A Lightweight Library for Building Scalable Tools

Emily R. Jacobson, Michael J. Brim, and Barton P. Miller

Computer Sciences Department, University of Wisconsin
Madison, Wisconsin, USA
{jacobson,mjbrim,bart}@cs.wisc.edu

Abstract. MRNet is a software-based multicast reduction network for building scalable tools. Tools face communication and computation issues when used on large systems; MRNet alleviates these issues by providing multicast communication and data aggregation functionalities. Until now, the MRNet API has been entirely in C++. We present a new, lightweight library that provides a C interface for MRNet back-ends, making MRNet accessible to a wide range of new tools. Further, this library is single threaded to accommodate even more platforms and tools where this is a limitation.This new library provides the same abstractions as the C++ library, using an API that can be derived by applying a standard translation template to the C++ API.

Keywords: scalability, tree-based overlay networks, tools.

1 Introduction

As high performance computers reach processor counts in the hundreds of thousands, or even millions of cores, runtime tools are needed to support the performance profiling and debugging of applications running on these computers. Unfortunately, tools that previously worked at smaller scales do not work effectively on the larger systems. To this end, we have developed a tree-based overlay network infrastructure, called MRNet, for building tools that can scale to the largest of computing platforms. MRNet makes operations such as command and control, and data collection and reduction, efficient at large scale.

Runtime tools are often organized around two main activities: data collection, with data originating from the tool *daemons* or back-ends and traveling to the front-end, and application process control, initiated by a tool's user interface or front-end and directed to the back-end. These are the two areas in which tools pay most costs: computation and communication. Computation is in the form of data collection, aggregation, and analysis, and communication arises from the transfer of data between tool components. Tools have typically been designed with the front-end talking directly to the back-ends, causing the front-end to become a bottleneck for both communication and computation.

MRNet is designed to address many of these issues by providing broadcast and aggregation functionality [7,8]. MRNet uses a tree-based overlay network

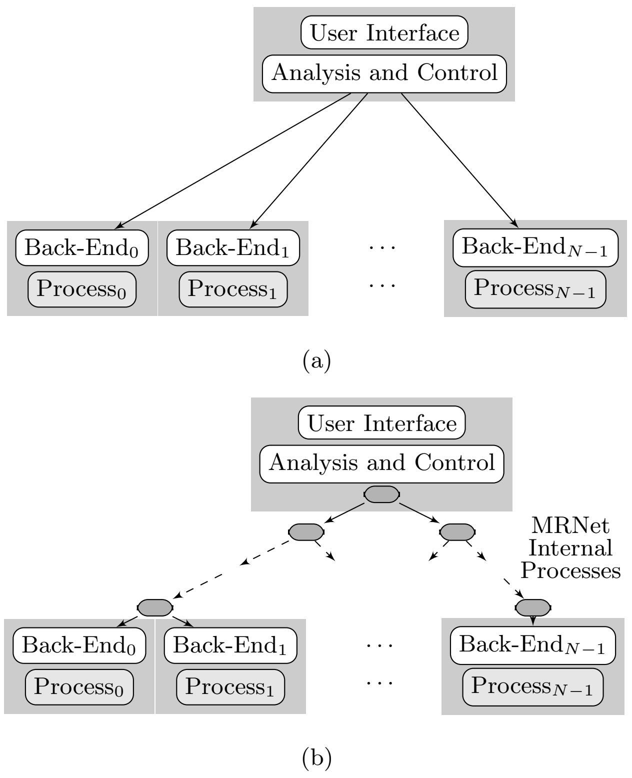

Fig. 1. The components of a typical parallel tool (a) and an MRNet-based tool (b). *Shaded boxes show potential machine boundaries.*

(TBON) of communication processes between the tool front-end and the tool back-ends, as shown in Figure 1. MRNet leverages this structure to distribute computation among internal processes and to support scalable multicast operations. Data can be filtered as it is passed up the tree; such filtering might do data transformation or might simply aggregate packets to be passed to the front-end. Data is transferred between nodes using an efficient, packed binary representation, which provides high-bandwidth communication. Further, the user may designate multiple concurrent data channels, allowing for a variety of types of data processing and aggregation to happen simultaneously. Together, these features all work to mitigate the high costs of communication and computation on large-scale systems. MRNet has been demonstrated on tools running on the largest of existing computing platforms [2,4].

MRNet has two components: `libmrnet`, a library API that is linked into a tool's front-end and back-end components, and `mrnet_commnode`, the program that runs on intermediate nodes that forms the communication processes of the tree-based network interposed between the front-end and back-ends.

The standard MRNet library used in the front- and back-ends [7] provides a C++ interface. While C++ offers many attractive software engineering features, MRNet previously was incompatible with tools written in C. In practice, few parallel applications are written using C++ and not all high-performance computing systems support general-purpose threading. In this paper, we introduce a lightweight back-end library with a pure C interface. The lightweight library is intended for use only within MRNet back-ends and offers a subset of the functionality normally provided by the C++ library to back-ends.

There are two classes of tools for which we anticipate this being useful. The first class is tools written in C; these tools need a C interface with which to interact. The second class are tools that interact with applications written in C, where such a tool might instrument the application with MRNet API calls in order to extract information; the language of the application limits the API that may be used for such purposes. The new library makes MRNet available to many tools that were previously unable to use it. In addition, this lightweight library is single threaded, to accommodate even more platforms and tools where this is a limitation. This lightweight MRNet provides the same abstractions as the C++ library; only the API is different. In most cases, however, there is a direct translation between the C++ API call and the new C version.

We use the term "lightweight" to reflect that new library is less cumbersome to integrate into existing tools as it does not require tool back-ends to use C++. For many tools, a C-based library interface is much easier to develop against, as C binding layers exist for many programming languages. As an example of tools that benefit from MRNet's new lightweight back-end library, consider two MPI application profiling tools, TAU and the CEPBA Tools. The TAU Performance System from the University of Oregon ([5],[9]) uses MRNet to aggregate performance data from parallel processes. TAU inserts a tracing library into the application and instruments the application with trace routines that use MRNet to send collected performance data for processing. A similar approach for MPI profiling is used by the CEPBA-Tools from the Barcelona Supercomputing Center([3]). Although it was previously possible to use standard MRNet in these tracing libraries, it required them to be redeveloped in C++. This introduces a dependency on the `libstdc++` library that many tool and parallel application developers feel is too heavyweight in terms of code size. The new lightweight library makes it much easier to use MRNet in tools for profiling C-based MPI applications with minimal overhead.

The remainder of this paper is organized as follows. In Section 2, we present the basic abstractions of MRNet and describe how these are expressed in the new C-based library. In Section 3, we describe our interface, and in Section 4, we provide an example of a tool that can leverage this new library.

2 Abstractions

The MRNet library, `libmrnet`, allows a tool to use an overlay network of internal processes as a communication and data aggregation substrate between the tool's front-end and back-end processes. MRNet uses a variety of abstractions to support these functions. An MRNet *end-point* represents a tool or application process. In particular, a back-end is a leaf node in the TBON. The front-end can communicate in a multicast fashion with one or more of these endpoints. MRNet uses *communicators* to represent groups of network end-points. Communicators are created and managed by the front-end; currently communication is allowed only between a tool's front-end and its back-ends.

MRNet uses logical channels called *streams* to connect the front-end to the end-points of a particular communicator. Streams can carry data *packets* downstream, from the front-end to back-ends, and upstream, from back-ends toward the front-end. Packets are sets of data elements, where types are specified using a format string similar to that used by C formatted I/O primitive `printf`. For example, a packet whose data is described by the format "%d %f %s" contains an integer, float, and character string.

Data aggregation, the process of transforming input data packets into one or more output packets, is a vital component of MRNet. MRNet uses *filters* to aggregate data packets. When a stream is created, a filter is bound to the stream that defines the aggregation operation to be performed and also the expected packet data format that will be sent on the stream. MRNet supports two types of filters: synchronization filters and transformation filters. Synchronization filters provide a mechanism to deal with asynchronous arrival of packets from child nodes; these filters do no data transformation and operate on packets in a type-independent fashion. MRNet supports three synchronization modes: *Wait For All*, *Time Out*, and *Do Not Wait*. In contrast, transformation filters combine data from multiple packets by performing an aggregation that yields one or more new packets. Several general-use transformation filters are provided, including basic scalar operations like min, max, sum, average, and concatenation operations. Additionally, MRNet allows tool developers to use custom filters. The developer simply writes one or more filter functions, compiles them into a shared library, and loads the filter library in the network. Filters use a standard function signature and can perform arbitrary computations.

The internal processes of the MRNet TBON provide the core functionalities, including the logical channels for control messages and data. Further, these processes perform the data aggregation or reduction operations. When a stream is established, an internal process creates a new *stream manager* and initializes it with a set of end-points to be associated with the stream and the filter(s) to be used on the data packets sent on the stream. Upstream data buffers must be unbatched, demultiplexed, processed, and then rebatched; downstream data is similar, though the data packets may be placed in multiple output buffers because the packet may be destined for multiple back-ends.

Although the new lightweight library provides the same abstractions as the C++ library, there are a few cases in which an abstraction is not applicable in the new library. Communicators are not present in the lightweight library because they are a handle necessary only for the front-end. Standard MRNet allows for both blocking and non-blocking receive operations; because the C-based API is not multi-threaded, only blocking receive is supported. Additionally, there are slight differences in how filters are used. In standard MRNet, filtering is done at every level of the tree, including at the back-end nodes. However, because filtering at the C-based back-end nodes adds an additional level of complexity, we have chosen initially to not filter at the back-ends.

3 Interface

To support the above abstractions, the MRNet API contains `Network`, `NetworkTopology`, `Communicator`, `Stream`, and `Packet` classes. The `Network` class is used to instantiate the TBON and access end-point objects representing tool back-ends. The `NetworkTopology` class provides an interface for discovering topology details of the instantiated `Network`. The `Communicator` class is used to represent a group of end-points when creating a `Stream` for unicast, multicast, or broadcast communication. The `Packet` class encapsulates the data packets that are sent on a `Stream`.

```
C++:                        C:
return_type                 return_type
class:function_name (       class_function_name (
    param1_type param1,         class class_object,
    ...);                       param1_type param1,
                                ...);
```

Fig. 2. API Translation Template

```
C++:                        C:
int                         int
Stream::send (              Stream_send (
    int tag,                    Stream_t * stream,
    char * fmt_string,          int tag,
    ...);                       char * fmt_string,
                                ...);
```

Fig. 3. API Translation Example

The lightweight library provides similar functionality for lightweight back-ends, so its public API is comparable to the standard MRNet API. Lightweight API classes are directly translated from the standard API. The translation scheme is shown in Figure 2 and an actual example from the Stream class is provided in Figure 3. In practice, creating a tool that uses the lightweight library will require familiarity with both the C++ and C APIs. However, because they are so similar, this should not be difficult.

Creating the MRNet overlay network is complicated by interactions with various job management systems. In the simplest environments, MRNet launches jobs using facilities like *rsh* or *ssh*. However, in more complex environments it is necessary to submit requests to a job management system. In this case, we are constrained by the operations provided by the job manager. To allow for these models, 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 components should be located.

In the second mode, MRNet relies on a process management system to create some or all of the MRNet processes [1]. This mode accommodates tools that require their back-ends to create, monitor, and control parallel application processes. MRNet creates its internal processes as in the first instantiation mode, but does not instantiate any back-end processes. When the back-ends are started by the process management system, MRNet provides the information necessary to connect the back-ends to the MRNet internal process tree. This information includes the leaf processes' host names and connection port numbers. A tool front-end can extract this information and provide it to the back-ends via the environment, using shared file systems or other information services available on the target system. The new lightweight library supports both methods of instantiation. Additionally, examples of both methods of instantiation are provided with the MRNet source code.

4 Example Tool

MRNet can be used in a wide variety of tools and application. Here, we provide a simple example to demonstrate a few key concepts. At a high level, the tool front-end sends an integer value and a number of iterations to each back-end. During each iteration, or "wave," the back-end sends the integer multiplied by the wave number back up the tree. The values are aggregated using a summation filter and passed to the front-end as a single value, which should be equal to `num_backends` × `val` × `wave_number`.

Figure 4 provides code for a custom filter used in this example. For each packet being aggregated, a value is extracted and added to the current sum (lines 10-13). Then, a new packet containing this summation is created (lines 16-18) and added to the outgoing packets (line 19).

Code for the tool front-end is shown in Figure 5. After several variable definitions in lines 2-7, an instance of the MRNet `Network` is created on line 10, using the topology specified in `topology_file`. The `Network` then loads a filter, queries for the auto-generated broadcast communicator that contains all available end-points, and then establishes a stream that will use this filter. The front-end broadcasts two integers, a value and then number of iterations to com-

```
extern "C" {
void Integer_Add(std::vector<PacketPtr> & packets_in,
                 std::vector<PacketPtr> & packets_out,
                 std::vector<PacketPtr> & packets_out_reverse,
                 void ** /* filter state */
                 TopologyLocalInfo & /* topology information */)
{
  int sum = 0, val;

  for (unsigned int i = 0; i < packets_in.size(); i++) {
    PacketPtr cur_packet = packets_in[i];
    cur_packet->unpack("%d", &val);
    sum += val;
  }

  PacketPtr new_packet (new Packet(packets_in[0]->get_StreamId(),
                                   packets_in[0]->getTag(),
                                  "%d", sum));
  packets_out.push_back(new_packet);
}
} /* extern "C" */
```

Fig. 4. MRNet filter example code

plete, on the new stream (line 27). For each iteration, the front-end performs a blocking receive (line 32); it unpacks a single integer and checks the value (lines 33-34). Finally, the front-end sends a message to the rest of the TBON to shut down the network, and then deletes the network itself (lines 38-39).

Figures 6 and 7 provides code for the back-ends that reciprocate the actions of the front-end. We provide code both for a standard back-end and a lightweight back-end. It is easy to observe that the code is nearly identical. Each tool back-end first connects to the MRNet network in line 8, using a `Network` constructor that receives its arguments using the program argument vector. While the front-end makes a stream-specific receive call, the back-end uses a stream-anonymous network receive that returns the tag sent by the front-end, the `Packet` with actual data, and a `Stream` object representing stream that the front-end has established (line 12). In both cases, this receive operation is a blocking receive; for the C++ version, this is the default mode, and for C this is the only mode supported. For each iteration, the back-end sends an integer value upstream towards the front-end (line 20).

While this example shows many of the basic concepts of MRNet, including `Network` and `Stream` creation, `Filter` loading, and `Stream` send and receive, these basic ideas allow for many other functionalities. The MRNet API Guide provides a thorough explanation of these core abstractions and their possible uses[6].

```
void front_end_main(int argc, char ** argv) {
  int send_val=32, recv_val=0;
  int tag, retval, filter_id, num_backends, num_iters;
  Network * net;
  Communicator * comm;
  Stream * stream;
  PacketPtr pkt;

  // Create a new instance of a network
  net = Network::CreateNetworkFE(topology_file,
                          backend_exe, &dummy_argv);
  filter_id =
        net->load_FilterFunc(so_file, "Integer_Add");

  // A Broadcast communication contains all the backends
  comm = net->get_BroadcastCommunicator();

  // Create a stream that will use the Integer_Add filter
  stream = net->new_Stream(comm, filter_id,
                              SFILTER_WAITFORALL);
  num_backends = comm->get_EndPoints().size();

  // Broadcast a control message to back-ends to send us "num_iters"
  // waves of integers
  tag = PROT_SUM;
  num_iters = 5;
  stream->send(tag, "%d %d", send_val, num_iters);

  // We expect "num_iters" aggregated respnoses from all back-ends
  for (unsigned int i = 0; i < num_iters; i++) {
    // Receive and unpack packet containing single int
    retval = stream->recv(&tag, pkt);
    pkt->unpack("%d", &recv_val);
    assert(recv_val == send_val*i*num_backends);
  }

  // Tell the back-ends to exit; cleanup the network
  stream->send(PROT_EXIT, "");
  delete stream;
  delete net;
}
```

Fig. 5. MRNet front-end sample code

```
void back_end_main(int argc, char ** argv) {
   Stream * stream;
   PacketPtr pkt;
   int tag = 0, retval = 0, num_iters = 0;

   // Create a new instance of a network
   Network * net = Network::CreateNetworkBE(argc, argv);

   do {
     // Anonymous stream receive
     net->recv(&tag, pkt, &stream);
     switch(tag) {
       case PROT_SUM:
         // Unpack packet with two integers
         pkt->unpack("%d %d", &recv_val, &num_iters);

         // Send num_iters waves of integers
         for (unsigned int i = 0; i < num_iters; i++) {
           stream->send(tag, "%d", recv_val*i);
         }
         break;
       case PROT_EXIT: break;
     }
   } while (tag != PROT_EXIT)

   // Wait for stream to shut down before deleting
   while(!stream->is_Closed()) sleep(1);
   delete stream;

   // Wait for the front-end to shut down the network, then delete
   net->waitfor_ShutDown();
   delete net;
 }
```

Fig. 6. MRNet standard back-end sample code

```
void back_end_main(int argc, char ** argv) {
   Stream_t * stream;
   Packet_t * pkt = (Packet_t*)malloc(sizeof(Packet_t));
   int tag = 0, retval = 0, num_iters = 0;

   // Create a new instance of a network
   Network_t * net = Network_CreateNetworkBE(argc, argv);

  do {
     // Anonymous stream receive
     Network_recv(net, &tag, pkt, &stream);
     switch(tag) {
       case PROT_SUM:
         // Unpack packet with two integers
         Packet_unpack(pkt, "%d %d", &recv_val, &num_iters);

         // Send num_iters waves of integers
         for (unsigned int i = 0; i < num_iters; i++) {
           Stream_send(stream, tag, "%d", recv_val*i);
         }
         break;
       case PROT_EXIT: break;
     }
   } while (tag != PROT_EXIT)

   // Wait for stream to shut down before deleting
   while (!Stream_is_Closed(stream)) sleep(1);
   delete_Stream_t(stream);

   free(pkt); // Cleanup malloc'd variables

   // Wait for the front-end to shut down the network, then delete
   Network_waitfor_ShutDown(net);
   delete_Network_t(net);
}
```

Fig. 7. MRNet lightweight back-end sample code

5 Conclusions

MRNet is a customizable, software-based multicast reduction network for scalable performance and system tools. MRNet reduces the cost of tool activities by leveraging a tree-based overlay network of processes between the tool's front-end and back-ends. MRNet uses this overlay network to distribute communication and computation, reducing analysis time and keeping tool front-end loads manageable. Previously, MRNet had only a C++ interface. We have presented a new lightweight library for MRNet back-ends that is single-threaded and in C. This addition makes MRNet accessible to a wide variety of tools that previously were unable to use MRNet.

6 Acknowledgements

This work is supported in part by Department of Energy grants DE-SC0004061, 08ER25842, DE-SC0003922, DE-SC0002153, DE-SC0002155, Lawrence Livermore National Lab grant B579934, and a grant from Cray Inc. through the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. D. H. Ahn, D. C. Arnold, B. R. de Supinki, G. Lee, B. P. Miller, and M. Schulz. Overcoming Scalability Challenges for Tool Daemon Launching. In *37th International Conference on Parallel Processing (ICPP-08)*, pages 578–585, Portland, OR, September 2008.
2. D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Procesing Symposium (IPDPS)*, pages 1–10, Long Beach, CA, March 2007.
3. G. Llort and J. Gonzalez and H. Servat and J. Gimenez and J. Labarta. On-line detection of large-scale parallel application's structure. In *2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, Atlanta, GA, April 2010.
4. G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons Learned at 208K: Towards Debugging Millions of Cores. In *Supercomputing 2008 (SC2008)*, Austin, TX, November 2008.
5. A. Nataraj, A. D. Malony, A. Morris, D. C. Arnold, and B. P. Miller. TAUoverMRNet (ToM): A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet. In *International Workshop on Scalable Tools for High-End Computing (STHEC 2008)*, Island of Kos, Greece, June 2008.
6. Paradyn Project. *The Multicast/Reduction Network: A User's Guide.* University of Wisconsin-Madison. `http://www.paradyn.org/mrnet/release_3.0`.
7. P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Supercomputing 2003 (SC2003)*, Phoenix, AZ, November 2003.
8. P. C. Roth, D. C. Arnold, and B. P. Miller. Benchmarking the MRNet Distributed Tool Infrastructure: Lessons Learned. In *2004 High-Performance Grid Computing Workshop, held in conjunction with the 2004 International Parallel and Distributed Procesing Symposium (IPDPS)*, Santa Fe, NM, April 2004.
9. S. Shende and A.D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications, SAGE Publications*, 20(2):287–331, 2006.