node name space. The resulting name is worldwide unique with high probability.[1] It is generated when a node brings up (i.e., calls `ph_initialize`) and returns the same value until it disconnects from the application (i.e., calls `ph_finalize` below), either temporarily or permanently. Besides regular messages whose destinations are in the virtual node name space, Phoenix also routes messages destined for such node names. In short, the node name returned by `ph_resource_name`() serves as the name bound to the physical node. Thus, we hereby call this name *resource name* of the node.

Applications should not use resource names for the application-level logic. They are only used for supporting migration. To see why resource names are necessary, consider how a fresh anonymous node could join an application. Such a node can first send a message to a random virtual node to ask any processor to share some of its virtual nodes. It clearly needs to receive a reply, for which its resource name is necessary because it obviously does not assume any virtual node yet. A similar situation occurs when a node permanently leaves an application, because such a node has no virtual node names either just before leaving.

`ph_initialize`(p, f) initializes the Phoenix runtime. Among other things, it opens a local end point of the underlying communication layer bound to name p. The underlying communication layer we currently support is socket implementing TCP, optionally tunneled through SSH. Parameter p therefore is a socket address (IP address + TCP port number). Parameter f is a file name from which logged messages are read (may be null, meaning there are no such messages). Details are described below with `ph_finalize`.

`ph_add_port`(p), where p is an end point name in the underlying communication layer, specifies "a neighborhood" in the underlying communication layer. The effect is to let the local runtime system try to maintain a connection from the local node to p. The runtime system may use the connection to route messages between any pair of nodes, not just between its end points. Thus, the programmer does not have to be aware of the topology of the underlying connections in most part of the application code. All the application programmer must guarantee is that all the participating nodes can form a single connected component with the specified connections. `ph_add_port` can be called as many times as necessary and at anytime, though typically used when an application brings up. Phoenix specifically tolerates end points that are actually disallowed to connect to. Such connections are simply unused.

A more convenient interface can be built on top that obtains necessary information from a configuration file (as in `machines` file in MPI implementations) or a configuration server. It is beyond the scope of this paper how such information is automatically obtained and/or kept up-to-date. Security (authentication in particular) is another interesting aspect beyond the scope of this paper.

`ph_finalize`(f) shuts down the runtime system and allows the node to leave computation, either temporarily or permanently. The node may be responsible for a set of virtual nodes, in which case this node or another with the same state and virtual nodes should join the computation later. Messages to such virtual nodes are queued until one shows up. It does not have to join the computation with the same underlying connections. It may specify another set of connections and may even have a different IP address.

The parameter f is a file name to which undelivered messages may be logged. Both messages destined for the caller node and those that should be routed to another may potentially be written. `ph_finalize` tries not to finish having messages of the latter type written to f, because doing so will block messages destined for virtual nodes that have nothing to do with the leaving node. This cannot be guaranteed with 100% certainty, however. For an extreme example, if the node (or a cluster of nodes including it) is completely disconnected from the rest of the nodes just before calling `ph_finalize`, it should leave rather than waiting for connections to become available again.

3.4 Temporal Disconnect and Re-connect

A node does not have to do anything particular when it leaves computation only temporarily. It simply calls `ph_finalize`. By temporarily leaving, we mean the node may disconnect from the network and/or possibly turn off, but it is going to join the computation again in future, having same state and assuming the same virtual nodes it was assuming at the point of the leave. While the node is absent, messages destined for it are queued.

When a temporarily leaving node joins again, it may connect to the network with a different address of the underlying communication layer (e.g., IP address). It may also specify different neighborhoods (via calling `ph_add_port`). Phoenix runtime system automatically routes queued messages to the node's new location.

From the application programmer's perspective, temporarily leaving nodes do not affect the semantics of message passing. Messages destined for a temporarily leaving node are perceived as experiencing a very long latency. The application logic does not have to change as long as it (or the user) tolerates such latencies.

3.5 Remapping and Migration Protocol

The disjoint-cover property introduced in Section 3.1 can be maintained in several ways. Most trivially, we can statically partition the space among participating nodes, assuming nodes are fixed throughout the entire computation. In this way, Phoenix trivially subsumes message passing models.

Building applications in which each node can autonomously decide to join and leave computation requires a dynamic protocol to maintain the property, of course. Specifically, we need a protocol in which one node can migrate all or a part of its assuming virtual nodes to another. Such a protocol must fulfill the following requirements.

[1] We assume it is in fact unique.

1. Upon migrating virtual nodes, nodes should be able to migrate application-level states in such a way that the migration becomes transparent from the nodes not involved in it.

2. Each node should be able to *autonomously* trigger migration. This is necessary to allow each resource to join and leave computation for its own convenience.

3. Each node should be able to trigger migration in both directions—from another node to it and vice versa. The former is necessary when a new node (which by definition does not have any virtual nodes assigned to it) joins a computation and the latter when a node permanently leaves a computation.

To motivate the first requirement, consider an application that partitions a large hash table (or any "container" data structure such as an array) among participating nodes. Such an application typically uses a simple mapping between hash keys to *virtual* nodes. Most simply, hash key k is mapped to virtual node k and lookup/update of an item with hash key k is destined for whichever node assumes virtual node k at that moment. This is essentially how all recent peer-to-peer information sharing systems are designed [19, 33, 30, 29, 42].

For such a mechanism to support transparent migration, we must guarantee that a node assuming virtual node k always has all valid items of hash key k. This requires nodes to migrate hash table items from one node to another upon migrating virtual nodes. The same situation arises in every application that partitions application-level states among nodes.

Maintaining such a property in the presence of leaving/joining nodes is not a trivial problem and whether previous systems address this issue is not clear from the literature. Phoenix API `ph_assume` and `ph_release`, equipped with the protocol outlined below, achieve the property.

The protocol actually guarantees a property stricter than the disjoint-cover. That is, it maintains that each node always assumes *a single interval* (a contiguous range of integers) of virtual nodes, rather than a general set. We believe it is typical for an application to assume a single or a few intervals for the sake of simplicity and worst case storage requirements.

In essence, the protocol is a mutual exclusion protocol in which a node locks two resources (itself and another node from/to which virtual nodes migrate) before a transaction. We avoid deadlocks by the usual technique that introduces a total order among resources and requires each node to lock resources following the total order (the smaller first). We use the node's resource name (returned by `ph_resource_name`) to define the total order, though it can be any unique number that persists throughout a single invocation (i.e., persists from a point a node joins a computation and to a point it temporarily or permanently disconnects).

After a node, say p, grabs the lock of itself and another node, say q, it migrates virtual nodes along with some application level states in either direction (from p to q or from q to p). In the former case, p first releases the migrating virtual nodes, say S, by calling `ph_release`(S). It then sends application level states as necessary to q. After all application states arrive at q, q assumes S by calling `ph_assume`(S). Meantime, messages to S are queued and Phoenix delivers

1: p : resource name of this node;
2: s : lock state of this node (`FREE` or `LOCKED`);
3: h : resource name of the node that currently holds this node's lock, if any
4: I ($= [a, b)$) : the interval of virtual nodes this node currently assumes
5: l : true if a lock attempt by this node is in progress;
6: V : constant representing the entire virtual node space

Figure 1: Variables used in the migration protocol.

7: /* executed whenever the node feels like migration (typically when it just joined or wants to leave) */
8: BEGIN_TRANSACTION() {
9: $s =$ `FREE` and $\neg l$ and $I = \emptyset \Rightarrow$
10: $v =$ any virtual node; `ph_send`(v, `query`(p));
11: $l =$ `true`;
12: $s =$ `FREE` and $\neg l$ and $a - 1 \in V \Rightarrow$
13: $v = a - 1$; `ph_send`(v, `query`(p));
14: $l =$ `true`;
15: $s =$ `FREE` and $\neg l$ and $b \in V \Rightarrow$
16: $v = b$; `ph_send`(v, `query`(p));
17: $l =$ `true`;
18: }

Figure 2: Code that triggers a migration. Called whenever a node feels like doing so.

messages for S neither to p nor q. In the latter case, p first sends a request to q to send virtual nodes and application states. The rest of the transaction is similar to the former case. In essence, we make the migration appear to be atomic from the application's point of view.

Figure 1, 2, and 3 outline the mutual exclusion protocol, from a point where a node decides to migrate some virtual nodes, up to points where two locks are successfully granted (lines 44 and 56) or the attempt fails because of conflicts (lines 22, 31, 39, 47, and 58). Each node tries to lock itself and a node that is adjacent to it in terms of their assuming virtual node ranges. That is, when a node p assumes range $[a, b)$, it tries to lock a node that assumes $a - 1$ or b. As an exception, when p currently assumes no nodes, it can lock an arbitrary node (Figure2).

The algorithm is written as a list of guarded commands of the form:

$$G \Rightarrow A$$

where G is a condition and A an action executed when G holds. A message is written in a form $m(a, b, \cdots)$ where m is a tag and $a, b, \cdots$ arguments. A special predicate "received($m(a, b, \cdots)$)" becomes true if the node receives a message of the form $m(a, b, \cdots)$.

The protocol is essentially a message passing implementation of the dining philosopher problem with deadlocks avoided by the total order among resources. The protocol is complicated by several facts, however.

- Each node does not know in advance the name of the resource it should lock, so it does not know which one (itself or the other resource) it should lock first. Therefore an additional message exchange to compare

```
19: /* message handlers for the locking protocol */
20: received(query(q)) ⇒
21:   if (p = q) {
22:     l = false;
23:   } else if (q < p) {
24:     /* the sender q must lock itself before p */
25:     ph_send(q, lock_you_first(p, I));
26:   } else if (p < q and s = FREE) {
27:     /* grant p's lock to q */
28:     s = LOCKED; h = q; ph_send(q, ok1(p, I));
29:   } else if (p < q and s = LOCKED) {
30:     /* say p is already locked */
31:     ph_send(q, fail1(p));
32:   }

33: received(lock_you_first(q, S)) ⇒
34:   if (s = FREE and (S and I are adjacent as intended)) {
35:     /* lock itself; go ahead to grab the second lock */
36:     s = LOCKED; h = p; ph_send(q, lock2(p, S));
37:   } else {
38:     /* the lock attempt failed */
39:     l = false;
40:   }

41: received(lock2(q, S)) ⇒
42:   if (s = FREE and I = S) {
43:     /* grant the lock to q */
44:     s = LOCKED; h = q; ph_send(q, ok2(p));
45:   } else {
46:     /* say p is already locked */
47:     ph_send(q, fail2(p));
48:   }

49: received(fail1(q)) ⇒ l = false;

50: received(fail2(q)) ⇒
51:   /* failed to get the remote lock. cleanup local state */
52:   l = false; s = FREE; h = NULL;

53: received(ok1(q, S)) ⇒
54:   if (s = FREE and (S and I are adjacent as intended)) {
55:     /* got the first lock remotely and now the second */
56:     s = LOCKED; h = p; l = false;
57:   } else {
58:     ph_send(q, cleanup(p));
59:   }

60: received(ok2(q)) ⇒
61:   /* got the second remote lock */
62:   l = false;

63: received(cleanup(q)) ⇒
64:   l = false; s = FREE;
65:   ph_send(q, cleanup_ack(p));

66: received(cleanup_ack(q)) ⇒ l = false;
```

Figure 3: Message handlers implementing migration protocol. Assume it is called whenever a message arrives. Multiple messages are handler in sequence and mutually excluded with code in Figure 2

resource names may be necessary before trying to acquire locks (query() message in Figure 2 and Figure 3). A node should not hold any lock during this exchange to avoid deadlocks.

- Intervals assumed by nodes may change during the above exchange, so a node, which was adjacent to node p when p first decides to lock it, may no longer be so when p knows its resource name. When this occurs, p must abort the transaction (lines 34, 42 and 54).

This protocol was modeled by a protocol description language Promela and verified via a model checker SPIN [16, 15]. Although complete verification was not achievable with a 10 nodes experiment (due to out of memory), more than 300M states were examined and no errors were found.[2] We also wrote a simple Phoenix program in which nodes randomly migrate their virtual nodes and occasionally leave and join, while the application forwards a message to a random virtual node forever. We confirmed that no application messages are lost.

3.6 Fresh Join and Permanent Leave

Given the protocol explained in the previous section, joining and (permanently) leaving a computation is simple. When a node attempts to leave a computation permanently, it should first evacuate its virtual nodes (and perhaps application states) by running the migration protocol explained in the previous section. When a fresh node joins a computation, it should first obtain some virtual nodes, again by running the migration protocol.

4. WRITING PARALLEL PROGRAMS IN Phoenix

4.1 Basic Concepts

Given the similarity between Phoenix and regular message passing models, it is not surprising that many parallel algorithms can be naturally expressed in Phoenix. There are ramifications, however, mainly coming from the fact that the size of the virtual node name space is much larger than the actual number of physical processors. This prohibits a naive use of $\Theta(P)$ operations where P is the number of physical processors. For example, sending a message to *all processors*, a commonly used idiom in message passing, has no obvious counterpart in Phoenix.

Yet, there is a method that can, often straightforwardly, obtain a Phoenix program from a regular message passing algorithm. The basic ideas are as follows.

Map Data ↔ Virtual Nodes: Associate each piece of application data with a *virtual* node name. This is analogous to what is usually referred to as "data partitioning" in regular message passing programs. Since the size of data is usually much larger than the number of processors, data is literally "partitioned" among processors. In Phoenix,virtual node name space can be arbitrarily large, so one may choose a mapping that is most natural for the application. For example, when we have an array of N elements distributed over

[2]We verified that, among other things, whenever an interval I migrates and arrives at q, I is in fact adjacent to the interval assumed by q.

nodes, we may fix the size of virtual node name space to N and associate array element i to virtual node i. The mapping between data and virtual nodes does not usually change unless the application logic specifically needs it. When the mapping between virtual nodes and physical nodes changes, application data must migrate too, to maintain the data $\leftrightarrow$ virtual node mapping.

Derive Communication: Interpret each message send in the message passing code as a message "to a piece of data," even though the program text specifies the message is to a physical node. Given this interpretation, Phoenix program can simply send a message to the virtual node that is associated with the piece of data. Suppose, for example, the original message passing code sends a message to a processor p, which upon a receipt of this message increments its local array element $a[x]$. Porting this to Phoenix involves a reasoning that this message, literally sent to processor p, is logically sent to an array element $a[x]$, or more specifically, to whichever processor owns the array element $a[x]$. With this reasoning, Phoenix program can simply send a message to a virtual node associated with $a[x]$.

Reduce Message Number: Programs straightforwardly obtained in this way may lead to too many fine grain messages. Suppose, for example, a "FORALL"-type data parallel operations on a distributed array. In regular message passing, starting such operations typically needs only a single command message per node. Upon receipt of a message, the receiver operates on *all* local elements. Applying the previous bullet to such data parallel operations would send a message to each array element, which is obviously too many. Regular message passing code effectively "combines" messages to many array elements into a single message, taking advantage of the knowledge that they are on a single physical node. Phoenix,starting from the assumption that the number of physical nodes is unknown, cannot use the idea as is. Section 4.3 explains a convention and a method to achieve a similar effect.

The rest of this section shows several case studies of applying the above basic ideas.

4.2 Parallel Divide-and-Conquer

Divide-and-conquer is a framework that solves a large problem by recursively dividing problems into smaller sub-problems and then combining these sub-results. In sequential programs, it is most naturally expressed by recursions. In parallel programs, each recursive call is replaced by a task creation so that sub-problem calls can be executed in parallel.

Lazy Task Creation (LTC) is a general and efficient method for executing parallel divide-and-conquer algorithms [2, 22]. It has been shown to achieve good performance both on shared memory and distributed memory machines [8, 12, 37]. It primarily needs two kinds of coordination/communication between nodes (processors), namely, load balancing and synchronization.

For load balancing, each node maintains its own *local* task deque (doubly-ended queue). When a processor creates a new task, it pushes the task to the head of its local deque and starts executing the new task immediately. As long as there is a task in its local deque, each processor executes the task at the head of it. When a processor's local deque becomes empty, it sends a task-steal request to a randomly chosen processor. The receiver will transfer the task at the *tail* of its deque to the requester (if there is one). This strategy effectively splits the entire task tree near its root and lets each processor traverse a sub-tree in depth-first fashion. In this way, LTC achieves a good load balance with a small amount of load-balancing traffic. Note that each processor maintains its local task deque even on shared memory machines, rather than sharing a single global queue, to achieve a good scalability. It is therefore difficult to implement this method with a client-server model. Also note that this makes dynamic join/leave of processors non-trivial because, in effect, each processor's name is exposed to all participating processors for task-stealing. More specifically, there is a race condition between a leaving processor and another processor that is trying to send a request to it.

For synchronization, a task needs to wait for completion of its child tasks and obtain sub-results at some point. On shared memory machines, this is naturally achieved by maintaining a synchronization data structure that matches a value from the child and the waiting task (i.e., parent). On distributed memory machines, they should be given a globally unique name, which usually consists of a processor number and a local identifier valid in the processor. Putting a value or waiting on the synchronization data structure involves sending a message to the processor known from its global identifier.

Porting to Phoenix is relatively natural and straightforward. For load-balancing, when a node finds its task deque empty, it sends a task-steal request to a randomly chosen *virtual* node. In Phoenix,the message is guaranteed to be received by a physical node that assumes the virtual node. In essence, we resolved the race condition between a task-steal message and the leaving processor by the migration protocol described in Section 3.5. Once such a protocol is implemented correctly, the other part of the application can assume such a request will always reach a working (not leaving) processor.

For synchronization data structure, we merely represent a global identifier as a virtual node number + local identifier. The size of the virtual node name space can be chosen so that a global identifier fits into a single machine word or two. For example, if we would like to make global identifier 32 bits long, we may fix the virtual node name space to, say, $[0, 2^{12})$ and let the length of the local part 20 $(= 32-12)$ bits. When a node allocates a new identifier, it should allocate one that has a virtual node number part assigned to it.

Overall, there is almost no fundamental difference between regular message passing code and Phoenix code except the logic for joining/leaving processors. We have written a simple parallel binary tree creation as a template of divide-and-conquer and a parallel ray-tracing as a more realistic application.

4.3 Array-based Applications

In the previous section, we have seen that random the load balancing logic of LTC can be naturally written in Phoenix,with additional benefits of safely supporting dynamically joining/leaving processors. This was so simple

largely because LTC, either in message passing or in shared memory, does not use the value of processor identifiers in any significant ways; it only assumes that each identifier corresponds to a processor.

It is trickier to write parallel algorithms that would use the number of processors and processor identifiers in more significant purposes, such as data partitioning and/or load balancing. Such techniques are common in regular scientific programs that use arrays and/or static load balancing. For example, it is common in many scientific computations to block- or cyclic-partition an array. Given an array of N-elements, a block-partitioning assigns section $[i\frac{N}{P}, (i+1)\frac{N}{P})$ to processor i. Every node knows this assignment and may take advantage of it to optimize communication. It is also common in parallel computation to send messages to *all* processors. It is not trivial to write such operations in Phoenix where the application does not see the notion of physical processors but only see a large virtual node space. Sending a message to each virtual node is clearly not a solution, so we must devise some way to achieve the same effect. Note that this problem is not an artifact of the Phoenix model, but arises whenever we would like to support processors that dynamically join and leave. As a matter of fact, the definition of "broadcast," for example, is not clear when participating processors dynamically change. In this setting, we must try to send a message *not to all physical processors,* but to whoever needs the message to accomplish the task of the algorithm. Phoenix virtual node name space mediates them.

In this section, we study two array-based applications, integer sort (IS) in NAS Parallel Benchmark [24] and an LU factorization algorithm. Throughout the section, we assume each processor assumes a single range of virtual nodes, and the size of the range is maintained roughly the same. Given this assumption, we focus on how to achieve load balancing and small communication overhead close to those of message passing with fixed processors.

4.3.1 Integer Sort

Integer Sort in NAS Parallel Benchmark suite uses bucket sort algorithm. Given an array A of N elements, each element of which is in a range $[0, M)$, it divides the range into L sub-ranges $R_0 = [0, M/L)$, $R_1 = [M/L, 2M/L)$, $\cdots$, $R_{L-1} = [(L-1)M/L, M)$. The array is block-partitioned, and each processor first counts the number of elements in its local sub-array that fall into each sub-range. The set of elements in a sub-range R_j is called *a bucket* j and denoted as B_j. After counting the number of local elements in each bucket, each processor broadcasts the counts to all other processors and receives counts from them. Now all processors know how many elements of the entire array are in each bucket and determine which buckets each processor should sort. In regular message passing code with P processors, processor 0 will sort the buckets that roughly cover the smallest N/P elements, processor 1 the buckets for the next N/P elements, and so on. Based on the assignment determined this way, each processor sends each of its local buckets to the appropriate processor. It uses `MPI_Alltoall()` to distribute bucket counts and `MPI_Alltoallv()` to distribute bucket elements.

Below we focus on how to derive a Phoenix program that counts the number of elements in each bucket and exchanges array data. Primary data structures used in this program is an N-elements array A to be sorted and an L-elements array B that counts elements in each bucket. They are mapped to virtual nodes as follows.

$$v_A(i) = \frac{|V|}{N} i,$$

and

$$v_B(j) = \frac{|V|}{L} j,$$

where V refers the virtual node space of an arbitrary size, $v_A(i)$ the virtual node associated with $A[i]$, and $v_B(j)$ the virtual node associated with $B[i]$. Whichever processor assumes $v_A(i)$ is responsible for determining the bucket for $A[i]$ and whichever processor assumes $v_B(j)$ for summing up the number of elements in bucket j. Thus, the elementary task during the counting phase is that a node assuming $v_A(i)$ reads $A[i]$, determines its bucket j, and send an increment message to $v_B(j)$.

Literally implementing the above task leads to too many fine-grain messages. Thus we need devise a way to effectively "combine" these messages. The first trivial optimization is to let each processor scan all local elements, accumulate the counts in a local array, and then sending the local count for bucket j to virtual node j. This still requires L messages to be sent from each processor, which is much larger than the optimal P. Here we introduce a powerful combining technique that frequently occurs in Phoenix programming. The problem can be formulated as follows.

> Given a range $I = [p, q)$ and a function $f : I \to V$, send at least one message to all processors that assume virtual node $f(j)$ for any $j \in I$. We would like to do this without sending $q - p$ separate messages.

In our current problem, $I = [0, L)$ and $f(j) = v_B(j)$. That is, each node likes to send its local counts to all processors responsible for summing up a bucket count.

We can accomplish this as follows.

- The sender attaches the range $I = [p, q)$ with the message content and sends it to virtual node $f(p)$.
- Upon receiving a message with range $I = [p, q)$ attached, it removes from I all elements t such that it assumes $f(t)$. The remaining elements are formed into several (more than one) ranges and each range is forwarded to an appropriate virtual node, in the above manner.

The number of messages is reduced if the receiving node happens to assume many virtual nodes $\in f(I)$. This will be the case if f is a "slowly increasing" function in the sense that $f(i)$ and $f(i+1)$ are likely to be in a range covered by a single physical processor. It is particularly the case in our current problem. We call this technique *the multicasting idiom*.

For exchanging array elements, the basic task is to send all elements in B_j to virtual node $v_B(j)$. Each processor can either send a separate message for each bucket (i.e., send local elements in B_j to virtual node $v_B(j)$), or merge some number of consecutive buckets and multicast them to the corresponding processors. For example, if a processor merges buckets $B_3, B_4, \cdots, B_{10}$ into a single message, this

```
1:  for (k = 0; k < N; k++) {
2:    for (j = k; j < N; j++)
3:      A_kj = A_kj / A_kk;
4:    for (i = k + 1; i < N; i++) {
5:      for (j = k + 1; j < N; j++) {
6:        A_ij -= A_ik A_kj; }}}
```

Figure 4: LU Factorization

will be multicast to processors corresponding to these buckets, using the multicasting idiom just introduced (i.e., let $I = [3, 10)$ and $f = v_B$).

If an individual message is sent for each bucket, there is no risk of consuming bandwidth with uselessly large messages, but the number of messages becomes large (LP instead of the optimal P^2 in case we know the number of physical processors and virtual node assignments). At the other extreme, if we merge all buckets from a processor into a single message, the number of messages becomes small, while a single bucket will be, on average, forwarded $O(\log P)$ times along the multicasting tree. Optimal or near-optimal merging strategy is inevitably based on an estimate of the typical number of physical processors.

4.3.2 LU Factorization

Figure 4 shows the pseudo code for sequential LU factorization of $N \times N$ matrix A. Consider the kth iteration of the outermost loop. It updates $(n-k-1) \times (n-k-1)$ sub-matrix $(A_{ij})_{k+1 \le i,j < N}$. An element A_{ij} is updated using the value of A_{ik} and A_{kj} (line 6). Hence under a given partitioning of matrix A, the value of A_{ik} need be sent to all other processors that have some elements in the same row, and the value of A_{kj} to all other processors that have some elements in the same column.

Block partitioning gives a poor load balancing especially in later iterations, so a block-cyclic partitioning (with a constant block size) is commonly used in message passing models. We used blocks of 64×64 elements, but we assume in the sequel 1×1 blocks purely for the sake of exposition. This pure cyclic partitioning simply represents the matrix as a distributed one-dimensional array a of N^2 elements, and maps $a[i]$ to processor ($i \mod P$) where P is the number of physical processors. We would like to find a Phoenix analogue of this partitioning.

Let us assume that the size of virtual node space, $|V|$, is N^2. This is in fact a reasonable choice in this algorithm because there are no other data structures that must be partitioned among processors. Simply associating $a[i]$ with *virtual* node ($i \mod |V|$) clearly does not work. This does not serve the very purpose of cyclic partitioning—load balance. A simple alternative we propose is to associate $a[i]$ with virtual node:

$$v(i) = Ki \mod |V|,$$

where a K is a large integer relatively prime to $|V|$. K should be a large estimate of the number of physical processors we like to support. The intent is to let the sequence $v(0), v(1), v(2), \cdots, v(N^2-1)$ cycle through space V approximately K times. Thus each one Kth portion of the above sequence is evenly distributed over V. This will effectively assign roughly the same number of elements from each one Kth of the entire array to each physical processor. With K chosen reasonably large, we achieve the effect of cyclic partitioning.

For communication, the multicasting idiom used in the Integer Sort applies. Recall that in the kth iteration of the LU factorization, the value A_{ik} (and A_{kj}, for that matter) needs to be sent to the node that performs the update $A_{ij} -= A_{ik}A_{kj}$, which is the processor responsible for A_{ij}. Assuming the row major representation of matrices where a matrix element A_{ij} is stored as an array element $a[Ni + j]$, we can state the necessary communication in terms of our multicasting idiom as follows.

- For A_{ik}, let $I = [k+1, N)$ and $f(s) = v(Ni+s)$ (That is, send A_{ik} to all processors responsible for any index of the form $Ni + s$).
- For A_{kj}, let $I = [k+1, N)$ and $f(t) = v(Nt+j)$ (That is, send A_{kj} to all processors responsible for any index of the form $Nt + j$).

5. OVERVIEW OF IMPLEMENTATION

The heart of the Phoenix runtime system is its routing infrastructure. Like other peer-to-peer information sharing systems, it builds an overlay network among participating nodes and routes messages via the overlay. We currently use either regular TCP connection where allowed and SSH connection over TCP where only SSH access is allowed. Unlike many other similar systems, we do not assume the underlying network allows any-to-any connection. We believe this is an important property for deploying applications across multiple clusters, where there are many reasons why a connection may be impossible or cumbersome to establish. They include firewalls, DHCP (which makes connections to them cumbersome), and NAT (which makes connections to them from outside the LAN impossible). Resources subject to one of the above restrictions constitute a large portion of the available resources in today's typical environment, thus we need a protocol that builds and maintains a routing table over an arbitrary set of possible connections.

Elsewhere, we have proposed one such protocol in a similar context [17]. It builds a spanning tree among participating nodes. While it is simple, it unfortunately does not use available bandwidth effectively. We need a protocol in which nodes establish allowed connections more aggressively and select a short route to the destination. It is technically close to routing table construction problems studied in the context of IP routing [5] and mobile ad-hoc network routing [27].

Among many proposed routing table construction protocols, we currently employ the destination-sequenced distance-vector routing (DSDV) [27] originally proposed in the context of mobile ad-hoc networks. It was chosen because it consumes a relatively small amount of memory compared to other schemes based on distance-vector and is relatively simple to implement.

In summary, each node tries to connect to ports specified by `ph_add_port`. Connections that cannot be established are retried in a fixed interval. Along with maintaining connections, nodes cooperatively construct a routing table by exchanging routing information. Each node announces *virtual* nodes it assumes and announcements are propagated to other nodes. We will publish implementation details in a separate paper.

	Nodes	CPU	Number of CPUs	Interconnect
Cluster A	SunBlade 1000 Cluster	UltraSPARC 750MHz	2CPU × 16 nodes	100Mbps switched Ethernet
SMP B	SunFire15K SMP	UltraSPARC 900MHz	72CPU	shared memory
Cluster C	SunBlade 1000 Cluster	UltraSPARC 750MHz	2CPU × 128 nodes	100Mbps switched Ethernet

∗ Throughput between these three systems are 30-60Mbps over SSH.

Table 2: Environments

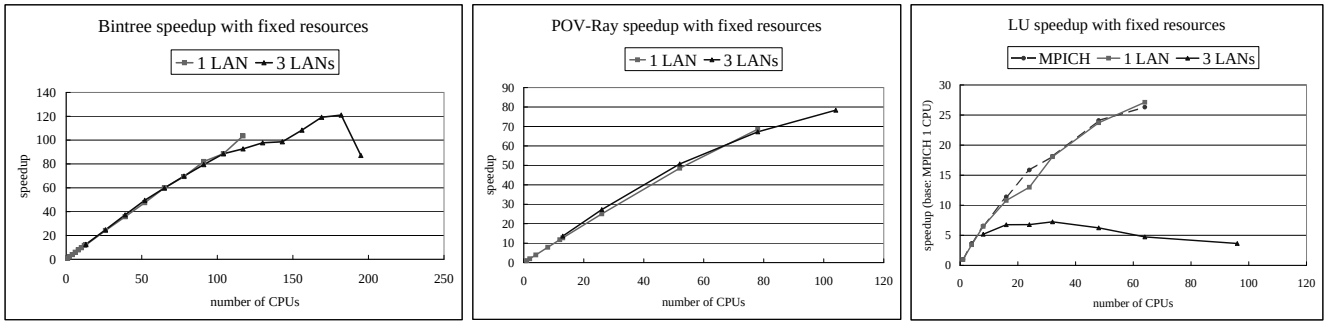

Figure 5: Speedup with Fixed Processors

6. PERFORMANCE EVALUATION

6.1 Programs and Platforms

We studied performance of three applications, Bintree, POV-Ray, and LU. Bintree is a binary task creation which serves as a template of many divide-and-conquer algorithms. It uses the algorithm described in Section 4.2. POV-Ray is a parallelization of a popular ray tracing program [28]. It also uses a divide-and-conquer algorithm. LU is an LU factorization algorithm written by us. We also implemented IS described Section4.3.1 both in Phoenix and MPI, but its scalability was poor in both platforms. This is because IS is very communication intensive (the amount of data exchange per processor is $O(N/P)$ whereas the time for local sort is $O(N/P \log(N/P))$). So we do not show its performance any further.

We used two cluster systems and a large SMP, each located in a separate local area network, summarized in Table 2. Nodes within a cluster are connected via 100Mbps switches. Only SSH connections are allowed across LANs. The raw TCP bandwidth (measured by the bandwidth of a large http GET request) between two LANs is approximately 100Mbps, but the actual throughput over SSH is 30-60Mbps. This is clearly a bottleneck for many parallel programs that would scale well within a LAN.

6.2 Results

Figure 5 shows speedup with fixed resources. We measured speedup both in a single cluster (Cluster C) and across the three SMP/clusters in Table 2. We used only one CPU for each node of the two clusters because they are shared by many users and many nodes were in fact occupied by one CPU-bound process. For the multi-LANs experiments, we mix CPUs from the three systems in a constant ratio (1:2:5 for LU and 1:4:8 for Bintree and POV-Ray).

Bintree makes a binary tree of depth 27 (2^{27} leaf nodes), taking approximately one hour on a single CPU. POV-Ray draws a picture of 8000 by 250 pixels, taking approximately two hours on a single CPU. Matrix size for LU is 8192 by 8192, taking approximately thirty minutes on a single CPU. For LU, we also wrote the same algorithm with MPI (MPICH) and show speedup relative to MPI performance on one CPU. Both POV-Ray and Bintree exhibit good speedups and LU was comparable to MPICH performance. We also confirmed that the basic messaging performance of our current Phoenix implementation was comparable or slightly better than MPICH.

Bintree and POV-Ray scale well across LANs, showing LTC is in fact a very communication-efficient load balancing scheme. On the other hand, LU performs poorly when deployed across LANs. This is of no surprise because LU is more communication-intensive and latency-sensitive than the other two.

Figure 6 demonstrates Phoenix's capability of dynamically adding/removing nodes. We begin with a small number of nodes, add one node at a time in a regular interval, up to 64 nodes. After running with the 64 nodes for a while, then we remove one node at a time. For each application, we defined a unit progress and measured the number of unit progresses made in each second. LU defined the unit progress as a completed floating point operation and POV-Ray as a line of the picture whose image has been calculated. Bintree defined it as any of the following event: task creation, synchronization, or a task completion. The graphs show "speedup" of each second, which is the number of unit progresses made in the second, divided by the number of progresses per second in a single node run. Also shown as "fixed" is the speedup obtained in the fixed-resource, single LAN experiment for the number of processors participating at that moment.

For all applications, a newly added node sends a join request to a randomly chosen virtual node name. Whichever physical node received the request splits its range of assuming virtual nodes into two equal ranges and gives the latter half to the requester. The graphs show Bintree and POV-Ray take advantage of dynamically added nodes very quickly. This is not surprising because they use dynamic load balancing and have relatively small application-level

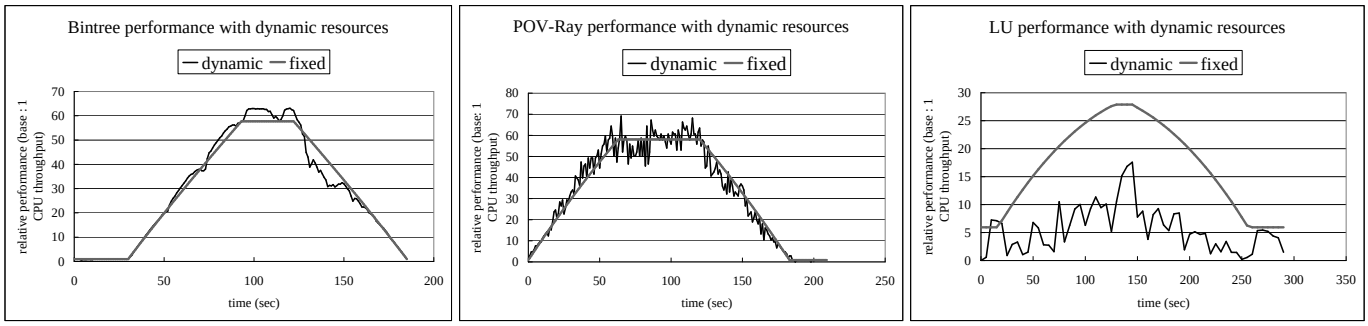

Figure 6: Dynamic Performance Improvement as Processors Join/Leave

states that need migrate or be copied to new nodes. For Bintree and POV-Ray, newly added nodes simply begin with an empty task deque, so tasks need not migrate at all when a node joins. There may be synchronization data structures that need migrate, but in this experiment it did not happen. On the other hand, LU exhibits a less encouraging behavior. First, it is not able to reach a peak performance comparable to the fixed-resources case after all nodes joined. Although we did not conduct detailed analyses, it is very conceivable that the above random partitioning of virtual node name space, and hence of the array, will lead to a load imbalance, where highly loaded nodes assume twice or more elements than others. Second, it exhibits significant performance drops many times on the way. This is because LU needs to move a large volume of array data when nodes join, especially when the number of processors are small. For example, when we have eight nodes, each node on average holds one eighth of the entire array, which is 64MB in our experiment. When a half of it should move to a new node, it takes at least 2.5 seconds on 100Mbps link. The data are sent in a single large message and the receiver does not make any progress during receiving the message. Moreover it blocks barrier synchronization performed at each iteration.

7. RELATED WORK

7.1 Wide-Area Parallel Computing Tools

Attempts to support parallel computations that run in wide area and/or for a long time are roughly classified into two groups.

- Attempts to take an existing programming model and make its implementation more suitable for wide-area/long-running computation.
- Attempts to define a simple, and rather restricted, computation model (task farming) to support dynamically joining/leaving resources and fault tolerance.

The first category includes MPI implementation supporting communication across LANs [10], accommodating node failures [7], and supporting checkpoints [3, 34]. Our main contribution, in contrast to these efforts, is that we proposed a new programming *model*, necessary for applications that acquire and release resources during computation. As we argued in Section 2, existing message passing *models*, not their particular implementation, have inherent difficulties in supporting applications using dynamic resources. We also sketched an implementation of Phoenix that supports communication across LANs over many restrictions (firewall, DHCP, NAT, etc.). MPI implementations should be able to take advantage of it.

The second category includes Nimrod/G [4] and many other task scheduling tools [21, 26]. There is a single or a few servers that pool tasks and whichever resources are live at that moment get a task from the servers, computes the result, and returns it to the server. No communication between resources is supported (they only communicate with the servers). This model naturally supports dynamically joining/leaving resources and tolerates crash of clients (not a server). So these tools have been very successful for many CPU-crunching experiments [9, 13, 31]. Our Phoenix model is an attempt to supporting dynamic resources in a more general computation model where involved resources directly and frequently communicate with each other. We believe this is becoming more and more important in near future environments where (1) wide-area bandwidth will increase to a point where complex coordination in wide-area becomes feasible in terms of bandwidth, and (2) many more applications will emerge and take advantage of the wide-area bandwidth to achieve a shorter turn around time for humans involved in an experiment. Phoenix will help such applications parallelize a single large task using more complex coordination between resources.

7.2 Peer-to-Peer Information Sharing Systems

Both models and implementation of Phoenix share many things in common with recent efforts on scalable peer-to-peer information sharing systems, such as Pastry [30], Tapestry [42], Chord [33], and CAN [29]. They are all based on a large and fixed name space abstraction mediating communication. They all build a routing infrastructure so that involved nodes can send messages to any name ("key" in terminology of peer-to-peer information sharing systems and "virtual node" in ours). The main differences are as follows.

- We discussed in depth how to support transparent migration in this setting. As we argued in Section 3.5, it is far from trivial, yet stateful applications (including most interesting parallel applications) need it. To the author's best knowledge, previous systems have not addressed this issue.
- Our routing infrastructure does not assume any-to-any connection is allowed. Previous systems enforce a predefined connection topology over involved resources, and seem to assume such connections are always al-

lowed (i.e., they are never blocked by firewall, the target is never behind a NAT router, etc.). Our routing infrastructure overcomes them using dynamic routing table construction, as outlined in Section 5.

8. CONCLUSION AND FUTURE WORK

We described Phoenix parallel programming model for supporting parallel computation using dynamically joining/leaving resources. Every node sees a large virtual node space. A message is destined for a virtual node in the space and whichever node assumes the virtual node at that moment receives it. A protocol to transparently migrate responsibility of virtual nodes and application states in sync has been clarified. This is the key step forward to supporting dynamically joining/leaving resources without making the programming model perceived by the programmer too complex or too restrictive. Several application studies have been described, demonstrating that this model is a general model that will facilitate porting many of existing parallel applications and algorithms to more dynamic environments. An implementation has been sketched, demonstrating a scalable implementation is indeed possible. Experiments have shown it achieves good speedups for divide-and-conquer applications having good locality, and takes advantage of dynamically joining resources for applications having small task migration overheads.

The next logical steps include a detailed study of the routing infrastructure, rollback mechanisms for fault tolerant applications, and higher-level programming models built on top of Phoenix.

Acknowledgments

We are grateful to anonymous reviewers for their constructive criticisms. We also wish to thank Takashi Chikayama, Andrew Chien, and members of Ninf project for their discussions and comments before publication. This work is financially supported by "Precursory Research for Embryonic Science and Technology" of Japan Science and Technology Corporation and "Grand-in-Aid for Scientific Research" of Japan Society for the Promotion of Science.

9. REFERENCES

[1] A. Beguelin and J. Dongarra. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *IEEE FOCS*, pages 356–368, 1994.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *SC 2002*, 2002.

[4] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture of a resource management and scheduling system in a global computational Grid. In *HPC Asia 2000*, pages 283–289, 2000.

[5] C. Cheng, R. Riley, and S. Kumar. A loop-free extended bellman-ford routing protocol without bouncing effect. In *ACM SIGCOMM*, pages 224–236, 1989.

[6] Entropia. http://www.entropia.com/.

[7] G. E. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *PVM/MPI 2000*, volume 1908 of *LNCS*, pages 346–353. Springer, 2000.

[8] M. Feeley. A message passing implementation of lazy task creation. In *Parallel Symbolic Computing: Languages, Systems, and Applications*, volume 748 of *LNCS*, pages 94–107. Springer-Verlag, 1993.

[9] fightAIDS@home. http://www.fightaidsathome.org/.

[10] I. Foster and N. Karonis. A Grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *SC 1998*, 1998.

[11] I. Foster and C. Kesselman, editors. *The GRID—Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM PLDI*, 1998.

[13] GIMPS. http://www.mersenne.org/prime.htm.

[14] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, December 1999.

[15] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[16] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[17] K. Kaneda, K. Taura, and A. Yonezawa. Virtual Private Grid : A command shell for utilizing hundreds of machines efficiently. In *CCGrid 2002*, 2002.

[18] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ACM ISCA*, 1992.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[20] B. Lang, C. Queinnec, and J. Piquer. Garbage collecting the world. In *ACM POPL*, pages 39–58, 1992.

[21] LSF - load sharing facility. http://wwwinfo.cern.ch/pdp/lsf/.

[22] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy Task Creation: A techinque for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[23] MPICH-a portable implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/.

[24] NAS parallel benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[25] Parabon computation inc. http://www.parabon.com/.

[26] Portable Batch System. http://www.openpbs.org/.

[27] C. Perkins. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM' 94 Conference on Communications Architectures Protocols and Applications*, 1994.

[28] POV-Ray. http://www.povray.org/.

[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, 2001.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

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

[32] SOCKS. http://www.socks.permeo.com/.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[34] Y. Takamiya and S. Matsuoka. Towards MPI with user-transparent fault tolerance. In *JSPP2002*, pages 217–224, 2002. (in Japanese).

[35] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *IEEE CCGrid*, pages 102–110, 2002.

[36] K. Taura and A. Yonezawa. An effective garbage collection strategy for parallel programming languages on large scale distributed-memory machines. In *ACM PPoPP*, pages 264–275, 1997.

[37] K. Taura and A. Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *ACM PPoPP*, 1999.

[38] United devices. http://www.ud.com/home.htm.

[39] H. Yamamoto, K. Taura, and A. Yonezawa. Comparing reference counting and global mark-and-sweep on parallel computers. In *LCR98*, volume 1511 of *LNCS*, pages 205–218, 1998.

[40] T. Ylönen. SSH – secure login connections over the internet. In *the Sixth USENIX Security Symposium*, 1996.

[41] V. C. Zandy and B. P. Miller. Reliable network connections. In *ACM MobiCom 2002*, 2002.

[42] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report Technical Report UCB//CSD-01-1141, University of California Berkeley, April 2000.

[43] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *ACM OSDI*, 1996.