Benchmarking the MRNet Distributed Tool Infrastructure: Lessons Learned

Philip C. Roth, Dorian C. Arnold, and Barton P. Miller
Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706-1685 USA
{pcroth,darnold,bart}@cs.wisc.edu

Abstract

MRNet is an infrastructure that provides scalable multicast and data aggregation functionality for distributed tools. While evaluating MRNet's performance and scalability, we learned several important lessons about benchmarking large-scale, distributed tools and middleware. First, automation is essential for a successful benchmarking effort, and should be leveraged whenever possible during the benchmarking process. Second, micro-benchmarking is invaluable not only for establishing the performance of low-level functionality, but also for design verification and debugging. Third, resource management systems need substantial improvements in their support for running tools and applications together. Finally, the most demanding experiments should be attempted early and often during a benchmarking effort to increase the chances of detecting problems with the tool and experimental methodology.

1. Introduction

The desire to solve large-scale problems has driven the development of increasingly large parallel and distributed computing resources. Performance, debugging, and system administration tools that work well in small-scale environments usually fail to scale as systems and applications get larger or more heterogeneous.

MRNet [10] is a parallel tool infrastructure designed to reduce the cost of many important tool activities. MRNet provides scalable multicast and data aggregation support designed especially for distributed tools. In contrast to the typical parallel tool organization shown in Figure 1a, MRNet-based tools incorporate a tree of processes between the tool's front-end and its back-ends (commonly called tool daemons) as shown in Figure 1b. Communication within MRNet-based tools occurs over logical data *streams*; data sent along a stream may be manipulated by *filters*.

We evaluated the performance and scalability of MRNet by measuring its performance within a simple test harness tool and within a real-world tool. The simple test tool provided a controlled environment for verification, benchmarking of fundamental multicast and data aggregation operations, and debugging. Integrating MRNet into the Paradyn [7] parallel performance tool provided opportunities not only for benchmarking MRNet when performing complicated (and often surprising) collective communication operations like performance data aggregation and clock skew detection, but also for identifying changes to the tool and our experimental methodology to ease the task of benchmarking large-scale parallel tools.

While evaluating the performance and scalability of MRNet, we learned several important lessons about benchmarking tools in large-scale environments. First, we found that automation is essential for a successful benchmarking effort and may be used throughout the benchmarking process. In our benchmarking effort, we leveraged automation by using a batch resource management system for scheduling our experiments, by using shell scripts to generate batch jobs for an entire scalability study at once, and by using scripting functionality to control our tool during an experiment run. We also found that leveraging automation in the benchmarking process may require substantial modifications to the tool, e.g., to remove the tool's graphical user interface so it can be used within a batch system. Second, we learned the value of micro-benchmark experiments. Although our most important goal was to show MRNet's scalability in a real-world

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.

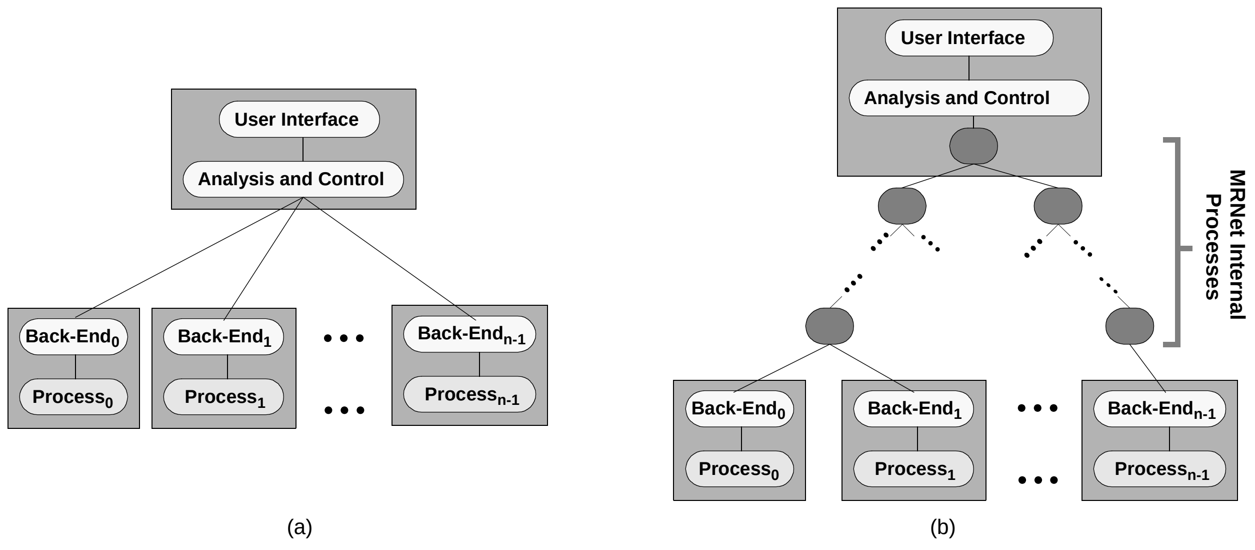


Figure 1: The components of a typical parallel tool (a) and an MRNet-based parallel tool (b). Shaded boxes show potential machine boundaries.

setting, examining MRNet within a simple test tool was invaluable not only for gauging the performance of low-level MRNet operations, but also for verifying its design and for debugging our initial implementation and experimental methodology. Third, we identified the critical need for improved tool support in resource management software. In our target environment, we found the resource management system lacked support for running applications under tool control, forcing us to rely on clumsy, ad hoc workaround schemes to perform our benchmarking experiments. Finally, we noted the debugging value of trying large-scale experiments early and often during the benchmarking process. Too often, we increased the scale of our experiments only to expose bugs in our software or experimental methodology that were hidden at the smaller scales.

This paper is organized as follows. In Section 2 we provide an overview of the MRNet infrastructure. We present the lessons we learned while benchmarking MRNet in Section 3, and we summarize lessons we learned during our benchmarking effort in Section 4.

2. MRNet

MRNet is software infrastructure that provides scalable group communication and customizable data aggregation for distributed tools. Tools use MRNet by linking with the MRNet library that provides the MRNet API. The MRNet API encapsulates interactions with a system of MRNet internal processes that form a tree which connects the tools front-end to its back-ends. This tree of processes enables scalable group multicast and reduction communication. The internal process' ability to apply filters that manipulate in-flight data also allow tools and applications to distribute their data processing functionality.

In this section, we present an overview of MRNet; a previous paper [10] describes MRNet's features, interface, implementation, and quantitative performance in detail.

2.1. MRNet interface

The MRNet library exports an API that allows a tool to use a network of internal processes as a communication substrate between the tool's front-end and back-end components. The MRNet API consists of *network, end-point, communicator,* and *stream* C++ objects.

The network object is used to instantiate the MRNet network of processes and access end-point objects that represent tool back-ends. The geometry of MRNet's network can be customized via a configuration file that specifies the connection topology and host assignment of the MRNet internal processes. Although MRNet can automatically generate a variety of standard topologies, users can specify custom MRNet topologies based on the physical topology of the underlying hardware for performance, load-balancing, or other reasons. The MRNet communicator object is a container for groups of end-points; communicators provide a handle that identifies a set of end-points for point-to-point, multicast or broadcast communication. A MRNet stream is a logical channel that connects the front-end to the end-points of a communicator. MRNet streams transport tool-level data packets downstream, from the front-end toward the back-ends, and upstream, from the back-ends toward the front-end.

```
    front_end_main(){
1.      Network * net;
2.      Communicator * comm;
3.      Stream * stream;
4.      float result;

5.      net = new Network(config_file);
6.      comm = net->getBroadcastComm( );
7.      stream = new Stream(comm, FMAX_FIL);
8.      stream->send(“%d”, FLOAT_MAX_INIT);
9.      stream->recv(“%f”, result);
    }
```

(a)

```
    back_end_main(){
1.      Stream * stream;
2.      Network * net;
3.      int val;

4.      net = new Network();
5.      net->recv(“%d”, &val, &stream);
6.      if(val == FLOAT_MAX_INIT){
7.          stream->send(“%f”,rand_float);
        }
    }
```

(b)

Figure 2: Example MRNet front-end (a) and back-end code (b).

After the MRNet network is instantiated, a tool can group end-points into a communicator, use that communicator to create a new stream, and use the newly created stream to send and receive data among the components connected by the stream. A simplified example of this 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. Filters are discussed in Section 2.3.. MRNet data packets carry typed data with types specified using I/O primitives and format strings similar to those used with the `printf`/`scanf` functions. In the example, `“%d”` and `“%f”` are used to specify an integer and a floating point scalar, respectively, in the send and receive calls. MRNet also adds specifiers for arrays of simple data types. In lines 8 and 9, we broadcast an integer initializer and await the single floating point value result. The back-end code reflects the actions of the front-end. A tool’s back-end connects to the MRNet network by creating a network object (with no configuration file) as in line 4. In contrast to the front-end’s stream-specific `recv` call, back-ends call a version of `recv` that returns both the integer *message tag* 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.

2.2. MRNet internal processes

MRNet internal processes implement streams, the logical channels over which data flows through the system. In addition to data packet routing and forwarding, the internal processes must appropriately apply filters to the data flow. When streams are established at the front-end, control messages are sent through the MRNet network to identify the components that are a part of the stream and the filter(s) to be used on data packets sent on the stream. A packet’s header contains an identifier which determines the stream to which the packet belongs and hence the filters which must be applied to the packet and internal processes or end-points to which the packet should be forwarded.

MRNet employs various techniques to provide high-throughput communication. *Packet batching* groups a series of packets destined for the same process into fewer larger messages to reduce communication overhead. MRNet also uses *zero-copy data paths;* as a packet passes through the functional layers of an internal process, it is manipulated by reference whenever possible to avoid unnecessary copying.

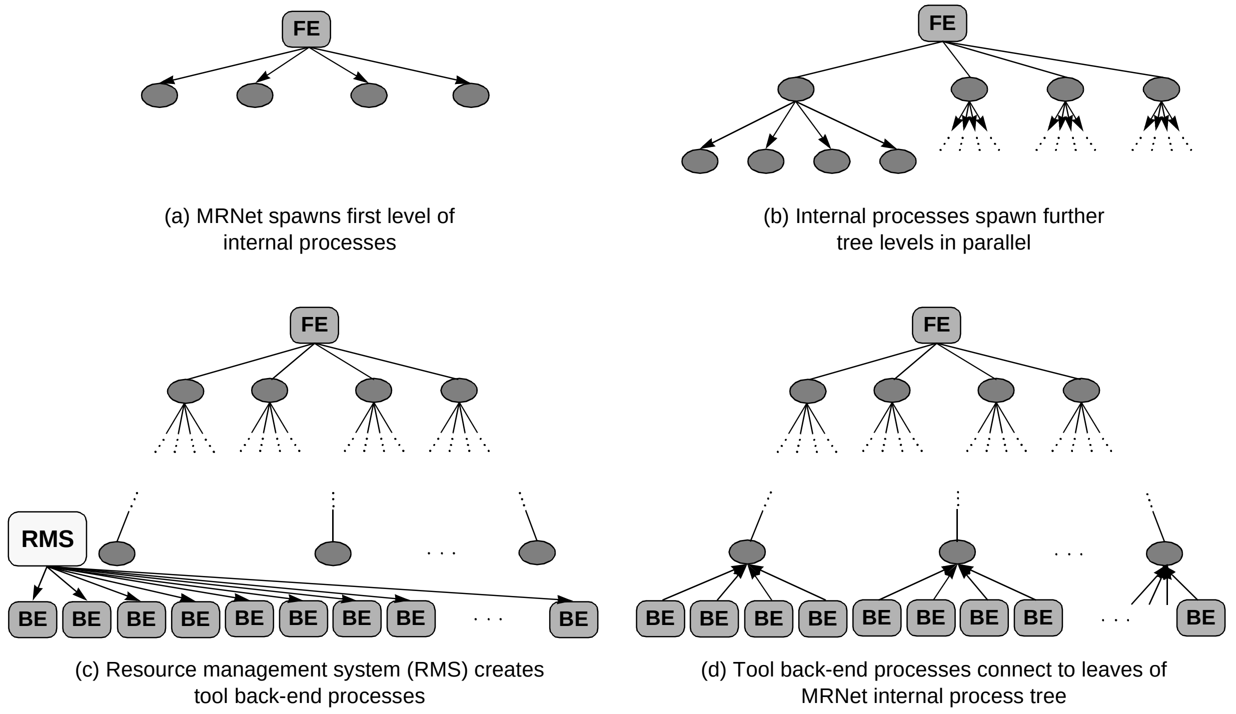
2.3. MRNet filters

Data aggregation is the process of transforming multiple input data packets into one or more output packets. MRNet uses *filters* to aggregate data packets. Filters are bound to a stream when it is created. MRNet distinguishes *synchronization filters*, which receive packets one at a time and do not output any packets until specified synchronization criteria are satisfied, and *transformation filters*, which input a group of packets, perform some type of data transformation on the data contained in the packets and output one or more packets. Synchronization filters are independent of the packet data type; transformation filters operate on packets of a specific type.

In MRNet, synchronization filters are the mechanism to deal with the asynchronous arrival of packets from children nodes. 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; and
- *Do Not Wait*: output packets immediately.

Transformation filters combine data from multiple packets by performing an aggregation that yields one or



(a) MRNet spawns first level of internal processes

(b) Internal processes spawn further tree levels in parallel

(c) Resource management system (RMS) creates tool back-end processes

(d) Tool back-end processes connect to leaves of MRNet internal process tree

Figure 3: MRNet instantiation using a resource management system. When the tool front-end instantiates MRNet, MRNet spawns its internal process tree recursively and in parallel using a remote shell utility like rsh (a, b). Once all MRNet internal processes are instantiated, the tool issues a request to the resource management system to start the tool back-ends (c). The back-ends connect to the leaves of the MRNet process tree to complete MRNet instantiation (d).

more new data packets. Transformations are synchronous, and may carry state from one transformation operation to the next. MRNet provides several built-in transformation filters:

- *Basic scalar operations*: min, max, sum and average on integers or floats; and
- *Concatenation*: input *n* scalars and output them as an *n*-element vector.

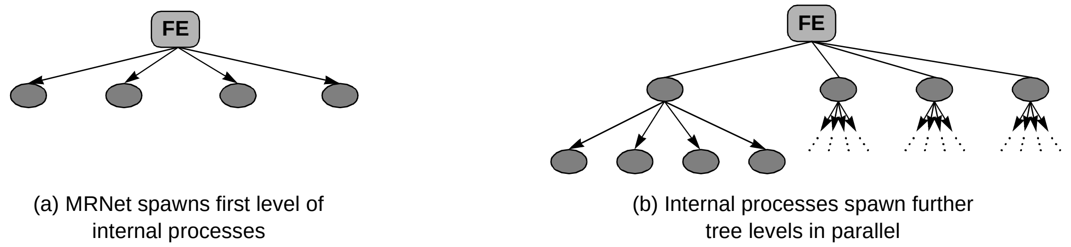
MRNet allows tool developers to add new filters to the provided set using `load_FilterFunc`, a function that takes the name of a filter function and the name of a shared object that contains the filter function and returns an identifier that can be used to bind the filter to streams. This function leverages the operating system's API for managing shared objects (e.g., `dlopen` and `dlsym` on UNIX systems).

2.4. MRNet instantiation

We currently support two modes of instantiating MRNet-based tools. In environments where a resource manager is available, MRNet cooperates with the system's resource manager to launch the tool. In simple environments, MRNet is instantiated using primitive remote shell facilities like `rsh` or `ssh`.

In the first mode of instantiation (shown in Figure 3), MRNet relies on the resident resource management system to create some or all of the MRNet processes to accommodate the cases where MRNet cannot properly instantiate a tool's back-end processes. For example, MRNet may not be able to provide the environment needed to create the processes successfully. In cases like these, MRNet creates its internal processes recursively as in the first instantiation mode, but does not instantiate any back-end processes. The tool starts the tool back-ends using the resource system. MRNet uses a shared filesystem or some other available information transfer method to provide the back-ends with the information they need to connect to the MRNet internal process tree, such as the parent processes' host names and connection port numbers.

In the second mode of instantiation (shown in Figure 4), 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. Starting with the front-end, as each *parent node* is created, it consults the configuration and uses the remote shell facility to create its children nodes (internal pro-


FE
(a) MRNet spawns first level of internal processes
FE
(b) Internal processes spawn further tree levels in parallel

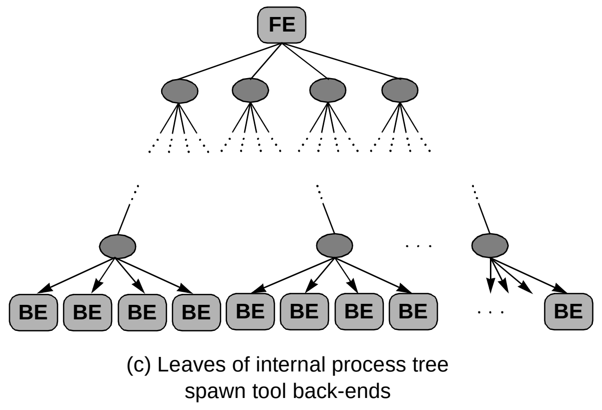


Figure 4: MRNet instantiation in environments without a resource management system. When the tool front-end instantiates MRNet, MRNet spawns its internal process tree recursively and in parallel using a remote shell utility like rsh (a, b). When all MRNet internal processes have been created, MRNet spawns the tool back-ends using the remote shell utility to complete MRNet instantiation (c).

cesses or application end-points) on the appropriate hosts. Each newly created process establishes a connection to the parent process that created it. The parent process uses this connection to send its child process the portion of the configuration relevant to each child so it may continue the instantiation of the sub-tree rooted at that child.

3. Benchmarking MRNet: lessons learned

To evaluate MRNet, we measured its performance and scalability in both a simple test tool for micro-benchmarking and when integrated into Paradyn, a real-world parallel performance tool. We evaluated MRNet on the ASCI Blue Pacific system [1] at Lawrence Livermore National Laboratory. While evaluating MRNet, we recognized several important lessons about benchmarking tools in large-scale and distributed environments.

3.1. Automation is essential

For all but the simplest experiment plans, automation is essential for completing an experiment plan successfully and on time. We exploited automation throughout our benchmarking effort including the use of LCRM [2], the batch system at LLNL, for scheduling and running our experiments, the use of shell scripts to generate the jobs comprising an entire scalability study at one time, and the use of tool scripting support and hardwired control within the tool to control individual experiment runs.

Early in our benchmarking effort, we chose to use the LCRM batch system for scheduling and launching our experiments on Blue Pacific. There were several reasons for our decision to use LCRM. First, Blue Pacific policy severely limits the number of nodes available to interactive jobs; the node limits are much more reasonable for batch jobs. Second, batch jobs can request dedicated resources, whereas interactive jobs may share their nodes with other interactive jobs. Third, the batch system greatly reduces the need for us as experimenters to schedule our experiments, find enough free resources, and launch and monitor our experiments. In fact, using the batch system often allowed us to run several smaller-scale experiments simultaneously, greatly reducing the time required to perform a scalability study. Although we could have performed this scheduling manually, we doubt we would have run many experiments concurrently because of the difficulty of scheduling and monitoring the jobs. When submitting experiment jobs to the batch system, we developed a set of parameterized shell scripts that generated and submitted jobs for an entire scalability study at once, exhibiting yet another use of automation to ease our benchmarking process.

In benchmarking MRNet, we used a crude, manual approach for managing our experiments and collecting performance data. We used `sed` and `awk` scripts to extract the data from each experiment's standard output file. We pasted that data into a spreadsheet, and used the spreadsheet charting functionality to generate our initial graphs. The next step beyond these simplistic techniques is to use an experiment management system like Electronic Notebooks [3], Zoo [5], Karavanic's experiment management system [6], and ZENTURIO [9] that automate many of the data management and experiment generation activities that we performed manually.

Using the LCRM batch system proved to be a substantial benefit for managing our experimental plan and for reducing the time required for experimentation. However, automating our experimental process with LCRM and the job submission scripts led to several significant and costly changes to our real-world tool and our approach for running experiments.

Unlike our simple test tool, Paradyn was designed to be an on-line interactive tool with a graphical user interface. Interactive experimentation using Paradyn's graphical user interface is not feasible for use in the scalability studies we used to evaluate MRNet. It is possible to run programs with an X Window System-based GUI from batch jobs on Blue Pacific with judicious use of the `xauth` command. In this case, the program's GUI is displayed once the job runs. We rejected this approach for our experimentation for two reasons. First, the delay between submitting the job and when it is actually run can vary greatly depending on the other jobs in the batch queue and the number of nodes requested for the job. Second, using interactive jobs for experimentation requires an operator be present to start, control, and monitor the experiments; this is a tedious and error-prone task. Third, the delay imposed by displaying the tool's GUI remotely was non-negligible and perturbed our benchmarking results. Therefore, we adapted Paradyn and our experimental methodology so that our experiments could be submitted and run without a user interface.

When run interactively, the user controls Paradyn by interacting with its graphical user interface. Since we removed its user interface for our experiments, we used scripting and hardwired control logic to control the tool during each experiment run. Paradyn supports minimal tool scripting functionality using Metric Definition Language [4] files, but such support is limited to setting parameter values and creating application processes at tool start-up. To control the tool once the application processes had been created, we added hardwired control logic to the tool to start the application and configure the tool to collect the performance data we needed for our benchmark tests.

Although we finally found an approach that allowed us to automate our benchmarking process, adapting our tool and experimental methodology was costly. We strongly recommend that all tool builders design their tools with automation in mind by supporting a non-interactive mode where the tool can be run without a graphical user interface, and by providing tool scripting support that exposes all of the tool's functionality to the scripting language. These recommendations are especially important for those developing *on-line* tools like Paradyn that interact with the application as it runs.

3.2. Micro-benchmarks are invaluable for design validation and debugging

Although our most important goal in benchmarking MRNet was to show its scalability and performance in the context of a real-world tool, we initially examined the behavior of important low-level MRNet functionality in a simple, controlled environment. We implemented a test harness tool that performed several micro-benchmarks. Using this tool, we measured the latency of instantiating the MRNet process network, the latency of a single broadcast/reduction pair, and the throughput while performing repeated MRNet reductions. The micro-benchmarks were invaluable not only for establishing the performance and scalability of MRNet (quantitative results were given in previous work [10]), but also for validating the MRNet design and for exposing subtle bugs in the implementation that would have been difficult to detect from the real-world tool benchmark results.

Our micro-benchmark experiments were critical for verifying the MRNet design and for finding bugs in its prototype implementation. For example, when looking at the results of the round-trip latency scalability study, we expected to find a clear exponential increase in the round-trip latency when not using MRNet, and at most a small, steady increase in the latency when using MRNet. Initially, the results roughly exhibited these trends, but did not exhibit the smooth curves that we expected as we increased the number of tool back-ends. After much investigation, we found that the MRNet micro-benchmark results were falling victim to the Nagling algorithm [11], the TCP/IP socket performance enhancement option whereby sends of small messages are delayed with the hope that another send will occur soon that can be concatenated with the original send. Once we turned off the Nagling algorithm for MRNet sockets, our micro-benchmarks confirmed the scalability of MRNet's multicast and data aggregation operations. Because of the higher level of communication complexity within Paradyn, this bug would have been much more difficult to uncover based solely on the results taken from our real-world tool experiments. Our experience with micro-benchmarks also provided strong evidence for the critical need for resiliency and reliability in distributed infrastructure; we will address MRNet resiliency and reliability in future work.

3.3. Resource management systems need improved tool support

Early in our MRNet benchmarking effort, we realized that the resource management system in our target environment had a critical need for improved tool support. Starting Paradyn, MRNet processes, and an application together within a batch job is not supported by the batch system on Blue Pacific, requiring a clumsy, ad hoc workaround to support our experimentation.

On the LLNL's IBM systems (including Blue Pacific), the batch system requests IBM's LoadLeveler to allocate a node partition for the job, and to set up the environment within that partition for the job's processes. In the typical case, the job's batch script invokes the `poe` parallel process launcher command to start an MPI application. The LoadLeveler-provided environment causes `poe` to create application processes on each node of the partition (though not necessarily one process per processor within the nodes). Unfortunately, the typical LLNL batch system/LoadLeveler/poe usage model is insufficient for our needs. For our experiments, the application processes, tool processes, and MRNet processes must be running at the same time. Because the LLNL batch system does not support co-scheduled jobs, we must run application processes, tool processes, and MRNet processes within the same node partition. However, we also wish to keep the application processes distinct from the tool processes to avoid having the tool perturb the application's behavior. In effect, we wish to divide the LoadLeveler node partition into a sub-partition for the application, a sub-partition for the tool processes, and a sub-partition for the MRNet internal processes.

Starting the application and tool in the same partition under the LLNL batch system requires several more modifications to Paradyn and a carefully-crafted job batch script. As with the micro-benchmark experiment jobs, the job's batch script must determine the names of the nodes available in the allocated partition. The node running the job script is allocated to run the tool's front-end. Next, the script allocates a subset of the available nodes for running the application and the tool daemons. (Paradyn's daemons run on the same nodes as the application processes because they communicate performance data between application process and daemon using shared memory.) The nodes that will run application processes and tool daemons is written to a `poe` "host file" for later use. The list of remaining nodes is passed to MRNet's configuration generator utility, which produces an MRNet configuration file. Finally, the job script starts the Paradyn front-end, providing it with the location of the application node file and the MRNet configuration file. The front-end launches the Paradyn daemons using `poe` and the host file written earlier. The Paradyn daemons initialize and waits to establish its connection to MRNet. The front-end then instantiates the MRNet network using the instantiation mode where MRNet creates internal processes but does not create the tool back-ends. The front-end finishes its tool start-up by issuing the request to MRNet to connect its leaves to the tool daemons. Once the daemons are connected to MRNet, Paradyn performs its MRNet-based start-up activities [10] like clock skew detection and determination of the name and location of all functions.

Lack of support for running tools with applications is not restricted to the resource management system on Blue Pacific; it is endemic within resource management systems for MPPs, clusters, and the Grid. The Tool Daemon Protocol (TDP) [8] promises to alleviate this problem. TDP defines an interface between tools and resource management systems that exposes information and control functionality from the resource manager to the tool, while allowing the resource manager to directly control the application process. For our experimentation, the ability to either (a) co-schedule jobs with LCRM or (b) sub-partition the LoadLeveler partition would have greatly simplified our experimental approach; we recommend that resource management systems provide these capabilities whenever possible to support tool experimentation and use.

3.4. Run at scale, early and often

As we neared the end of our MRNet benchmarking effort, we found that we followed a poor approach for running our large-scale experiments. When we started benchmarking, it seemed natural to experiment at the smaller scales first to expose and fix bugs in the MRNet implementation and experimental methodology. However, we found that each time we tried to increase the scale of our experiments (e.g., increasing the tool back-ends to the next power of two), we found new problems that were not evident at the smaller scale. These problems included hard-coded limits within Paradyn and the tool infrastructure and race conditions that did not manifest themselves until enough concurrency was present in the tool system. After debugging and fixing the problems at a given scale, we would inevitably have to re-run the experiments at the smaller scales because we had changed the code. In all, our approach turned out to be highly inefficient.

In retrospect, we would have been better served by adopting an approach where we ran large-scale experiments early in the benchmarking effort and frequently throughout our experimental plan. This approach maximizes the likelihood of finding the bugs that manifest themselves only at scale, leaving the most time to identify and fix them. Also, in situations where the resources required for extremely large-scale experimentation are available infrequently (e.g., jobs using more than 512 processors are not allowed to run during the daytime on Blue Pacific), this approach improves the chances of obtaining the needed resources at all.

4. Summary

MRNet is a parallel tool infrastructure that provides scalable multicast and data aggregation functionality. To evaluate MRNet, we measured its performance and scalability within a simple test tool and within Paradyn, a real-world performance tool. Using the test tool, we measured the latency and throughput of simple collective communication operations as we increased the number of tool back-ends. These experiments gave us confidence in MRNet's design and initial implementation. Using the MRNet-based Paradyn implementation, we measured the tool's start-up latency as we increased the number of tool daemons. We also measured the tool's ability to process the performance data generated by its daemons as we varied the load and increased the number of daemons.

In the process of evaluating MRNet, we learned several important lessons (or reinforced lessons we already knew) about benchmarking tools in large-scale environments:

- Automation is essential for a successful benchmarking effort, and should be leveraged throughout the benchmarking process.
- Micro-benchmarks are invaluable for verifying the design and for debugging low-level tool functionality.
- Resource management systems need significant improvement in their support for running tools and applications together. All resource management developers must recognize the importance of supporting tools and embrace approaches like the Tool Daemon Protocol. Until that happens, clumsy workarounds will continue to be necessary for using tools with resource management systems.
- Try the most demanding experiments early and often during the benchmarking effort. In benchmarking MRNet, the most demanding experiments were the large-scale experiments (i.e., those with the most tool back-ends). This approach increases the chances of detecting bugs in the tool and experimental methodology that occur only at scale, and the chances of having sufficient time to deal with such bugs.

Acknowledgments

This paper benefited from the hard work of past and present members of the Paradyn research group. We also thank Chris Chambreau, Jeff Vetter, Barbara Herron, Charlie Hargraeves, and especially John Gyllenhaal for help with the environment on ASCI Blue Pacific.

References

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

[2] Lawrence Livermore National Laboratory, "Livermore Computing Resource Management System (LCRM)", http://www.llnl.gov/computing/tutorials/lcrm/, November 25, 2003.

[3] A. Geist, J. Schwidder, D. Jung, and N. Nachtigal, "ORNL Electronic Notebook Project", http://www.csm.ornl.gov/~geist/java/applets/enote/, November 26, 2003.

[4] J.K. Hollingsworth, B.P. Miller, M.J.R. Goncalves, O. Naim, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation", *International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, San Francisco, California, November 1997, pp. 201–213.

[5] Y. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti, "Zoo: A Desktop Experiment Management Environment", *22nd International VLDB Conference*, Bombay, India, September 1996, pp. 274–285.

[6] K.L. Karavanic and B.P. Miller, "Experiment Management Support for Performance Tuning", *SC97*, San Jose, California, November 1997.

[7] 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.

[8] B.P. Miller, A. Cortes, M. A. Senar, and M. Livny, "The Tool Daemon Protocol (TDP)", *SC 2003*, Phoenix, Arizona, November 2003.

[9] R. Prodan and T. Fahringer, "A Web Service-Based Experiment Management System for the Grid", *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003, pp. 85–94.

[10] P.C. Roth, D.C. Arnold, and B.P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools", *SC 2003*, Phoenix, Arizona, November 2003.

[11] W.R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, Reading, Massachusetts, January 1994.