MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools [1]

Philip C. Roth, Dorian C. Arnold, and Barton P. Miller

Computer Sciences Department
University of Wisconsin, Madison
1210 W. Dayton St.
Madison, WI 53706-1685 USA
{pcroth,darnold,bart}@cs.wisc.edu

Abstract

We present MRNet, a software-based multicast/reduction network for building scalable performance and system administration tools. MRNet supports multiple simultaneous, asynchronous collective communication operations. MRNet is flexible, allowing tool builders to tailor its process network topology to suit their tool's requirements and the underlying system's capabilities. MRNet is extensible, allowing tool builders to incorporate custom data reductions to augment its collection of built-in reductions. We evaluated MRNet in a simple test tool and also integrated into an existing, real-world performance tool with up to 512 tool back-ends. In the test tool, MRNet's performance was comparable to that of previous tool infrastructure. In the real-world tool, we used MRNet not only for multicast and simple data reductions but also with custom histogram and clock skew detection reductions. Although the tool's start-up protocol was already highly tuned, our tests of MRNet with 512 tool back-ends show significant improvements in average start-up latency.

Keywords: Scalability, tools, multicast, reduction, aggregation.

1 Introduction

The desire to solve large-scale problems in areas like climate modelling, computational biology, and particle simulation has driven the development of increasingly large parallel computing resources. There has been a steady deployment of traditional high-end parallel systems with many processors, such as the various ASCI systems [1] in the USA, Japan's Earth Simulator [9], and HPCx [26] in the UK. Coupled with the low price/performance ratio of commodity hardware, this desire has also led to the proliferation of clusters with hundreds and even thousands of nodes (e.g., [7,13,23]). Unfortunately, performance, debugging, and system administration tools that work well in small-scale environments often fail to scale well as systems and applications get larger. To address this problem we have developed MRNet, an infrastructure providing scalable multicast and data aggregation support especially designed for scalable tools.

A parallel tool's functionality can be divided into two categories: (1) data collection, analysis, and presentation; and (2) control of application processes. These activities are implemented by one or more components within the tool system. The components of a typical tool system are shown in Figure 1a; tools like TotalView [10] and Paradyn [23] follow this organization. Data collection and process control occurs in the tool's back-end components (often called tool *daemons*) running on the nodes of a parallel or distributed system. The user interacts with the tool via the user interface component. Data analysis and high-level control may be implemented in a separate component or be co-located with the tool back-ends. Often, analysis and user interface are implemented in the same component, commonly called the tool's *front-end*. All tool functionality comes at a cost, though the cost may take different forms (e.g., computation, communication, and storage) depending on the activity. If the activity's cost is larger than the underlying system can support, that activity limits the tool's overall scalability.

MRNet is a parallel tool infrastructure that reduces the cost of many important tool activities. MRNet-based tools incorporate a tree of processes between the tool's front-end and back-ends as shown in Figure 1b. MRNet uses these *internal processes* to distribute tool activities, reducing analysis time and keeping tool front-end loads manageable. MRNet-based tools send data between front-end and back-ends on logical flows of data called *streams*. MRNet internal processes use *filters* to synchronize and aggregate data sent to the tool's front-end. Using filters to manipulate data

1. This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, Lawrence Livermore National Lab grant B504964, and NSF grants CDA-9623632 and EIA-9870684. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

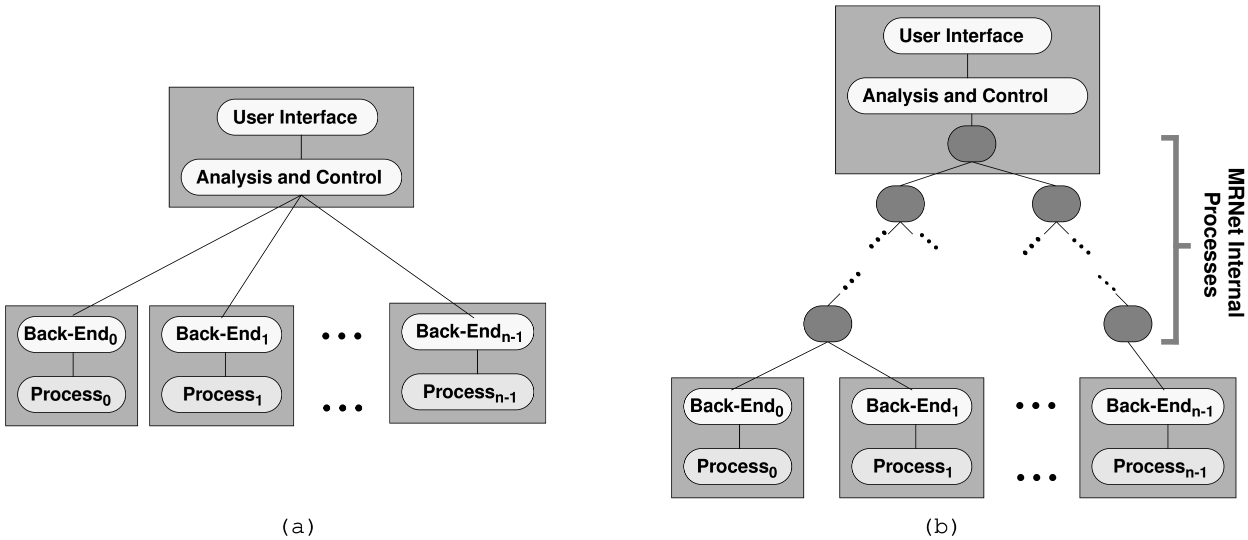


Figure 1: The components of a typical parallel tool (a) and an MRNet-based parallel tool (b).
Shaded boxes show potential machine boundaries.

in parallel as it passes through the network, MRNet can efficiently compute averages, sums, and other more complex aggregations on back-end data.

This type of communication structure has been examined previously (e.g., [3,11,17,20,21,25,28]). However, several features make MRNet especially well-suited as a general facility for building scalable parallel tools:

- **Flexible organization.** MRNet does not dictate the organization of MRNet and tool processes. MRNet process organization is specified in a configuration file that can specify common network layouts like *k*-ary and *k*-nomial trees, or custom layouts tailored to the system(s) running the tool. For example, MRNet internal processes can be allocated to dedicated system nodes or co-located with tool back-end and application processes.
- **Scalable, flexible data aggregation.** MRNet's built-in filters provide efficient computation of averages, sums, concatenation, and other common data reductions. Custom filters can be loaded dynamically into the network to perform tool-specific aggregation operations. For example, Paradyn uses custom filters to implement a scalable algorithm for detecting the clock skew between the tool front-end and each Paradyn daemon. Paradyn also uses a custom histogram filter to place its back-ends into equivalence classes based on the program resources (e.g. functions) discovered by each back-end.
- **High-bandwidth communication.** MRNet transfers data within the tool system using an efficient, packed binary representation. Zero-copy data paths are used whenever possible to reduce the cost of transferring data through internal processes.
- **Scalable multicast.** As the number of back-ends increases, serialization when sending control requests limits the scalability of existing tools. MRNet supports efficient message multicast to reduce the cost of issuing control requests from the tool front-end to its back-ends.
- **Multiple concurrent data channels.** MRNet supports multiple logical *streams* of data between tool components. Data aggregation and message multicast takes place within the context of a data stream, and multiple operations (both upward and downward) can be active simultaneously.

MRNet is part of a larger effort to improve the scalability, reliability, and resiliency of parallel performance and system administration tools. MRNet addresses the problem of non-scalable global data processing and non-scalable global command and control. *Global data processing* is the aggregation of data taken from all processes in an application or nodes in a system, whereas *local data processing* is the collection and analysis of data taken from a single process or system node. Other aspects of our scalability work involve a distributed strategy for automatically finding application performance problems, distributed performance data management, and scalable visualizations of performance analysis results. This paper introduces MRNet and evaluates its scalability; its reliability and resiliency characteristics will be addressed in future work. The context for our work is Paradyn [23], a parallel performance tool supporting automated application performance problem searches.

In the next section, we detail MRNet concepts, implementation, and API. Section 4 presents a quantitative analysis investigating integrating MRNet into the Paradyn performance tool. Section 4 presents a quantitative analysis investigating MRNet's impact on tool scalability. We discuss how MRNet relates to previous work in this area in Section 5.

```
front_end_main(){
1. MR_Network * net;
2. MR_Communicator * comm;
3. MR_Stream * stream;
4. float result;
5. net = new MR_Network(config_file);
6. comm = net->get_broadcast_communicator( );
7. stream = new MR_Stream(comm, FMAX_FIL);
8. stream->send("%d", FLOAT_MAX_INIT);
9. stream->recv("%f", result);
}
```

```
back_end_main(){
1. MR_Stream * stream;
2. int val;
3. MR_Network::init_backend( );
4. MR_Stream::recv("%d", &val, &stream);
5. if(val == FLOAT_MAX_INIT){
6.     stream->send("%f", rand_float);
   }
}
```

Figure 2: MRNet front-end and back-end sample code.

2 The Multicast/Reduction Network

MRNet is a customizable, high-throughput communication software infrastructure for parallel tools. MRNet has two main components: *libmrnet,* a library that is linked into a tool's front-end and back-end components, and *mrnet_comm*, a program that runs on intermediate nodes interposed between the front-end and back-ends. The MRNet library exports an API that enables interaction between the front-end and groups of back-ends via MRNet. The primary purpose of mrnet_comm is to distribute data processing functionality across multiple computer hosts and to implement efficient and scalable group communications. We present an overview of the MRNet architecture, followed by discussions of the interface, internal process implementation, data aggregation mechanisms, system instantiation, and process network topology issues.

2.1 MRNet Overview

The MRNet library, libmrnet, allows a tool to use a network of internal processes as a communication substrate between the tool's front-end and back-end processes. The internal processes are instances of the mrnet_comm program. The connection topology and host assignment of these processes is determined by a configuration file, thus the geometry of MRNet's process tree can be customized to suit the physical topology of the underlying hardware. While MRNet can generate a variety of standard topologies, users can easily specify their own topologies. See Section 2.6 for further discussion on MRNet process topologies.

MRNet uses *communicators* to represent groups of network end-points. Like communicators in MPI [22], MRNet communicators provide a handle that identifies a set of end-points for point-to-point, multicast or broadcast communications. In contrast to MPI applications that typically have a non-hierarchical layout of potentially identical processes, MRNet enforces a tree-like layout of all processes rooted at the tool front-end. Accordingly, MRNet communicators are created and managed by the front-end, and communication is only allowed between a tool's front-end and its back-ends, i.e. back-ends cannot interact with each other directly via MRNet. This limitation reflects the design of current run-time tools but might be relaxed in the future if there appears to be a demand for such interaction.

A *stream* is a logical channel that connects the front-end to the end-points of a communicator. All tool-level communication via MRNet uses streams. Streams carry data packets downstream, from the front-end toward the back-ends, and upstream, from the back-ends toward the front-end. Each stream has a unique *stream id* that is used to identify packets sent on that stream. MRNet uses this stream id to support multiple, simultaneous streams of communication among the same components within a tool instance. However, communication via MRNet between separate tool instantiations is not supported; each tool has its own MRNet network instantiation.

Data packets carry typed data, enabling data aggregation operations to be associated with a stream. Types are specified using a format string similar to that used by C formatted I/O primitives printf and scanf. For example, a packet whose data is described by the format string "%d %f %s" contains an integer, float, and character string. MRNet also adds specifiers for arrays of simple data types.

Data aggregation is the process of transforming multiple input data packets into one or more output packets. Though it is not necessary for aggregation to result in less data or even different data, aggregations that reduce or modify data values are most common. MRNet uses *filters* to aggregate data packets. A filter may be bound to a stream when the stream is created, thus specifying the aggregation operation to perform and the expected type(s) of the data sent on the stream. MRNet uses two types of filters: synchronization filters and transformation filters. Synchronization filters organize data packets from downstream nodes into synchronized waves of data packets, while transformation filters operate on the synchronized data packets yielding one or more output packets.

2.2 MRNet Interface

The MRNet API, provided by libmrnet, consists of network, end-point, communicator, and stream C++ objects that a tool's front-end and back-end use for communication. The network object is used to instantiate the MRNet network and access end-point objects representing available tool back-ends. The communicator object is a container for groups of end-points, and streams are used to send data to the end-points in a communicator.

Simplified code for an example tool front-end and back-end is shown in Figure 2. In the front-end code, after the variable definitions in lines 1-4, an instance of the MRNet network is created in line 5 using the topology specification from `config_file`. At line 6, the newly created network object is queried for an auto-generated broadcast communicator that contains all available end-points. In line 7, this communicator is used to build a stream that will use a "floating point maximum" filter to find the maximum value of floating point data sent upstream. The front-end then might send one or more initialization messages to the back-ends; on line 9 of our example code, we broadcast an integer initializer and await the single floating point value result. The back-end code reflects the actions of the front-end. Each tool back-end first connects to the MRNet network via the `init_backend` call in line 3. In contrast to the front-end's stream-specific `recv` call, the back-ends call a stream-anonymous `recv` that returns both the integer sent by the front-end and a stream object representing the stream that the front-end used to send the data. Finally, each back-end sends a scalar floating point value upstream toward the front-end.

2.3 MRNet Internal Processes

While libmrnet provides access to MRNet capabilities, it is the internal processes of a MRNet tree that provide the core functionality. MRNet internal processes implement logical channels for the flow of control messages and data between the tools components and perform data aggregation or reduction operations as appropriate. Consequently, an internal process' main task is to create and manage these logical channels or streams and correctly control the flow of packets through the system.

Internal processes use a *stream manager* object to manage control flow and route packets. When a stream is established, an internal process creates a new stream manager and initializes it with the set of end-points to be associated with the stream and the filter(s) to be used on upstream data. The stream manager also maintains an appropriate list of *children nodes* for the stream; a child node object represents a connection directly to an end-point or to another internal process through which at least one end-point in the set can ultimately be reached. Figure 3 illustrates the organization of the functional layers within an internal process. We describe these layers by discussing the path that user data packets take on upstream and downstream flows.

Upstream dataflow is the more complex process; it exercises all the layers of internal process functionality bounded by the dashed line in Figure 3. Packets must be unbatched, demultiplexed, synchronized, perhaps aggregated, and re-batched before continuing their upstream journey toward the front-end. Incoming packet buffers must first be unbatched into individual packets. Data packets are batched into packet buffers, which logically represent a series of communications destined for the same process, to allow for fewer larger messages to be sent over busy connections, reducing overall communication costs. Each packet is tagged with its stream id that is used to demultiplex the packets into their appropriate streams. At the demultiplexing phase, packets are passed to the appropriate stream manager instance that delegates control to filter objects for synchronization and aggregation. After aggregation, packets destined for the upstream node are re-batched into a single packet buffer that is then scheduled for transmission to the upstream node in the tree. Note that packets are never copied; they are manipulated by reference. It is only when passing through a transformation filter that data might be moved.

Downward dataflow is simpler since filters are usually not applied to downward flows. Incoming packet buffers only need to be unbatched and routed to appropriate child nodes. Data packets are then re-batched into buffers for the appropriate downstream nodes. Since a single packet may be directed to multiple destinations, the same data packet may be placed in multiple packet buffers. Like the upward path, packets are buffered by reference to avoid copying.

2.4 Filters

Filters operate on data flowing upstream in the network. Synchronization filters receive packets one at a time and do not output any packets until the specified synchronization criteria has occurred. Transformation filters input the group of synchronized packets, perform some type of data transformation on the data contained in the packets and output one or more packets. A distinction between synchronization and transformation filters is that synchronization filters are independent of the packet data type, but transformation filters operate on packets of a specific type.

Synchronization filters provide a mechanism to deal with the asynchronous arrival of packets from children nodes; the synchronizer collects packets and typically aligns them into waves, passing an entire wave onward at the

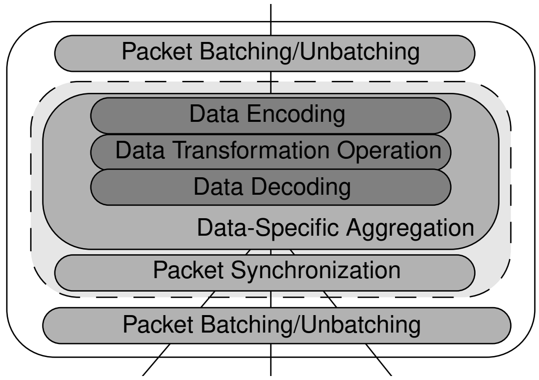


Figure 3: Functional layers within an MRNet internal process.

same time. Therefore, synchronization filters do no data transformation and can operate on packets in a type-independent fashion. MRNet currently supports three synchronization modes:

- *Wait For All*: wait for a packet from every child node;
- *Time Out*: wait a specified time or until a packet has arrived from every child (whichever occurs first); and
- *Do Not Wait*: output packets immediately.

Synchronization filters use one of these three criteria to determine when to return packets to the stream manager. Although we do not anticipate a need for it, new types of synchronization filters can be added by the user.

Transformation filters combine data from multiple packets by performing an aggregation that yields one or more new data packets. Since transformation filters are expected to perform computational operations on data packets, there is a type requirement for the data packets to be passed to this type of filter: the data format string of the stream's packets and the filter must be the same. Transformation operations must be synchronous, but can carry state from one transformation to the next using static storage structures. MRNet provides several transformation filters that should be of general use:

- *Basic scalar operations*: min, max, sum and average on integers or floats
- *Concatenation*: operation that inputs *n* scalars and outputs a vector of length *n* of the same base type.

MRNet is designed to allow tool developers to add new filters to the provided set. This discussion focuses on transformation filters; however, synchronization filters share the same basic design with transformation filters and may be added using similar techniques. In order to establish a new filter, a tool developer must provide a filter function that implements the data transformation operation. Filter functions have the following signature:

```
void filter_func(DataElements **in_elems, int in_size, DataElements ***out_elems, *out_size);
```

The filter function takes an array of data elements and outputs an array of data elements of arbitrary size. Each data element is essentially a C union of type integer, float, character, or a pointer to arrays of these types.

Filter functions implemented by the tool developer must be named and made known to MRNet. Both tasks are accomplished using the load_filter function provided by the MRNet API. load_filter takes the name of a shared object file that contains the filter function to be used by the new filter, and returns an id that identifies the new filter. MRNet requires that each shared object file implement a single filter function with a well-known name, allowing the front-end and internal processes to access the filter function using the operating system's API for managing shared objects (e.g., dlopen and dlsym on UNIX systems).

2.5 MRNet Instantiation

While conceptually simple, creating and connecting the MRNet process network is complicated by interactions with the various job management systems. In the simplest environments, we can launch jobs manually using facilities like *rsh* or *ssh.* In more complex environments, it is necessary to submit all requests to a job management system. In this case, we are constrained by the operations provided by the job manager (and these vary from system to system). We currently support two modes of instantiating MRNet-based tools.

In the first mode of process instantiation, MRNet creates the internal and back-end processes, using the specified MRNet topology configuration to determine the hosts on which the components should be located. First, the front-end consults the configuration and uses rsh or ssh to create internal processes for the first level of the communication tree on the appropriate hosts. Each newly created process establishes a connection to the process that created it. The first activity on this connection is a message from parent to child containing the portion of the configuration relevant to that child. The child then uses this information to begin instantiation of the sub-tree rooted at that child. When a sub-tree has been established, the root of that sub-tree sends a report to its parent containing the end-points accessible via that sub-tree. Each internal node establishes its children processes and their respective connections sequentially. However, since the various processes are expected to run on different compute nodes, sub-trees in different branches of the network are created in concurrently, maximizing the efficiency of network instantiation.

In the second mode of process instantiation, MRNet relies on a process management system to create some or all of the MRNet processes. This mode accommodates tools that require their back-ends to create, monitor, and control the application processes. For example, IBM's POE uses environment variables to pass information, such as the process' rank within the application's global MPI communicator, to the MPI run-time library in each application process. In cases like this, MRNet cannot provide back-end processes with the environment necessary to start MPI application processes. As a result, MRNet creates its internal processes recursively as in the first instantiation mode, but does not instantiate any back-end processes. MRNet then starts the tool back-ends using the process management system to ensure they have the environment needed to create application processes successfully. When starting the back-ends, MRNet must provide them with the information needed to connect to the MRNet internal process tree, such as the leaf processes' host names and connection port numbers. This information is provided via the environment, using shared filesystems or other information services as available on the target system.

2.6 MRNet Process Layout

MRNet allows a tool to specify a node allocation and process connectivity tailored to its computation and communication requirements and to the system running the tool. Choosing an appropriate MRNet configuration can be difficult due to the complexity of the tool's own activity and its interaction with the system. We briefly discuss the issues related to process layout, but because our current work focuses on tool scalability a full treatment of optimal MRNet configurations is beyond the scope of this paper. The configurations we used for our experiments in Section 4 were chosen for their ability to show MRNet's effect on tool scalability. We anticipate future research will examine the issue of MRNet topology in more detail.

When choosing the process configuration for an MRNet-based tool, there are two key issues to consider: whether the MRNet internal processes are co-located with the application processes under study, and how the internal processes are connected. Our primary measures of a configuration's quality are its: (1) latency for a single broadcast operation, measured from initiation by the front-end to the last receipt by a back-end; (2) latency for a single data aggregation operation, measured from initiation by the back-ends to receipt by the front-end; (3) throughput for streams of broadcasts and data aggregations; and (4) CPU utilization of the MRNet internal processes.

The first issue to consider when choosing an MRNet configuration is whether to co-locate MRNet internal processes and application processes on the same nodes. While the literature on broadcast/reduction networks assumes that internal processes will be co-located with application processes, we believe this approach has serious flaws in practice. First, the internal processes would contend with application processes for CPU and network resources, perhaps seriously impacting the application's performance. Second, differing loads across MRNet internal processes could create an imbalance among the application processes, skewing their performance. Because a parallel program's speed is often limited by its slowest process, this performance skew would increase the tool's impact on the application. As a result, we recommend that MRNet's internal processes be located on resources distinct from those running the application processes. Regardless of whether the MRNet internal processes and application processes are co-located or are run on distinct nodes, their overall resource usage is similar. Therefore, we advocate separate location to achieve more predictable and understandable application behavior.

The second issue to consider when choosing an MRNet configuration is the internal process topology. Both balanced and unbalanced tree topologies have attractive properties for MRNet configurations. The literature on parallel collective communication algorithms argues for unbalanced tree topologies in many situations. For example, Bernaschi and Iannello [5] show that the optimal communication tree for broadcast is somewhere between a single-level flat tree and a binomial tree, depending on the latency for transferring messages between processes and the minimum interval between message send operations in a process. Similarly, optimal algorithms for several broadcast and data aggregation problems evaluated under the LogP [8,16] and LogGP [3] models use unbalanced communication trees. Unfortunately, this literature assumes all processes involved in the operation are data sources (for reductions) or sinks (for broadcasts), which is not the case for MRNet's internal processes.

Balanced tree topologies provide several attractive advantages over unbalanced tree topologies for our work. Their regularity makes them easier to analyze when choosing the most appropriate size and shape for the MRNet internal process tree. Also, although the latency of individual collective communication operations may be worse with balanced trees than unbalanced trees, they can provide better throughput for sequences of collective communica-

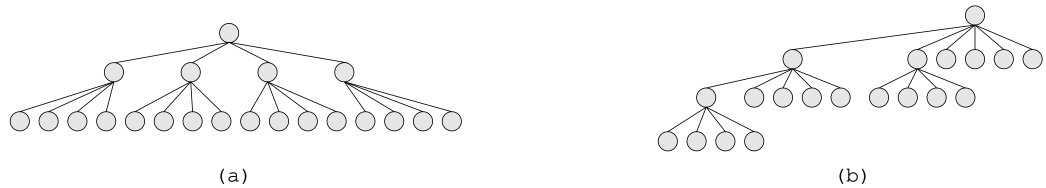


Figure 4: Comparable MRNet internal process topologies with the same number of back-ends.

The latency of a single broadcast or aggregation operation might be better with the unbalanced topology (b), but the balanced topology (a) has better throughput for pipelined operations.

tion operations. For example, consider the MRNet tree topologies shown in Figure 4 connecting a tool front-end to sixteen tool back-ends. Assuming a LogP model with a minimum gap g between successive send operations in a process, an overhead o for each send and receive, and a message transfer latency L, the time required to complete a broadcast operation to all sixteen back-ends using the balanced tree topology shown in Figure 4a is $8g+4o+2L$, but the tool can start a new broadcast each $4g$ cycles. A comparable unbalanced tree topology reaching sixteen back-ends is shown in Figure 4b. This topology is constructed from a binomial tree with four nodes providing low-latency broadcast to each binomial tree node, with four MRNet back-ends attached to each binomial tree node. Depending on the relative values of g, o, and L, a single broadcast operation using this topology may complete before the balanced tree's broadcast, but a tool using this topology needs at least $6g$ cycles between each broadcast operation due to the larger fan-out at the tree's root. Furthermore, if the tool supports six-way fan-out as is being used at the root of the unbalanced tree topology, then it could use a balanced topology with a six-way fan-out throughout the tree to reach far more than sixteen tool back-ends. Therefore, in this paper we chose to experiment using balanced tree topologies, leaving an examination of unbalanced trees and optimal communication topologies for future work. Because the ability of each internal process to keep up with its upward and downward data flow, the fan-out at each internal process is limited. Therefore, our experiments use multi-level balanced trees with moderate fan-outs of four and eight.

3 A Real-World Tool Example

To evaluate MRNet's usefulness for building real-world scalable parallel tools, we modified the Paradyn parallel performance tool to use MRNet. There are two main ways that Paradyn can use MRNet: to simplify the complex interactions between front-end and tool daemons during process start-up and initialization, and to off-load the performance data processing tasks from the Paradyn front-end. Here we report on our experience with using MRNet in the more complex and demanding case of tool initialization. A quantitative evaluation of this use is presented in Section 4.2.

Tools such as debuggers and performance tools may transfer large amounts of data during tool start-up when they create or attach to an application's processes. For example, a debugger that sets breakpoints by function name might deliver the names and addresses of all functions to the tool's user interface. In parallel tools that follow the process organization shown in Figure 1a, the front-end becomes a bottleneck when connected to a large number of application processes. Besides reducing tool interactivity, the start-up latency caused by this bottleneck may create problems for parallel runtime systems that fail if the application processes are not created in a timely fashion. Our modified version of Paradyn uses both built-in and custom MRNet aggregation filters for all activities involving the tool's daemons (i.e., its back-ends) during the tool start-up phase, including:

- reporting data about Paradyn daemons to the front-end;
- distributing data about known performance data metrics to all daemons;
- detecting clock skew between the front-end process and each daemon process; and
- reporting data about application processes to the front-end.

Although most of these activities manipulate Paradyn-specific data, our techniques for using MRNet to implement them are applicable to many activities commonly performed by parallel tools.

During Paradyn start-up, most of the data transferred within the tool system can be placed into one of two categories: data describing the daemon and application processes sent from the back-ends to the front-end, and configuration data sent from the front-end to all back-ends. At tool start-up, the Paradyn back-ends examine application processes to identify the relevant parts of the program, such as modules, functions, and process ids. Such items are called *resources* in Paradyn terminology. Once the back-ends have identified application resources, they are reported to the front-end along with statically-determined call-graphs for all application processes. The bulk of the start-up information sent from the front-end to the back-ends is a collection of performance metric definitions that specify how to instrument processes to collect performance data.

Paradyn uses MRNet in two ways to reduce the cost of reporting data from daemons to the front-end. The method used depends on whether the data is likely to be the same across a significant number of processes (e.g., function names and their addresses) or is likely to be different across processes (e.g., process ids and host names). If the data is likely to be the same across a significant number of processes, then most of the data transferred during tool start-up is redundant (especially if the application processes are created from a small number of executables and run on a collection of homogeneous hosts). To report this data, each Paradyn daemon first computes a summary of the data (i.e., a checksum). Next, the daemons write the checksums to an MRNet stream created to use a custom binning filter. This filter partitions the daemons into equivalence classes based on their checksum values. When the front-end receives the final set of equivalence classes, it requests complete function resource information only for each class' representative process. Unlike function names, data like process identifiers and host names are likely to be different across hosts. Nevertheless, Paradyn also leverages MRNet for reporting this data. Paradyn uses a parallel concatenation aggregation to construct larger resource report messages that are more efficiently delivered by the underlying communication subsystem than many small resource report messages.

Paradyn uses MRNet to deliver configuration data efficiently from the front-end to all back-ends. In Paradyn, metric definitions describing how to instrument processes to collect metric performance data are provided to the front end in a configuration file written in the Paradyn Metric Definition Language [15]. The front-end uses simple broadcast operations to deliver the metric definitions to all tool back-ends.

Clock skew detection is the only start-up activity that does not fall neatly into the two communication paradigms mentioned earlier. The MRNet-based clock skew detection scheme occurs in two phases. The first phase consists of repeated broadcast/reduction pairs on a special stream reserved for finding clock "local" clock skew between each process and the downstream processes to which it is directly connected (i.e., its children in the MRNet process tree). The second phase consists of a single broadcast to all daemons requesting them to initiate the collection of skew results. Each daemon initializes its "cumulative skew" value to zero, and passes it upstream into the MRNet network. When an MRNet internal process receives a cumulative skew value from one of its downstream connections, it adds its observed local clock skew value for that connection to the cumulative value, thereby computing the skew of its clock with each daemon reachable along that connection. By induction, when the algorithm finishes the Paradyn front-end holds the skews between its clock and the clocks of each tool back-end.

4 Evaluation

To evaluate MRNet, we measured its performance alone within a test harness and then integrated with Paradyn, a real-world parallel performance tool. Our micro-benchmark experiments with the test harness tool measured MRNet's start-up latency, the round-trip latency of a single broadcast followed by a reduction, and MRNet's reduction throughput using several process tree topologies. Our Paradyn experiments compared the performance of important start-up activities with and without MRNet. Our experiments were run on the ASCI Blue Pacific system [19] at Lawrence Livermore National Laboratory. Blue Pacific contains 280 nodes (256 compute nodes) connected by an IBM SP switch interconnect. Each node contains four 332 MHz PowerPC 604e processors, 1.5 GB RAM, and runs AIX 5.1.0 with Parallel System Support Programs version 3.4. Our results show that MRNet significantly improves the scalability of key activities in parallel performance and system administration tools.

4.1 Micro-benchmark Results

We began by measuring the low-level performance of MRNet within a minimal test harness. For each run of our test harness tool, we requested an appropriately-sized partition from the Blue Pacific batch scheduling system. Once we were given our partition, we determined the partition nodes' host names and used an automatic configuration generator program to build an MRNet configuration file with the desired topology within the partition. We then executed the tool's front-end program, passing the configuration file's name as an argument. During each run of the test harness, we measured three MRNet performance characteristics: the latency to instantiate the MRNet network, the latency of a broadcast operation followed by a data reduction, and the MRNet's throughput during a sequence of data reductions. The results of these experiments are shown in Figure 5.

Our micro-benchmark measurements show the necessity of infrastructure like MRNet for building scalable parallel tools. Using a flat, single-level topology (which closely approximates the architecture of many parallel tools), instantiation latency grows quickly as the number of tool back-ends increases due to the serialization of the process creation operations. The instantiation latency grows quite slowly when using MRNet with fully-populated balanced tree topologies with four- and eight-way fan-outs because MRNet creates the process tree in parallel. The round-trip latency and data reduction throughput measurements also show the benefits of MRNet to parallel tools. In the flat topology, each broadcast or reduce is implemented using serialized point-to-point message transfers. Although each message transfer is less time-consuming than the rsh used to create processes during tool instantiation, the effect of

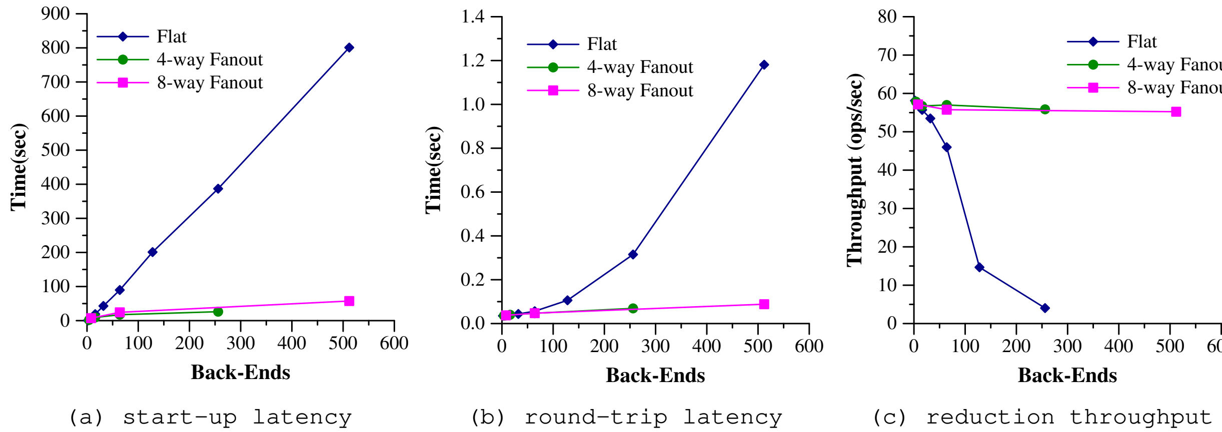


Figure 5: MRNet micro-benchmark experiment results.

Tool instantiation latency (a), round-trip latency of a single broadcast followed by a single reduction (b), and data reduction throughput (c) using single- and multi-level MRNet topologies. Compared to the single-level topology commonly found in parallel tools, multi-level MRNet topologies exhibited dramatically better scalability and overall performance, showing the necessity of multi-level process networks like MRNet for building scalable parallel tools.

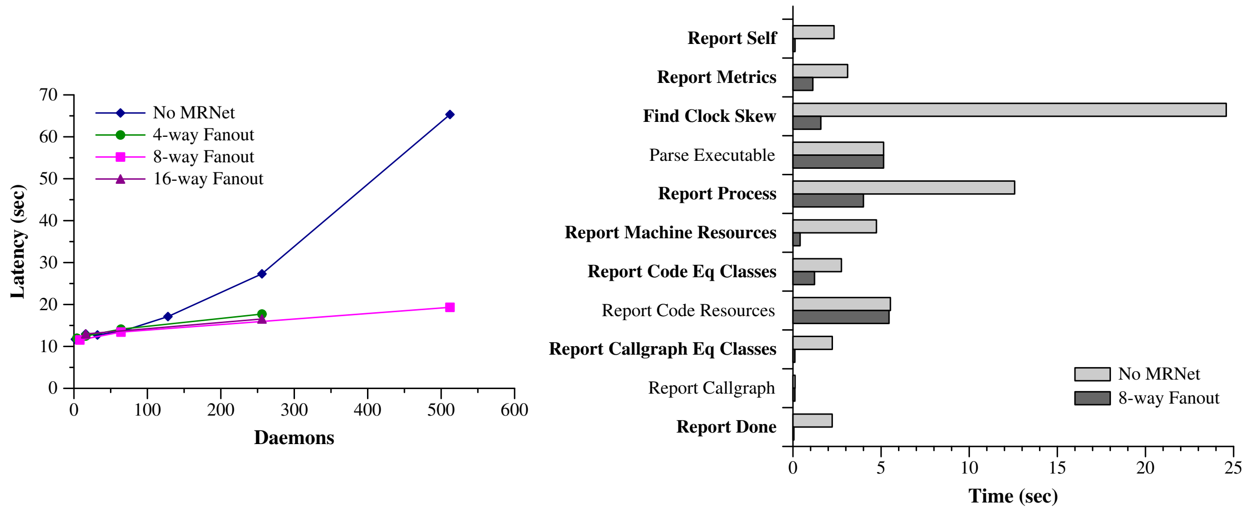
serialization is similar: the latency grows rapidly as the number of back-ends increases. Also, the tool front-end is involved in every message transfer, so it cannot start a subsequent reduction before the previous operation completes. Multi-level MRNet process configurations allow MRNet to perform point-to-point message transfers in parallel. Furthermore, the moderate fan-outs at each MRNet process allows data reductions to be pipelined as they pass through the network, keeping reduction throughput high as application size increases. The trends in MRNet's micro-benchmark scalability studies are perhaps to be expected; previous tool infrastructures using a hierarchy of processes such as the Ladebug parallel debugger [4] and Lilith [11] show similar scalability trends.

4.2 Integrated Performance Results

To evaluate MRNet's real-world performance, we modified the Paradyn parallel performance tool to use MRNet during tool start-up as described in Section 3. Paradyn's start-up protocol was already highly tuned to reduce redundant data transfer. For several data transfers from tool daemons to the front-end, it used a technique whereby each tool daemon computes a checksum over its own data, the front-end partitions the daemons into equivalence classes based on the checksum values, and then requests the complete data from only a single representative of each equivalence class. We measured the latency of Paradyn's start-up activities when preparing to monitor smg2000 [6], a parallel linear equation solver. The smg2000 executable is relatively small, containing approximately 434 functions in a 290 KB executable. We started the timer when all daemons were known to have been started (but not yet reported themselves to the tool front-end), and stopped the timer after the daemons had reported information about themselves and the application processes they created, and were ready to run the application.

The results of our scalability study with several MRNet topologies are shown in Figure 6a. Without MRNet, serialization of the communication between Paradyn's front-end and daemons causes overall start-up latency to rise exponentially as the number of daemons increases. Using MRNet and process topologies with moderate fan-outs, the start-up latency curves are much flatter and growth is nearly linear, indicating a significant improvement in overall tool scalability. To investigate how much of the overall start-up latency that MRNet could affect, we measured the latency of individual start-up activities with and without MRNet for our largest experimental configuration; these results are shown in Figure 6b. The individual activities shown in the figure are:

- **ReportSelf:** Using an MRNet concatenation filter, each daemon reports basic characteristics to the front end such as the host on which it is running;
- **ReportMetrics:** The front-end broadcasts Metric Definition Language data to all daemons; the daemons respond using the equivalence class algorithm described above to report all metrics that they support (including internal metrics not specified in the MDL data);
- **Find Clock Skew:** The front-end finds its clock skew with respect to each daemon using the clock skew detection algorithm described in Section 3;
- **Parse Executable:** Each daemon examines the application executable and the shared libraries it uses to find names and addresses of all functions, and parses the code to discover the application's static call graph;
- **Report Process:** After creating or attaching to an application process, each daemon reports data about the pro-



(a) (b)

Figure 6: Paradyn start-up latency for increasing numbers of daemons (a) and by activity for 512 daemons (b). *In (b), bold activity names indicate use of MRNet for data aggregation or concatenation for some part of the activity.*

cess to the front end including its process id, its command-line arguments, whether it was created by the daemon or was already created when the daemon attached to it, and whether the front-end should issue the command to resume the process when all start-up activities are complete;

- **Report Machine Resources:** Using a concatenation filter, each daemon defines Paradyn resources for the host, process, and initial thread of its application processes via Paradyn's resource definition protocol;
- **Report Code Eq Classes** and **Report Code Resources:** Using the equivalence class algorithm, the daemons define resources for all functions and modules in the application executable;
- **Report Callgraph Eq Classes** and **Report Callgraph:** Using the equivalence class algorithm, the daemons report their static call-graph information (built during the "Parse Executable" activity described above) to the front-end; and
- **Report Done:** The daemons indicate the end of the start-up phase.

Each activity that used MRNet to communicate with all daemons showed a significant latency reduction by using MRNet. The activities that did not show a significant improvement from using MRNet are activities that consist either of work done entirely in parallel by the daemons ("Parse Executable") or point-to-point communication between a small number of daemons and the front-end ("Report Code Resources", "Report Callgraph"). In fact, the point-to-point communication activities transferred data via MRNet; the additional overhead of passing through intermediate MRNet processes was observed to be negligible. Overall, the benefit of using MRNet increased as we increased the number of tool daemons. With our largest configuration of 512 back-ends, the latency for performing all start-up activities was 3.4 times faster with MRNet and a balanced, fully-populated tree configuration with eight-way fan-out than without MRNet. Based on our investigation of MRNet's benefit for each individual activity during Paradyn start-up, we expect this trend to continue with configurations significantly larger than 512 daemons.

Clock skew detection was the Paradyn start-up activity that benefitted most from using MRNet, because it uses repeated broadcast/reduction operations to distribute and collect clock samples and intermediate skew results whereas the other activities perform only one or two collective operations. We evaluated the clock skews computed by the MRNet-based clock skew detection algorithm by comparing them to skews computed using Blue Pacific's SP switch clock (a globally-synchronous clock) and to skew results computed using a commonly-used direct-communication scheme. To compute its clock skew with respect to a given daemon under the direct communication scheme, the front-end sends a small amount of data to the daemon. The daemon samples its clock when it receives the data and sends this sample to the front-end. When the front-end receives the daemon's sample, it samples its own clock and computes the round-trip latency of the sends and receives. The front-end approximates the one-way latency from the round-trip latency, adds the one-way latency to the daemon's clock sample, and uses the difference between this value and the front end's receive timestamp as the clock skew. In our experiments, the front-end measured the skew using the direct communication scheme 100 times and used the observed skew with the smallest absolute value as the actual clock skew. Using a 64-daemon topology with four-way fan-out (a three-level topology), the MRNet-based clock skew detection algorithm produced skews with an average error of 10.5% as compared to the skews computed using the globally-synchronous switch clock, while the average error in the skews produced by the direct-connection method was 17.5%. However, the standard deviation of the errors produced by the MRNet-based algorithm was 80.4, slightly higher than the standard deviation in the direct connection method's errors at 78.9. In short, MRNet's clock skew detection algorithm produced results comparable to the direct-connection method but is significantly more scalable.

5 Related Work

MRNet provides data aggregation and multicast services for building scalable parallel tools. Similar functionality has been found previously in software-based collective communication infrastructure for parallel tools and applications, and in parallel databases and overlay networks.

MRNet, Lilith [11], and Ygdrasil [3] are parallel tool infrastructures providing multicast and data aggregation functionality. MRNet differs from Lilith and Ygdrasil in its communication model, tool architecture, and software engineering trade-offs. In Lilith's communication model, synchronous waves of messages are sent to or from the tool's front-end at the root of the process tree [12]. Generalizing the multicast/reduction capabilities of the Ladebug [4] parallel debugger, Ygdrasil is best suited to a synchronous request/response model for tools like parallel debuggers. In contrast, MRNet's communication model supports multiple simultaneous asynchronous collective communication operations. Tools built with MRNet and Ygdrasil share a similar architecture with internal processes distinct from the tool's back-ends. Lilith's architecture allows tool back-end code at each process throughout the Lilith process network. For tool extensibility, both Lilith and Ygdrasil are implemented in Java and take advantage of that language's natural ability to load code dynamically. MRNet trades this ease of extensibility for the higher potential data throughput of C++-based data serialization.

A network of processes as is used in MRNet is often called an *overlay network* because it defines a logical network that overlays a physical network. Several overlay network projects have data aggregation functionality similar to MRNet. Ganglia [21] defines a hierarchical overlay network like MRNet's in an infrastructure for monitoring clusters and federations of clusters, and Supermon [25] servers can be organized into a hierarchical infrastructure for data aggregation. Neither of these systems is designed to support high throughput, and would be ill-suited for collecting and manipulating application performance data sampled with high frequency. Also, Ganglia relies on the availability of IP multicast within clusters which may not be enabled for all target systems.

Data aggregation has also been studied in the context of parallel databases. Shatdal and Naughton [24] suggest several algorithms for efficient data aggregation in parallel databases. Gray et al [14] suggest ways for efficiently implementing their "data cube" aggregation operator. Neither approach uses a separate network of aggregator processes as is used in MRNet. Like a parallel database, TAG [20] provides a SQL-based interface for expressing data aggregation queries, and a relational database model for representing aggregation results collected from wireless sensor networks. Similar to MRNet, TAG supports multiple simultaneous aggregation operations and supports streams of aggregated data in response to an aggregation request. However, TAG only supports ordinal data aggregation, whereas MRNet's flexibility allows filters that align and aggregate timestamped data. TAG uses a SQL/relational interface, in contrast to our RPC-style interface. Also, TAG organizes its sensors with an ad-hoc routing tree, whereas MRNet's network configuration is specified *a priori* via a configuration file.

Most work in software-based collective communication has focused on providing multicast and data aggregation support for applications. The Message Passing Interface [22] standard defines broadcast and a few data reduction operations. Whereas some MPI implementations use serialized point-to-point operations to implement these collective operations, others provide optimized implementations. For example, MagPIe [17] provides MPI collective communication primitives optimized for applications run in a geographically-distributed environment like the Grid. MagPIe uses a process tree consisting of a flat, single-level tree at the root for efficient communication across a WAN, followed by a binary tree for efficient communication within the local network. As another example, the ACCT [27] system automatically tunes its MPI collective communication algorithms based on modelling and experimental results, tailoring the algorithms to the system on which the MPI application runs. Unfortunately, because optimized MPI implementations are not universally available, we cannot depend on the availability of a high-performance MPI layer for efficient collective communication in parallel tools. Also, MPI reductions are more restrictive than MRNet's data aggregations because they are applied ordinally to the operands. Finally, a tool's use of MPI may conflict with MPI use in the monitored application. For example, in a common tool start-up scenario, a process manager creates tool back-end processes, which then create application processes. The back-end processes are supposed to be transparent to the process manager, but may not be if they are also MPI-based programs. MRNet does not use MPI for collective communication, so it is safe to use in tools that monitor MPI applications. We would advocate using MRNet as a substitute for MPI's implementation for efficient broadcast and data reduction support.

Acknowledgments

This paper benefited from the hard work of many past and present members of the Paradyn research group. We especially wish to thank Victor Zandy and Bryan Wylie for several fruitful discussions on the topic. We also thank John Gyllenhaal, Jeff Vetter, and Chris Chambreau for help with the computing environment on ASCI Blue Pacific.

References

[1] Advanced Simulation and Computing program, National Nuclear Security Administration, United States of America Department of Energy. <http://www.nnsa.doe.gov/asc/home.htm>, February 6, 2003.

[2] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model. *Journal of Parallel and Distributed Computing* **44**, 1, July 1997, pp. 71–79.

[3] Susanne M. Balle. Personal communication, November 2002.

[4] S.M. Balle, B.R. Brett, C.-P. Chen, and D. LaFrance-Linden. A New Approach to Parallel Debugger Architecture. *Sixth International Conference PARA 2002*, Espoo, Finland, June 2002. Published as *Lecture Notes in Computer Science* **2367**, J. Fagerholm et al (Eds), Springer, Heidelberg, June 2002, pp. 139–149.

[5] M. Bernaschi and G. Iannello. Collective Communication Operations: Experimental Results vs. Theory. *Concurrency: Practice and Experience* **10**, 5, April 1998, pp. 359–386.

[6] P.N. Brown, R.D. Falgout, and J.E. Jones. Semicoarsening Multigrid on Distributed Memory Machines. *SIAM Journal on Scientific Computing* **21**, 5, 2000, pp. 1823–1834.

[7] Center for Computational Research, University at Buffalo, The State University of New York. <http://www.ccr.buffalo.edu>, February 6, 2003.

[8] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *ACM SIGPLAN Notices* **28**, 7, July 1993, pp. 1–12.

[9] Earth Simulator Center. <http://www.es.jamstec.go.jp>, February 6, 2003.

[10] Etnus LLC, "TotalView User's Guide", Document version 6.0.0-1, January 2003. <http://www.etnus.com>

[11] D.A. Evensky, A.C. Gentile, L.J. Camp, and R.C. Armstrong. Lilith: Scalable Execution of User Code for Distributed Computing. *Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC '97)*, Portland, Oregon, August 1997, pp. 306–314.

[12] D.A. Evensky. Personal communication, November 2001.

[13] Forecast Systems Laboratory, National Oceanic and Atmospheric Administration. <http://hpcs.fsl.noaa.gov>, Feb 6, 2003.

[14] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery* **1**,1, April 1997, pp. 29–53.

[15] J.K. Hollingsworth, B.P. Miller, M.J.R. Goncalves, O. Naim, Z. Xu, and L. Zheng. *International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, San Francisco, California, November 1997, pp. 201–213.

[16] R.M. Karp, A. Sahay, E.E. Santos, and K.E. Schauser. Optimal Broadcast and Summation in the LogP Model. *Fifth ACM Symposium on Parallel Algorithms and Architectures*, Velen, Germany, June 1993, pp. 142–153.

[17] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, R.A.F. Bhoedjang. MagPIe: MPI's Collective Communication Operations For Clustered Wide Area Systems. *ACM SIGPLAN Notices* **34**, 8, August 1999, pp. 131–140.

[18] Lawrence Livermore National Laboratory. Multiprogrammatic Capability Cluster. <http://www.llnl.gov/linux/mcr>, February 6, 2003.

[19] Lawrence Livermore National Laboratory. Using ASCI Blue Pacific. <http://www.llnl.gov/asci/platforms/bluepac>, February 13, 2003.

[20] S. Madden, M.J. Franklin, J.M Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, Massachusetts, December, 2002.

[21] M.L. Massie, B.N. Chun, and D.E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. University of California, Berkeley Technical Report, <http://ganglia.sourceforge.net/talks/parallel_computing/ganglia-twocol.pdf>, February 2003.

[22] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputing Applications* **8**, 3/4, Fall/Winter 1994.

[23] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* **28**, 11, November 1995, pp. 37–46.

[24] A. Shatdal and J.F. Naughton. Adaptive Parallel Aggregation Algorithms. *ACM SIGMOD Record* **24**, 2, May 1995, pp. 104–114.

[25] M.J. Sottile and R.G. Minnich. Supermon: A High-Speed Cluster Monitoring System. *Cluster 2002*, Chicago, Illinois, September 2002.

[26] UoE HPCX Ltd. <http://www.hpcx.ac.uk>, February 6, 2003.

[27] S.S. Vadhiyar, G.E. Fagg, and J. Dongarra. Automatically Tuned Collective Communications. *2000 ACM/IEEE Conference on Supercomputing (SC2000)*, Dallas, Texas, November 2000.

[28] A. Waheed, D.T. Rover, and J.K. Hollingsworth. Modeling and Evaluating Design Alternatives for an On-Line Instrumentation System: A Case Study. *IEEE Transactions on Software Engineering* **24**, 6, June 1998, pp. 451–470.