Lilith: A Software Framework for the Rapid Development of Scalable Tools for Distributed Computing

A.C. Gentile, D.A. Evensky, and R.C. Armstrong
< gentile, evensky, rob>@ca.sandia.gov
Sandia National Laboratories
Livermore CA

Abstract

Lilith is a general purpose tool that provides a highly scalable, easy distribution of user code across a heterogeneous computing platform. By handling the details of code distribution and communication, such a framework allows for the rapid development of tools for the use and management of large distributed systems. This speed-up in development not only enables the easy creation of tools *as needed* but also facilitates the ultimate development of more refined, hard-coded tools as well. Lilith is written in Java, providing platform independence and further facilitating rapid tool development through Object reuse and ease of development. We present the user-involved objects in the Lilith Distributed Object System and the Lilith User API. We present an example of tool development, illustrating the user calls, and present results demonstrating Lilith's scalablility.

1. Introduction

Lilith is a general purpose tool that provides a highly scalable, easy distribution of user code across a heterogeneous computing platform. This capability is of value in the development of tools to be employed in the use and administration of very large (thousands of nodes) clusters. Because users are only minimally responsible for writing their user tool code, with Lilith handling the details of propagation and distribution and ensuring scalablility, Lilith is easy to use and Lilith-based tools can be rapidly developed.

Lilith can be used for the creation of tools employed for both the control of user processes on the distributed system as well as for general administrative tasks on the system itself (e.g., creation of user accounts, monitoring of system status). Although there exist tools to accomplish some of these tasks, they are not scalable or rely on relatively weak security. Furthermore, the rapid development cycle promotes the building of throw-away, limited-use tools that may not be cost effective to write using more traditional Unix tools (e.g., shell commands, rsh). Finally, Lilith-based tools can be written as prototypes in the ultimate development of more refined, hard-coded tools as well.

Lilith is written in Java, providing platform independence and allowing easy creation of graphical user interfaces with browser front ends. Java is object-oriented, further facilitating rapid Lilith-based tool development through object reuse.

In what follows we describe Lilith's uses, and compare it with other tools in Section 2. In Section 3 we describe those objects in the Lilith Distributed Object System with which the user interacts. In Section 4 we discuss an example of tool development, illustrating calls in the user code. In Section 5 we discuss the start-up of the Lilith environment and the distribution of the user code. In Section 6 we present results demonstrating Lilith's scalbility. We discuss results in Section 7.

[redacted]

2. Uses for Lilith and Comparisons with Other Tools

Lilith's[1] principle task is to span a tree of machines executing user-defined code. Beginning from a single object, Lilith recursively links host objects, LilithHosts, on adjacent machines until the entire tree is occupied. The LilithHosts propagate down the tree code objects, called Lilim[2], performing user-designated functions on every machine (Figure 1).

To provide modular security and support high performance asynchronous operations Lilith is targeted to be compatible with the Legion[3] object system. Legion has been chosen over more established object systems, such as CORBA[4] , primarily because of the need for designed-in security, supporting fine security gradations and degrees of control. (Security in Lilith is in the prototype phase and is discussed in more detail in Ref. 5.) Lilith builds upon the Legion framework by adopting a tree structure for scalability and using Java to gain first class objects.

This scalable distribution of code makes Lilith ideal for the basis for the development of tools for controlling user processes and general system administrative tasks to be used in the management of distributed systems. For example, such tools could allow a user to first query the system to find the status of the nodes in a system, use that information to determine which subset of nodes on which to run his code, and finally distribute his code to those nodes. Although there exist tools to accomplish some of these tasks, they are not scalable or rely on relatively weak security. The scalable nature of Lilith-based tools in these tasks vastly improves the time such queries and code distribution would take: a one second operation performed on a thousand nodes would take 16 minutes under a serial distribution, as opposed to 10 seconds via a binary tree distribution.

While Lilith shares some functionality with other tools, there are some important differences that distinguish Lilith. NASA's LAMS package[6] has been used to control and modify system files in a secure fashion on hundreds of hosts. The software works by having a daemon running on each host, and sending shell scripts to run on those hosts. The channel between the send and receiver is encrypted and the script can be signed. The files on the hosts are owned by root and kept in a directory with only root access. The scripts are created and built on a master station and sent to each of the slave machines. This mechanism, while currently used just for scripts it could be generalized to other programs. While the security aspects are similar, the scaling is O(N) rather than O(logN) with Lilith. Also, Lilith is object-oriented and written in a more portable language (Java vs. C) allowing for more rapid development and deployment.

Platform Computing's LSF[7] is primarily a batch system with the ability to run interactive jobs within the system. While LSF does have some security, it is weaker than encrypted and signed messages. It also tends to run as in master/slave mode and suffers from the same scaling problems as LAMS. The latest version does seem to have a notion of a hierarchy, but it isn't clear from the documentation how this will affect scaling or general execution behavior. Of course, LSF is a batch system rather than a distributed object tool, so the targets are quite different.

Finally there is the Ptools project[8]. This project has produced many tools of use for distributed systems. These include parallel versions of the traditional UNIX tools and corefile browsers among others. Again, while these may scale better than O(N), they are not general-purpose and lack a flexible security model.

3. User-interactive Objects in the Lilith Distributed Object System

The purpose of Lilith is to span user code to large numbers of machines scalably, not to invent a new distributed object system. Currently, we implement a simple distributed object system, based on the Legion object system, written in Java. In this way, we have access to the data at every step and can provide hooks to implement security schemes[9]. Furthermore, by writing the object system in Java, we get all the advantages of an object-oriented language: encapsulation and modularity; the ability to test components separately; extensibility; and platform independence.

The Lilith Distributed Object System was presented in detail in Ref. 5. Here, then, we present only those objects with which the user has to be directly concerned. These are the object encompassing the user's code, the host objects which maintain the tree and (indirectly) provide the basic functionality of Lilith, the object by which the user code communicates with those host objects, and finally the messaging object used in communications.

Lilim are the Lilith objects that carry user code within them. Lilim, as well as the core Lilith objects, export well known interfaces which allow the user code to interact with the Lilith environment. The Lilim run as threads or autonomous processes under the LilithHosts. Lilith-based tools are created by construction of suitable Lilim. The Lilim need only be written with the goal of the tool itself in mind; details of code distribution and communication between nodes in the tree are handled by objects in the Lilith environment.

The LilithHost object is responsible for protecting the computing resource on which it is running from other Lilith objects and for instantiating Lilith objects on that host. For purposes here, we define a host to be a system running under a single OS. If Lilith is being used simply as a platform for remote objects there may be more than one LilithHost per host. LilithHosts maintain the tree and, via lower level objects, communicate with one another. Host to host communications are done via an object-oriented Remote Procedure Call (OORPC). Details of building the tree and instantiating the Lilim are given in Section 5. Details of the OORPC are not germane to this discussion and are discussed in Ref. 5.

Lilim interact with the Host Object and each other through the Lilim-Implementation-CommunicationsObject, aka LICO. Thus each node of the tree consists of a LilithHost with a Lilim running on it, and a LICO used for communications between the two. The LICO provide a well defined set of methods by which Lilim can send data up and down to other Lilim in the tree. LICO pass arguments to the Lilim and gather up their return messages for passage back up the tree. These methods are discussed in detail in the next section. By compartmentalizing the interactions of Lilim with Lilith in this way, not only is the entry-level knowledge needed to use Lilith small but also access to Lilith Objects by a potentially malicious user is contained. For instance, rogue Lilim cannot by *direct* call get illegal control of the lower level objects in the system which provide socket access. This restriction is enforced though Java Package assignments and checks on the sequence of classes in the execution stack whenever the SecurityManager is invoked.

Communications are handled through the sending of Message Objects, MOs. The MO is a general-purpose data rack that can hold a list of data objects. It is capable of marshaling this data into a byte stream and unmarshaling it from a byte stream and recreating the data in a new MO. Each data item consists of a length, type, and the actual data. Data is placed into and removed from the MO through a well-defined set of calls pertaining to the primitive data types such as push/pull/peekInt()[10], push/pull/peekString(), as well as push/pull/peekMO(). Push places an object into the MO, pull removes it from the MO, and peek returns the value without removing it from the MO. There is also hash table interface which allows the users to assign a label to each item. This interface can be used to provide random access to internal MO data structures. The methods are hashedPut(), hashedGet(), and hashedRemove() and they support the Java wrapped types corresponding the primitive types supported by MO. The total set of user-related calls for assembling/disassembling the MO is defined in Table1.

MOs also carry with them an id field (MOUUID) used for identification of the MO and for matching sends and returns. The MOUUID is in the form of a user-defined String. MOUUIDs are set/returned using set/getMOUUID (see Table 1). Use of the MOUUIDs is described in more detail in the next section.

In the next two sections we discuss starting the Lilith environment, instantiating the Lilim, and the methods LICO provides by which the Lilim interacts with Lilith. We begin in Section 4 with LICO since users will more often be sending Lilim down existing trees in independently started Lilith environments. Section 5 then covers Lilith start-up and Lilim distribution.

4. LICO API and Example Lilim

The LICO provides a number of methods for distributing data down the tree and for processing results up the tree. In this section, we discuss these methods, and present an example of their use.

The LICO API is given in Table 2. Methods used in distributing data down the tree are getArg(), get(), and scatterToChildren().These methods are illustrated in Figure 2. The call getArg() is used by a Lilim to get data, in the form of an MO, initially sent down with the Lilim bytecode. (Details of Lilim distribution are discussed in the next section.) Data sent in this manner can be, for example, general command line options to the user code. Sending this information down with the Lilim reduces the number of messages required. In order to support multiple concurrent Lilim, the data for each Lilim are distinguished by a unique String tag, called a MOUUID. Thus getArg() takes as an argument a MOUUID and returns from the LICO the corresponding MO. Tagging of the MO occurs either via the call MO.setMOUUID() where the tag is a user-defined String or by specifying the MOUUID in the MO's constructor (See Table 1).

Tagging of MOs via the MOUUID is also used for matching up messages sent from parent Lilim to child Lilim and vice versa. (Although the terms "parent" and "child" are more accurately used in terms of the LilithHosts which maintain the tree, the extension of this terminology to the Lilim residing on those hosts is straightforward and unambiguous.)

The methods get() and scatterToChildren() are used in tandem to get messages from the parent Lilim and send messages down to the child Lilim. A Lilim sends a message to its children via LICO.scatterToChildren() using a tagged MO as its argument. This method puts each MO into the LICO corresponding to each child. A Lilim gets an MO from its LICO via LICO.get() using the appropriate MOUUID tag as the argument. (Thus get() is functionally similar to getArg() except that getArg() refers only to those messages send down with the Lilim, while get() refers to messages that sent during the running of the Lilim.)Messages can then be sent *recursively* down the tree by each Lilim first calling LICO.get(myTAG) and then calling scatterToChildren(myMO) where myMO has been tagged with the same tag, myTAG. (This will be illustrated in the example later in this section.) Communications both down and up the tree are thus only with the levels in the tree directly above or below the current level.

Processing results up the tree also are performed recursively. In this case the methods put() and gatherFromChildren() are used in tandem. These methods are illustrated in Figure 3. A Lilim calls gatherFromChildren() with a tagged MO as its argument to receive an MO array containing all MO results from its children. This method gathers tagged MOs from each of the children's LICOs. The call gatherFromChildren() blocks execution in the Lilim until returns from all the children have come into that Lilim's LICO. The same Lilim then makes its own results available to its own parent by calling LICO.put() with an argument of its own identically tagged MO. Note that a child Lilim calls LICO.put() in anticipation of the parent calling gatherFromChildren() to collect that MO. Thus the processes of sending messages down the tree and of gathering returns back up the tree are both initiated by the parent.

The recursive up and down calls and tagging are illustrated in an example using a distributed sort. This usage capitalizes on the fact that it is faster to sort a set of presorted sets than to sort an entire unsorted list from scratch. In this case the downward processing will consist of: each Lilim getting a list of numbers to sort, subdividing that list into pieces for itself and its children to sort, and then sending those pieces down to its children. After a Lilim has sorted its own piece, it then processes the results back up the tree by: gathering the children's sorted list, combining their results with its own via a merge sort, and then passing the combined sorted list up to its parent. The relevant pseudo-code is as follows:

```
public Class Lsorter implements Lilim{
  private LICO myLICO;    // field for LICO with which this Lilim
                             communicates
     ...
  public void run(){
     MO tmpMO = myLICO.get(TAG1); /* gets an MO from the LICO
                             containing the list of numbers to be sorted */

     ... /* Code here which:
              1) Unpacks MO to get array of numbers to sort via calls to
                 tmpMO.pullInt();
              2) Divides array into subarrays for self and children.
```

```
                    3)  Packs arrays for children into MO[] kids_piecesMO via
                        calls to kids_piecesMO[i].pushInt(int)
                    4)  Set MOUUIDs on each MO to TAG1 via
                        kids_piecesMO[i].setMOUUID(TAG1)
        */

    myLICO.scatterTOChildren(kids_piecesMO); /* Scatters MO's
                                        with arrays for children to sort */
    sort(myArray); /* sort own piece using own sort method */
    kids_piecesMO = myLICO.gatherFromChildren(TAG2); /*
                        gather MO's containing sorted arrays from children */

    ... /* Code goes here which unpacks sorted arrays from kids_piecesMO
        and places them intokidsArrays*/

    finalArray = mergeSort(myarray,kidsArrays); /* merge sort
                                                all sorted arrays */

    .../* Code here which:
                    1)  Packs final array into tmpMO
                    2)  Sets MOUUID tag on tmpMO to TAG2
      */
    myLICO.put(tmpMO); /* Put MO containing final sorted array into LICO
                                        for parent to gather */
  }
}
```

In the above example, the packing and unpacking of messages is not explictly shown - these are straightforward calls to push/pullInt() and setMOUUID(). The key thing to note is the usage of the tags in the operation of the recursive calls. TAG1 is used to obtain the correct MO from the parent via get(), and is therefore also used to make the MO sent from the parent in scatterToChildren(); thus TAG1 is used for the signaling in sending the messages down the tree. Similarly, TAG2 is used in put() and gatherFromChildren() on the sending returns back up the tree.

Many tools can be written using this basic structure. In the most general case, the code section handling the sort can be replaced with code to execute a shell script that performs some action on each node. The sequence of calls to scatter information down the tree and gather it back up, as well as the tagging, can be reused unchanged. Only the packing and unpacking of the messages will have to be tailored to reflect the specific types involved.

5. Lilith Client Operation

In this section we discuss the start-up of the LilithHosts, the building of the tree of Hosts, and the distribution of Lilim down the tree. The tree, once built, can support multiple sequential or concurrent Lilim; in many cases, then, users will not be responsible for starting Hosts or building trees, but will merely be taking advantage of preexisting trees. (In such cases the user need only be concerned with the information pertaining to the call sendLilim() discussed below).

In order to start the Lilith environment, the LilithHosts must be started upon all participating systems. This can be done by several mechanisms. The most simple of these is that all LilithHosts are started externally to Lilith and the root LilithHost takes a list of these hosts and their ports and protocols and connects them into a tree. The syntax for each host in the list is the LilithObjectAddress (LOA), a host:port:protocol tuple. (Lilith supports both TCP and UDP.) There is also a start-up mechanism in which initially only the root LilithHost is started and the other Hosts participating in the tree are started as the tree is built. In this case, the running LilithHost takes the list, starts its immediate children, and passes them the list of their descendants. This process is then repeated, building the entire tree in a scalable fashion.

Lilith provides utility classes that can parse a specified file on the client. The format of the file is a <label> and <LOA>, one tuple per line, and lines that indicate the children for the specified LilithHost. A sample file might look like:

```
cp1 bach.sandia.gov:2300:TCP
cp2 mozart.sandia.gov:2300:TCP
cp3 chopin.sandia.gov:2300:TCP
cc1 cooltime.ca.sandia.gov:3456:TCP
tree: cc1 children cp1 cp2
tree: cp2 children cp3
```

The client can use these classes to construct the list to send to the root host. Initially the client must create a stub object to access the root LilithHost; this is done by:

```
 LilithHost root =
     new LilithHostStub("HOST1:2345:UDP");
```

The tree is most simply built by creating the list of hosts (usually using the utility classes) and passing that to the root's buildtree method, (the detailed mechanism of building the tree is described in Ref. 5)

```
/* create String called TreeList containing
     list of addresses */
root.buildtree(TreeList);
```

The user then creates and sends the Lilim down the tree. (This is also the starting point for cases where the user is taking advantage of a preexisting tree.) The Lilim is a Java class file with a class that implements a Lilim interface in the package lilim. These Lilim objects are loaded using a Lilim specific ClassLoader object. The client reads this class into a byte array when it is executed, and sends it as an argument of LilithHost.sendLilim(). The client can also send initial data that the Lilim can read with getArg() by specifying a tagged MO as the second argument to the call:

```
byte[] mylilim = new byte[LILIM_LENGTH];
// read class file, named  "dostuff.class" into mylilim
```

```
MO initial_data = new MO("initdata4567");
initial_data.pushInt(30);
root.sendLilim("dostuff",mylilim,initial_data);
```

The user then starts the Lilim on the hosts by:

```
root.runLilim("dostuff");
```

The client can send down additional data to the Lilim and get results from the Lilim using LilithHost.scatterToChild(MO), and MO LilithHost.gatherFromChild(String). On the server side, these methods simply call LICO.put(MO) and LICO.get(String) respectively on the LICO corresponding to that host. That these methods are reminiscent of the communcations between parent and child Lilim discussed in the previous section is not suprising; in fact, LICO.scatterToChildren() and LICO.gatherFromChildren() call these Host methods as part of their functionality. From the child Lilim's perspecitive, calls initiated by the client or by the parent Lilim appear the same - the Lilim has only to interact with the LICO via LICO.get(String) and LICO.put(MO) to get the scattered information or to return information irrespective of the initiator of the caller. Tagging the MO's for LilithHost's scatter and gather calls thus proceeds identically to the tagging for the LICO's scatter and gather calls. For example:

```
MO additional_data = new MO("more data 042376");
additional_data.hashedPut("first int",
     new Integer(1021));
root.scatterToChild(additional_data);
```

This will be picked up by the child calling:

```
MO data2 = myLICO.get("more data 042376");
```

6. Scaling Behavior

To demonstrate the scaling behavior of Lilith we consider the scaling behavior of Lilith in two cases: constant work per processor (increasing total work) and constant total work. In both these cases, we first establish the tree. Then, the wall clock time is measured in the client, and the Lilim is sent to the server tree. After the server returns to the client, the wall clock time is again measured and the results are tabulated for varing number of servers. In each server, the Lilim is first sent to its children, the Lilim is then instantiated and executed locally, and finally the results from the children are collected. Timings were generated on a 32 processor SGI Origin 2000 and only one server was run per processor.

Figure 4 shows the scaling behavior for constant work per processor, i.e., increasing total work. The outer plot is 100 random prime numbers generated per processor; inset is 500. This calculation was chosen since each node could be assigned the same seed, guaranteeing the same amount of work per node. We observe the expected overall logarithmic scaling. The step-like behavior occurs due to increasing the tree depth through addition of nodes such that additional time is required for communications. The steps are still seen for the case of greater work/processor, although curve flattens out. As the ratio of work to communications time increases, the work overwhelms the communication time causing all nodes to essentially run in parallel. It is no suprise that the greatest benefit of the logarithmic scaling thus comes in cases where the communications/work ratio is kept small.

Figure 5 shows the scaling behavior for constant total work, i.e., decreasing work per processor. This plot shows the timings for calculation of a fixed total number of random numbers. In this case, the total number of primes generated per Host decreases as the number of participating Hosts increases. As expected, the overall time for the calculation decreases and logarithmic scaling is again observed.

7. Results and Conclusions

Results for constant work per processor and constant total work are shown in Figures 4 and 5. Timings obtained by use of Lilith to send the Lilim down the tree exhibit logarithmic behavior. This scalability enhances the ability of users and administrators to function when using a very large (thousands of nodes) cluster. Ideally, a one-second operation to be performed on 1000 processors will take 16 minutes to execute in the traditional serial fashion, contrasting with the approximately 10 seconds for the same operation under Lilith.

Lilith's purpose is to provide a highly scalable, easy distribution of user code across a heterogeneous computing platform. By handling the details of code distribution and communication, such a framework allows for the rapid development of tools for the use and management of distributed systems.

We have presented an example tool, illustrating Lilith-based tool development and use of the Lilith User API. We have shown that Lilith has excellent scaling behavior. Lilith fills a gap in the repertoire of tools for current, and planned computing environments, particularly environments consisting of high-performance commodity computers connected by low-latency, gigabit networks.

Work supported by the U. S. Department of Energy under contract DE-AC04-94AL85000.

Figures and Tables

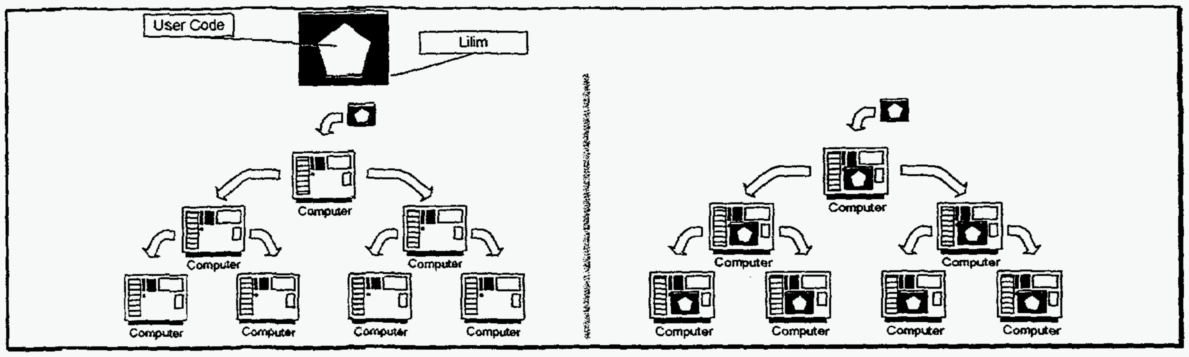

Figure 1: User code, contained in Lilim, propagate down the tree of LilithHosts.

MO METHOD	ACTION
void pushXXX[a](yyy)	adds item yyy of type XXX to the MO
XXX pullXXX[a]()	removes an item of type XXX from the MO
XXX peekXXX[a](m)	returns the value of the item of type XXX at position m (integer) from the MO *without* removing it from the MO
Object hashedPut[b](String, yyy)	adds object yyy to be specified in the hash table interface by the String value returning the previous object.
Object hashedGet[b](String)	returns a reference of the object specified by the String value when placed in using hashedPut(). This method does not remove the object from the MO.
Object hashedRemove[b](String)	returns the value of the object specified by the String value when placed in using the hash table interface, hashedPut(). This method removes the object from the MO.
boolean hashedIsEmpty()	returns true if there are no hashed items in this MO.
Enumeration hashedKeys()	returns an object that allows the user to enumerate over all the keys in the hash table.
void setMOUUID(String)	sets the MOUUID to the String value
String getMOUUID()	returns the MOUUID value of this MO

Table 1: MO User API. [a]Valid types for XXX are all the primitive types, as well as String, ByteArray (for byte[]), and MO. [b]Valid types for the hash table interface are all the wrapped types as well as String, ByteArray, and MO. These are returned as class Object references and must be cast to the desired class.

LICO METHOD	ACTION
MO get(String)	returns an MO from LICO with MOUUID equal to the String value
void put(MO)	puts MO into LICO
void scatterToChildren(MO[])	scatters MO[] to children
MO[] gatherFromChildren(String)	returns MO's from children corresponding to MOUUID equal to the String value
MO getArg(String)	returns initial MO sent down with Lilim with MOUUID equal to the String value

Table 2: LICO User API. Methods in the LICO called by the Lilim.

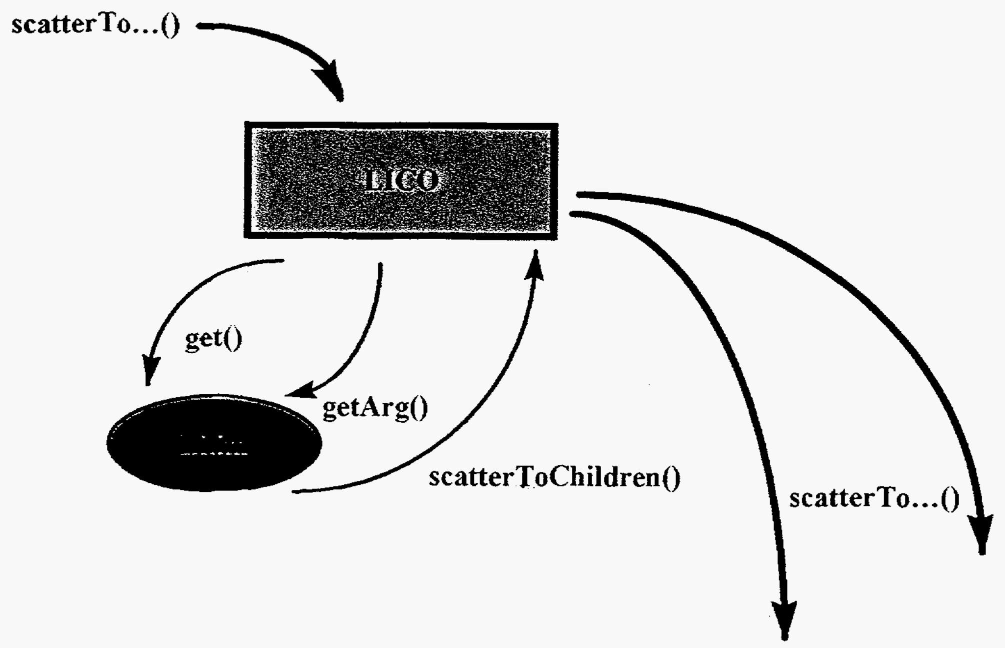

Figure 2: Methods involved in sending data down the tree.

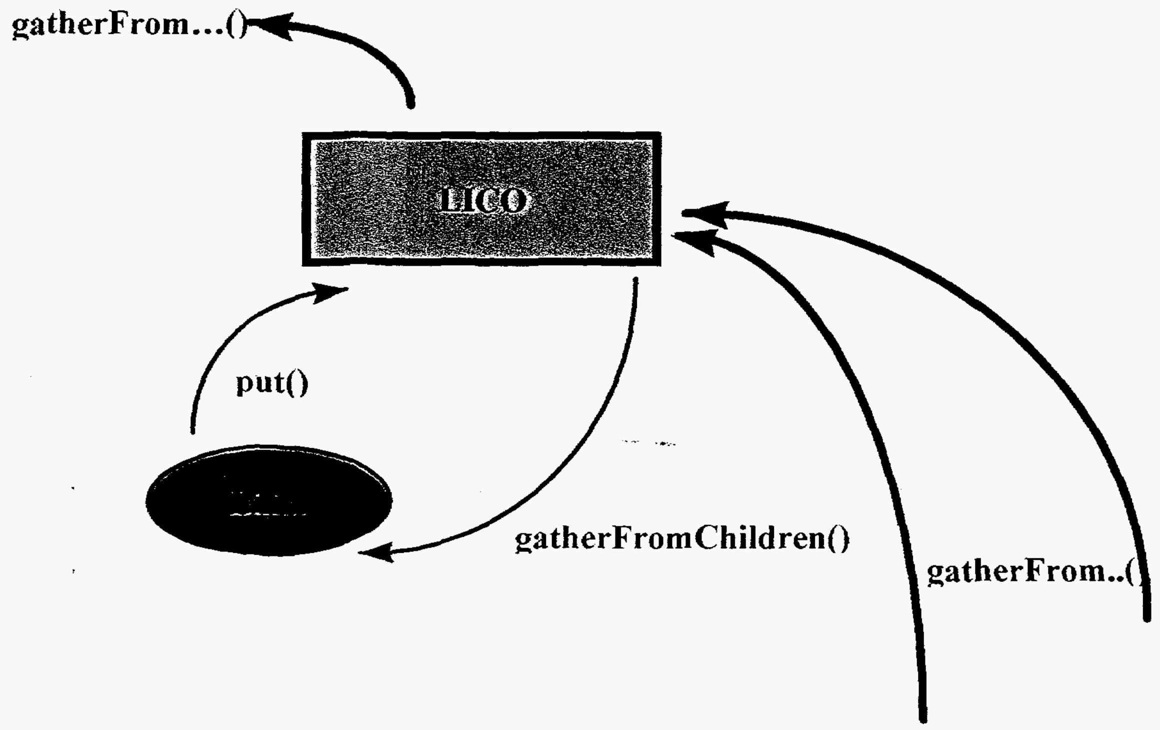

Figure 3: Methods involved in returning data back up the tree.

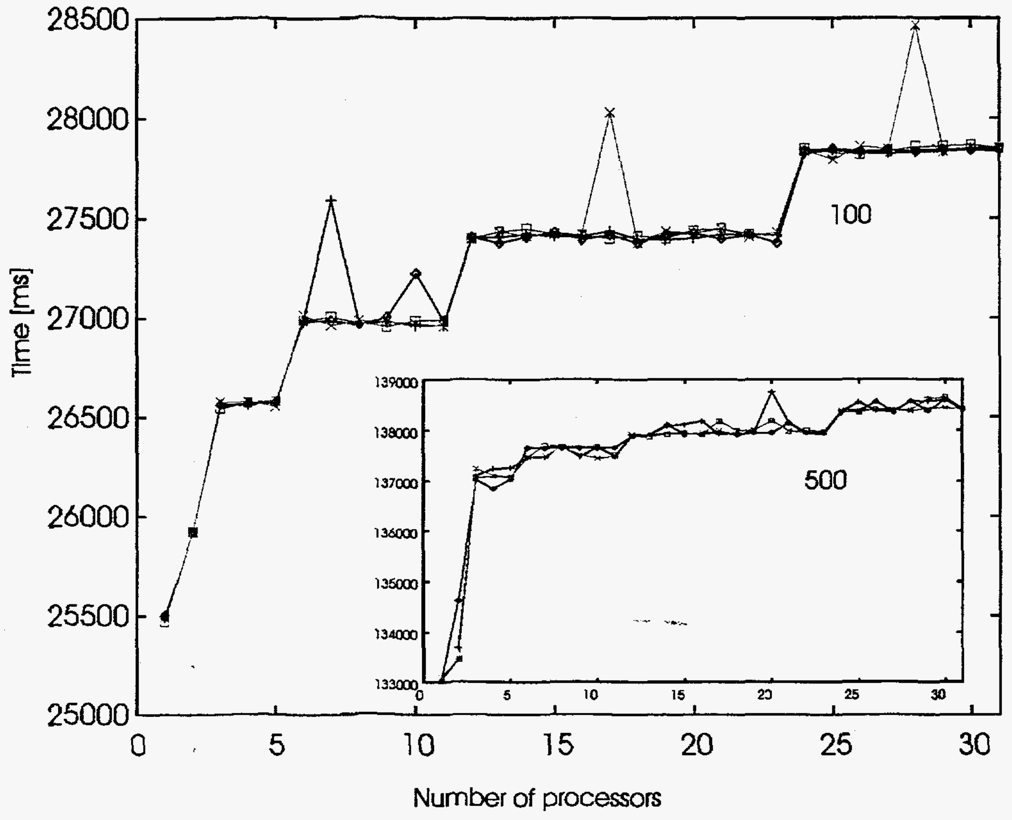

Figure 4: Scaling behavior for constant work per processor, i.e. increasing total work. Larger plot is 100 prime numbers generated per processor; inset is 500. Scaling is overall logarithmic. Steps occur due to increasing tree depth such that additional time is required for communications. Steps are still seen for increasing work/communications time, though curve flattens out.

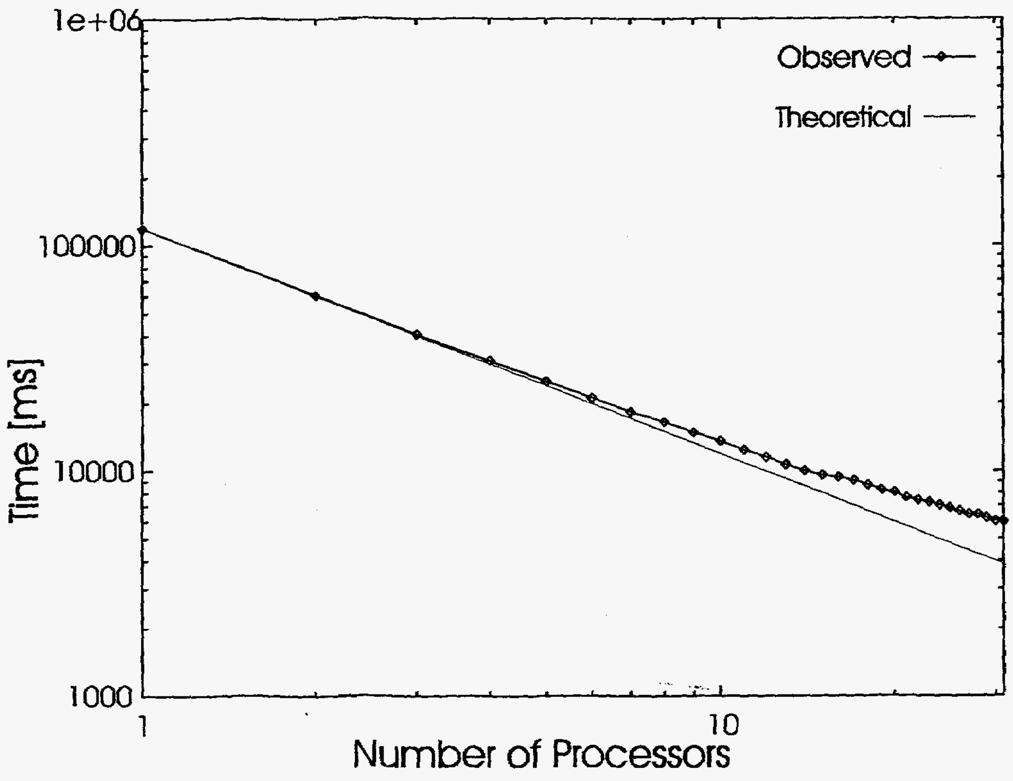

Figure 5: Scaling behavior for constant total work (a fixed total number of random numbers calculated) , i.e., decreasing work per processor. Overall time for the calculation decreases with logarithmic scaling.

[1] Lilith is the mythological "Mother of D(a)emons".

[2] Lilim (the children of Lilith) will be used for both the plural and singular name of the first class objects that are sent down the tree.

[3] A.S. Grimshaw and W.A. Wulf, "Legion -- A View From 50,000 Feet", Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California, August 1996, and references therein.

[4] Common Object Request Broker Architecture, by the Object Management Group (OMG), see http://www.omg.org.

[5] D. A. Evensky, A. C. Gentile, L. J. Camp, and R. A. Armstrong, "Lilith: Scalable Execution of User Code for Distributed Computing", Proceeding of the 6th IEEE International Symposium on High Performance Distributed Computing, Portland, OR, August 1997.

[6] Login Account Management System (LAMS), Final Report on Subcontract CSC/ATD-WR-MV-NAS2-9005, Matt Bishop, National Aeronautics and Space Administration, Ames Research Center, Moffett Field, CA 94035-1000.

[7] Load Sharing Facility (LSF), see http://www.platform.com.

[8] The Parallel Tools Consortium (Ptools), see http://www.ptools.org.

[9] This is not possible in the current implementations of Legion or Java's Remote Method Invocation (RMI), though they may well prove suitable in the future.

[10] Method calls will be given as "functionname()" with the arguments only specified when germane to the discussion.