SAND--98-8513C

A Visualization Tool for Parallel and Distributed Computing using the Lilith Framework

CONF-980805--

Ann C. Gentile, David A. Evensky, and Pete Wyckoff
P.O. Box 969, Sandia National Laboratories, Livermore, CA USA 94551
{gentile,evensky,wyckoff}@ca.sandia.gov

Abstract

We present a visualization tool for the monitoring and debugging of codes run in a parallel and distributed computing environment, called *Lilith Lights*. This tool can be used both for debugging parallel codes as well as for resource management of clusters. It was developed under Lilith, a framework for creating scalable software tools for distributed computing (`http://dancer.ca.sandia.gov/Lilith`). The use of Lilith provides scalable, non-invasive debugging, as opposed to other commonly used software debugging and visualization tools. Furthermore, by implementing the visualization tool in software rather than in hardware (as available on some MPPs), Lilith Lights is easily transferable to other machines, and well adapted for use on distributed clusters of machines. The information provided in a clustered environment can further be used for resource management of the cluster. In this paper, we introduce Lilith Lights, discussing its use on the Computational Plant cluster at Sandia National Laboratories, show its design and development under the Lilith framework, and present metrics for resource use and performance.

1 INTRODUCTION

We present Lilith Lights, a visualization tool for parallel and distributed computing. Lilith Lights provides graphical information about the CPU usage of nodes on a machine or within a cluster, as well as information about the communication among nodes. This information is invaluable in the debugging of parallel and distributed codes. Even to the application designer, the visualization of the operation of the code can yield some surprises, especially if there are complications, such as the use of multiple parallel libraries within the same communication context. Tracing problems in algorithms utilizing such libraries can be difficult, if not impossible, without access to information on which nodes are working and communicating. Real-time visualization of the activity frequently offers the insight necessary to point the debugging or optimization efforts in the right direction.

Several popular options exist for visualizing and debugging parallel codes. Common software implementations are XPVM[1] and XMPI[2]. These work by inserting additional information gathering subroutine calls into the underlying message-passing software employed by the cluster, and hence alter the conditions under

which the code is being run. Additionally the state of the information displayed by such tools is always lagged behind the application, making the debugging process less direct, especially when attempting to monitor other external devices which are not "controlled" by the message-passing layer, such as remote network connections. In the same vein, this level of tool is unable to monitor multiple independent applications on the same machine, such as is frequently the case with clusters of multi-user SMPs.

The other point for implementation of this sort of status gathering is in hardware. These implementations are available on some MPPs, such as the Intel Paragon, and can be more useful to the programmer since they are, by design, non-invasive and real-time. Hardware implementations are, of course, inherently platform-dependent and are not practical in the use of distributed clusters where the system is not self-contained and the configuration is not static. Furthermore, the information is not readily accessible: viewing the MPP lights typically involves a stroll down to the machine room.

Lilith Lights combines the best of the hardware and software implementations, providing a low impact, nearly real-time, platform independent software solution, adaptable to both MPPs and distributed clusters. Also, the presentation of the traffic can be application specific, giving the programmer further insight into the communication of his parallel code in as direct a manner as possible. The non-intrusiveness, timeliness of results, and platform independence of Lilith Lights is provided by the use of the Lilith[3] framework for development of scalable tools for distributed computing.

Lilith is a freestanding framework for tool development. The Lilith-based visualization tool runs independently from the application being debugged. In this way, utilization of Lilith Lights does not directly affect the system under study. With Lilith, tools are constructed by the creation of code which fits into the Lilith framework, providing the tool functionality on a single node. Lilith itself handles the details of the propagation of the code containing the tool code to the nodes, and communication among nodes. Lilith employs communication patterns in a binary tree configuration, providing scalability. This vastly improves the time that conventional serial queries and code distribution require: a one second operation performed on a thousand nodes would take 16 minutes under a serial distribution, as opposed to 10 seconds using a binary tree distribution. In this way, real-time debugging information can be provided to the user while the parallel code is running. Finally, Lilith is written in Java, providing platform independence.

Use of Lilith Lights is not limited to debugging purposes alone. By providing information about node usage and communications on a distributed system, the tool can be used for establishing resource management decisions as well. This includes both at the system administration level as well as at the user level, because full information of which nodes and communication links are busy is provided.

In the next section, we present Lilith Lights, discussing its use in Sandia's Computational Plant. Then we discuss design of the code internals and their relation to the Lilith framework. A brief overview of Lilith is given in this section. Finally we present metrics describing the tool's performance and its demand on the system resources.

2 LILITH LIGHTS AND ITS USE IN HIGH PERFORMANCE CLUSTER COMPUTING

In this section, we discuss Lilith Lights and its use in distributed cluster computing. In Section 2.1 we discuss the tool interface; code internals are covered in Section 3. In Section 2.2 we describe the use of the tool on Sandia's Computational Plant distributed cluster.

2.1 User Interface

The graphical user interface of the tool is shown in Figure 1. The system load of each computational node is displayed as a moving vertical bar, and traffic on the network links connecting the nodes is represented by coloring the unidirectional arrows, with brighter colors indicating a higher level of consumed bandwidth.

There is also a text area for displaying messages from the top node in the data-gathering tree, such as the state of the other nodes and the current display latency. Pop-up windows allow changes to the coloring of the network traffic arrows to highlight a particular range of interest, such as links which are operating in a near-saturation condition, or links which should be nearly devoid of traffic. There are also pull-down menus to control the virtual network topology, freeze the display, print the current image, and other functions.

The interconnections displayed between the computational nodes are chosen at runtime by the user to represent the virtual topology of interest. In Figure 1, for example, we show a rectangular two-dimensional mesh, which is appropriate for many physical simulations. The underlying physical network need not be a 2D mesh, and traffic on it is mapped into the displayed virtual mesh. The order of the nodes and their connections are read from a configuration file at run time which corresponds to a user's chosen topology. Routes for traffic between the nodes are also specified if the default routing algorithm is not appropriate. This works well for CPlant, described in the next section, which uses a static source-based routing protocol. The display may also be configured to show the true physical representation of the network, which is useful for system designers and some software developers, in which case no routing algorithm is applied since the arrows represent physical links.

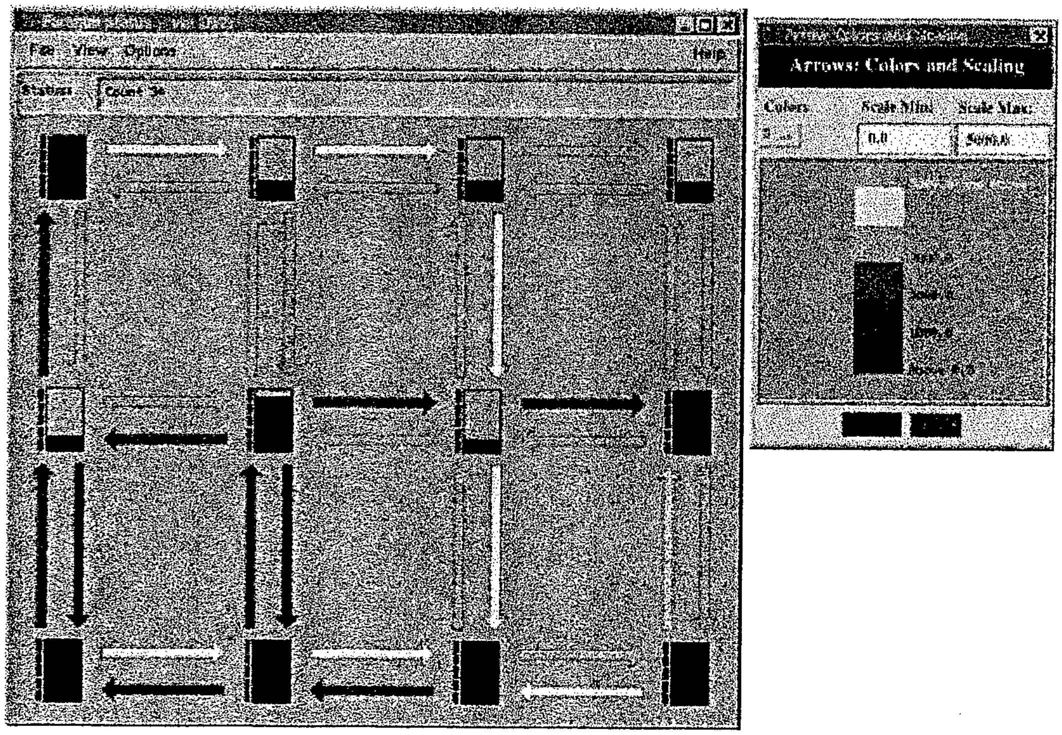

Figure 1. Screen shot of the Lilith Lights, showing the panels for controlling the scale of the network bandwidth and parameters affecting data gathering.

2.2 Lilith Lights in Cluster Computing

The future of high-performance computing lies in massively parallel simulations, which traditionally have been performed on monolithic massively parallel processors (MPPs). These are unfortunately becoming too expensive on a per-flop ratio when compared to their smaller commodity brethren. These cheap, widely available personal computers (PCs) can be purchased whole or piece-wise from a variety of vendors and, to a large extent, conform to the same basic set of hardware standards such as having a PCI bus. Networking components required to assemble many PCs into a single system are likewise cheap and plentiful.

Sandia has embarked on a program to build its next-generation supercomputer out of commodity components. The idea behind the Computational Plant (CPlant) is that installations of new hardware will be attached to the base "tree" every so often, and older "growths" will be pruned. The development cycle of computer-related technologies has been shown to be about 18 months, making it important not to rely solely on one manufacturer or system for long-range computing. Instead, by sticking with some well-defined set of standards for hardware, recent evolutions in hardware design can be quickly added to the current computing platform, which still remains viable.

The necessary pieces to make this plan work are the software infrastructure. It is unacceptable to demand that application programmers must learn a new operating system, or even a new set of compiler flags, to use the newest growth of the machine. To meet this goal, a freely available, widely ported operating system and set of tools is necessary, such as Linux and MPI.

A major stumbling block for cluster builders has been the failure to ensure that the system would scale. This is distinct from the usual requirement that applications must scale well to be efficient, and speaks to the reliability and usability of the system itself. CPlant is designed as an arbitrarily connected aggregation of individual quantized pieces called scalable units, each of which is entirely self-contained and self-managing.

Lilith Lights is one example of a tool which capitalizes on the scalable structure of the CPlant hardware system. It provides a valuable monitoring component which had been available as hardware in some MPPs, and is designed along the lines of the CPlant hardware to take advantage of the inherent scalable nature of the system.

3 DESIGN

In this section we discuss the internals of the visualization tool. There are three major elements to the code. The first is the Lilith package itself, which is responsible for the details of sending and gathering information in a scalable fashion on a group of machines. The second is the kernel code which obtains the CPU and network activity information on each node. The last is the code that is the glue between these first two pieces: it obtains the information from the kernel code, filters and combines it for its purposes, packages the information in a way accessible to the Lilith code, and delivers it to the Lilith framework.

3.1 Overview of Lilith in Tool Implementation

Lilith's principle task is to span a tree of machines executing user-defined code. Lilith is a framework encompassing a number of code objects that handle all the details of maintaining the tree and enabling communication among nodes in the tree. Beginning from a single object, Lilith recursively links host objects on adjacent machines until the entire tree is occupied. The host objects propagate user code, called "Lilim" (both a singular and plural noun), down the tree. The Lilim performs user-designated functions on every machine.

Finally, the results of the Lilim are propagated back up the tree. The Lilim can further process the results from lower nodes as those results are passed back up the tree. Typically, the Lilim undergoes a three-phase process: distribution, execution, and result collection.

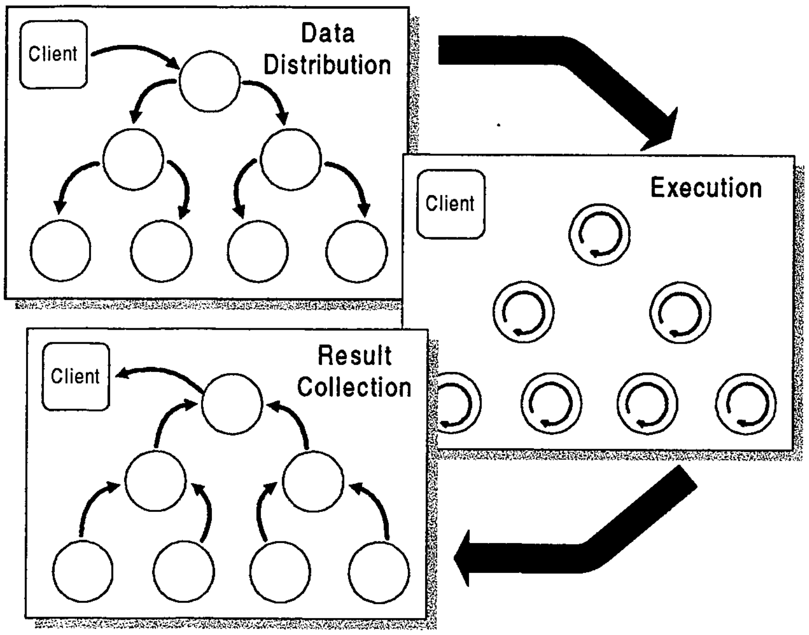

Figure 2. Typical three-phase process for Lilim under Lilith: distribution, execution, and result collection. Lilith handles the details of distribution, collection, and communication. The tree structure is used for scalability.

Lilith-based tools are created by writing suitable Lilim. The Lilim need only be concerned with the functionality of the tool itself; details of code distribution, result collection and communication between nodes in the tree are handled by the Lilith framework. By alleviating the tool writer of these details, Lilith enables rapid tool development. The tree structure in Lilith is chosen to provide logarithmic scaling. This scalable distribution of code under Lilith makes it ideal as the basis for the development of tools to be used in the management of large distributed systems.

For the tool presented here, the Lilim consists of code which gathers information on the CPU usage and communications of each node. Lilith handles the details of propagating that code down, and collecting the results from each node. Note that the communications structure in Lilith is independent of that going on in the computations on the nodes. In fact, on CPlant we route the Lilith information across a physical network distinct from the network used for application message-passing traffic. Lilith's structure is chosen to provide scalable communications to minimize latency; the communications on the nodes are a result only of the computations on the nodes. The Lilith process on each computational node consumes only a fraction of available CPU resources, as discussed in Section 4.

3.2 Computational Node Data Extraction

The component which sits at the lowest level of this data-gathering framework consists of a set of modules, or interfaces, which sit in the kernel and record the transfer of every packet across a chosen device. In particular, we have written a small routine which is called from the physical device driver of interest for each packet. In the case of the receipt of a packet, the kernel is operating in an interrupt handler and the recording subroutine must be very fast so as not to alter the performance of the system as a whole.

To be more concrete, the implementation we used involved modifications to our operating system of choice, Linux. First, in the existing device drivers themselves, code is added in three places:

- Incoming packet. The driver receives an interrupt from the hardware signaling the receipt of a packet. As soon as the source of this packet and its length are known, the driver calls the Lilith Lights module with this information, then continues processing as normal.
- Outgoing packet. Depending on the hardware, an interrupt may be generated to signal the completion of a queued packet transmission. At this point the Lilith Lights module is invoked. For devices which do not interrupt, the packet is recorded as transmitted when it is queued in the device.
- Initialization. A hook to allow the Lilith Lights module to inform the driver of its presence or absence takes the form of two small routines which set a local variable in the driver.

Since every network device and driver must support the functionality of transmitting and receiving packets, the addition of the above-described lines of code is always very natural. Even for devices which use "operating system bypass," the kernel still must be involved in setting up the shared communication buffers, and handling completion notifications from the device. We have modified the sources to the drivers for three devices so far: 3Com 3c509 10 Mb/s ethernet, DEC tulip-based 100 Mb/s ethernet, and Myricom 1.28 Gb/s myrinet.

The next-higher layer still resides in the kernel, and serves as the interface between the device driver and the user code which wishes to gather information about its operations. This module provides the following services:

- Device callback. The routine which is called by the device driver quickly adds the entry to a hashed list of remote hosts. Only the length of the packet is recorded, not a timestamp or information about the packet contents. The list of hosts is dynamically generated by adding new hosts as traffic to them is generated, and quickly becomes static after the machine has been operating for a few minutes.
- Data integration. The display client which graphically shows the network traffic and CPU load requires a temporal resolution no greater than the refresh rate of the human eye, but packets are handled much more

frequently. Thus the interface module performs binning of the data at a user-specified rate, and returns integrated values of communication load.

- User-code interface. A character device entry is created by the module which allows any code to perform reads and input/output control (IOCTL) operations which affect the module. After the reporting interval is established, the module will only return information from a read request if there is an interval's worth of data to report.

The above functionality operates in the kernel, and must be written in C. The adaptation of the code to any other kernel should be straightforward, provided source code for the device driver is available, and that a mechanism for module insertion is provided. Full source of the kernel is not required.

Incidentally, it is also possible to record traffic at the level of the message passing framework. Our goal initially was to provide as accurate a representation as possible of the utilization of the physical communication medium, and thus we strove to gather information at the lowest available level. Future additions will involve the insertion of code to gather similar information from message-passing library calls. This layer, although potentially far removed from the physical hardware, has the advantage of providing application-level information about the traffic as well. Using the Lilith framework to gather information at this level will still allow for real-time application monitoring.

3.3 Amassing and Delivering Node information within the Lilith Framework

There are two parts to the Java tool code: the client that presents the data to the user in a graphical fashion, and the Lilim that is sent down the Lilith tree and processes the data from the kernel device and its descendants.

At startup, the client creates ordered lists of the compute nodes and the pathways connecting them. It then constructs the routing tables for sending messages between nodes, and sends this table and the list of nodes down the tree. The routing table consists of a matrix where each element is an array of lights representing the hops a message must take to reach its destination. A Lilim processes these lists, keeping only the information that it needs (*i.e.*, routes from itself to all nodes with which it can communicate.) We also create hash tables that map IP addresses as read from the device driver to the node numbers sent by the client. Each Lilim then waits for a message from its parent requesting it to return traffic and load information. Upon receiving this message, the Lilim first passes it on to its children, then reads information from the kernel. System load comes from `/proc/stat`, and traffic delivered from the node is provided by the kernel interface through `/dev/lights0`.

The Lilim then stores the number of bytes transmitted from this node to each of the receiving nodes by setting the value of each intervening light as instructed by the routing table. After the Lilim updates the state of the display lights with its own information, the state values from child nodes are merged in as well. The final step in each iteration is to send the accumulated state to the next-higher level in the tree. Overlap of communication and computation is achieved by first requesting information from the child nodes so that the integration of local data into the lights proceeds concurrently with the same calculation performed by the children.

4 PERFORMANCE METRICS

There are several measures of performance that affect the usability of Lilith Lights. We will examine the CPU utilization of the tool, the interaction of message passing network traffic and the tool traffic, and finally the latency of the display.

We have measured the CPU utilization for a variety sampling rates in the device driver. by averaging over each measurement realization and over all the nodes. For long sampling times (3-5 sec.), we see a CPU utilization of 4-7%, with longer sampling times yielding lower CPU utilization as there is less work for the tool to perform. As one would expect, the CPU utilization increases as the sampling time decreases and can reach 25% for very short times. The measurements we have made are on 12 unloaded systems (only standard daemons, no application processes) and the averages were made over roughly 100 samples per node. The CPU usage difference between the root of the Lilith tree and the leaves is comparable to the standard deviation of the measurements.

The effect on network utilization depends on the sample rate and the number of hosts. In the worst case, each message consists of: the number of nodes, the system load for each node, four traffic measurements, and, possibly a load measurement for the Lilith process itself. For our runs, as described above, this is roughly 300 bytes sent by each host every 1-5 seconds. This is much less than the bandwidth of fast ethernet, 100 Mb/s, and greatly less than that of myrinet bandwidth. For our system, the message passing traffic is on the myrinet network while the Lilith traffic is on the ethernet, so we see no contention for network bandwidth.

While it is important to present the state information to the user rapidly enough that a true representation is displayed, the display should not update so frequently that it appears meaningless to the user. We have explored this parameter space by measuring the real (wall-clock) round trip time (RTT) elapsed between the invocation of a remote method and the reception of a return from that method to be about 40 ms. We typically instruct the device driver to deliver traffic information every 500 ms, which matches well with the observed RTT on a 12-node system, about 500-600 ms. This half-second sampling period is visually acceptable as well.

In our current version, the sampling period is controlled by the device driver, which returns data to the Lilim only at regular intervals. Because there is some delay time associated with the communication of traffic up and down the tree, and because the compute nodes do not share a global clock, we account for this discrepancy by adjusting the start of the sampling period on each machine. This latency is discovered at the start of a run by including timestamps in some data packets and calculating the skew along the tree.

5 CONCLUSIONS

Lilith Lights is a new visualization tool for parallel and distributed computing. It provides graphical information about the CPU usage of nodes on a machine or within a cluster, as well as information about the communication among nodes. Lilith Lights can be used for the debugging of parallel and distributed codes and for resource management of clusters.

Unlike popular software tools or hardware implementations for visualizing and debugging parallel codes, Lilith Lights is a low impact, nearly real-time, platform independent software tool, adaptable to both MPPs and distributed clusters. The non-intrusiveness, timeliness of results, and platform independence of Lilith Lights is provided by the use of the Lilith framework for development of scalable tools for distributed computing.

We have discussed design of the code internals and their relation to the Lilith framework. We have presented metrics characterizing the CPU utilization of the tool, the interaction of message passing network traffic and the tool traffic, and the latency of the display. We find that, for reasonable selections of the monitoring cycle time, the impact which the tool has on the system is negligible, both in terms of CPU consumption and network traffic. If the message-passing traffic uses a network distinct from that used by Lilith Lights, as in CPlant, this network impact completely disappears.

---

[1] A graphical console and monitor for PVM, http://www.epm.ornl.gov/pvm.

[2] A run/debug interface to MPI, http://www.osc.edu/Lam/lam/xmpi.html.

[3] a) http://dancer.ca.sandia.gov/Lilith; b) Lilith: Scalable Execution of User Code for Distributed Computing, D. A. Evensky, A. C. Gentile, R. Armstrong, Proceedings of the 6th International Symposium on High Performance Distributed Computing, IEEE, 1997; c) Lilith: A Software Framework for the Rapid Development of Scalable Tools for Distributed Computing, submitted to the 7th International Symposium on High Performance Distributed Computing, 1998.

Report Number (14) SAND--98-8513C

CONF-980805--

Publ. Date (11) 19980[illegible]3

Sponsor Code (18) DOE/ER , XF

UC Category (19) UC-405 , DOE/ER