MRNet: A Scalable Infrastructure for Development of Parallel Tools and Applications

Michael J. Brim, University of Wisconsin
Luiz DeRose, Cray Inc.
Barton P. Miller, University of Wisconsin
Ramya Olichandran, University of Wisconsin
Philip C. Roth, Oak Ridge National Laboratory

ABSTRACT: MRNet is a customizable, high-throughput communication software system for parallel tools and applications. It reduces the cost of these tools' activities by incorporating a tree-based overlay network (TBON) of processes between the tool's front-end and back-ends. MRNet was recently ported and released for Cray XT systems. In this paper we describe the main features that make MRNet well-suited as a general facility for building scalable parallel tools. We present our experiences with MRNet and examples of its use.

KEYWORDS: XT, scalability, tree-based overlay networks, tools

1 Introduction

The desire to solve large-scale science problems in areas of national and global significance, including climate modeling, computational biology, and particle simulation, has driven the development of increasingly large parallel computing resources. Unfortunately, performance, debugging, and system administration tools that work well in small-scale environments often fail to scale as systems and applications get larger. In response to these deficiencies, we have developed a tree-based overlay network infrastructure, MRNet [11], for building tools that can scale to the largest of computing platforms, including current extreme-scale Cray XT systems that contain tens to hundreds of thousands of processors. MRNet makes operations such as command and control, and data collection and reduction, efficient at large scale.

Typically, tools are organized using the structure shown in Figure 1a, where a single tool front-end interacts with a large set of tool back-ends (often called *tool daemons*). This structure is commonly referred to as a *master-slave* architecture. Tool back-ends are responsible for data collection and application control, when applicable. The tool front-end often provides the interface to users, and is responsible for analysis of data collected at the back-ends. For tools using this structure, the front-end quickly becomes a bottleneck due to centralized computation and communication with all back-ends. MRNet provides a scalable solution for these tools by interposing a tree-based overlay network (TBON) of processes between the tool front-end and back-ends, as shown in Figure 1b.

The TBON is used to distribute tool activities normally performed by the front-end across the overlay processes, thus reducing analysis time and keeping the front-end load manageable. MRNet takes advantage of the logarithmic performance properties of trees to provide scalable multicast communication and data aggregation. Tools built using MRNet send and receive data between front-end and back-ends on logical data flows called *streams*. Data flowing on streams is encapsulated as *packets*, which are synchronized and aggregated using built-in or user-defined *filters*. MRNet's general-purpose abstractions allow tools to completely control how communication and computation is performed. Furthermore, MRNet lets tools define the TBON topology and the placement of processes on distributed hosts. MRNet supports any tree topology, and provides a utility for easily generating common topology structures such as balanced and k-nomial trees.

MRNet has been integrated into several existing tools, and has been used as the basis for many new tools. Performance monitoring and tracing tools such as Paradyn [7], Open|SpeedShop [9], TAU [8], and an online clustering analysis tool [5] have all benefited from integrating MRNet for improved scalability. MRNet has also served as the scalable substrate for new tools. STAT [1,4] and Cray ATP are new tools for lightweight debugging of parallel applications. TBON-FS is a system built using MRNet that provides scalable group operations on distributed files to several new and existing tools [2].

This paper is intended to guide the development and use of MRNet within tools and applications targeted for the Cray XT systems. Section 2 briefly reviews the important MRNet abstractions and interfaces. In Section 3, we provide examples of several use cases, including simple master-worker parallel applications and first- and third-party tools for debugging or performance monitoring. In each case, we highlight the API methods that may be useful and give example code. Section 4 provides instructions for building and running MRNet-based tools and applications on XT platforms, and suggestions for generating appropriate topologies.

2 MRNet Abstractions and API

MRNet has two main components: a library API that is linked into a tool's front-end and back-end components, and a `mrnet_commnode` program that runs on nodes interposed between the front-end and back-ends. The library API enables interaction between the front-end and groups of back-ends. The `mrnet_commnode` program is used to distribute data processing across many hosts and implements efficient and

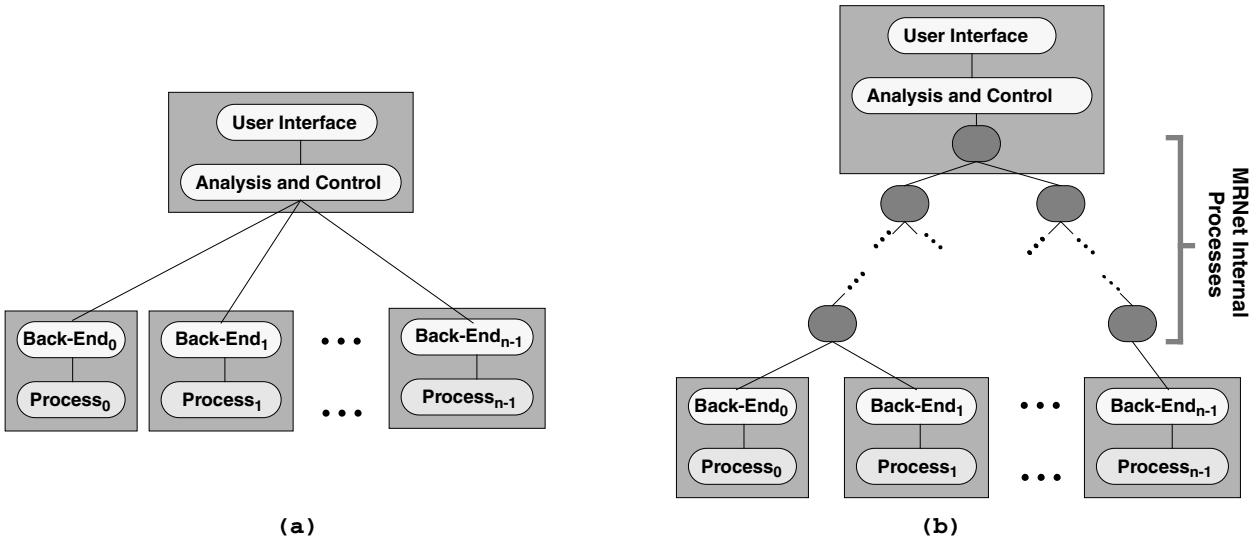

Figure 1. The components of a typical parallel tool (a) and an MRNet-based parallel tool (b). *Shaded boxes show potential machine boundaries.*

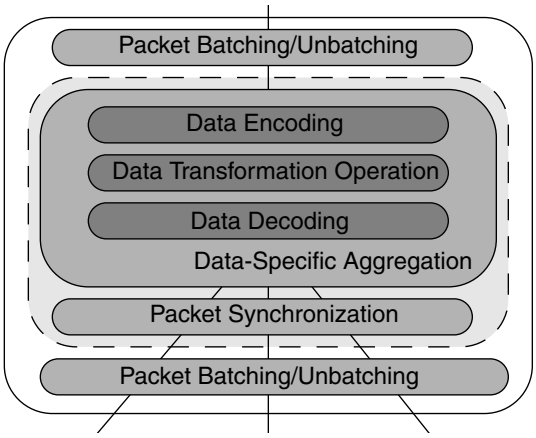

Figure 2. Functional layers in an MRNet internal process.

scalable group communications. We briefly review the MRNet abstractions, and then discuss the interfaces to the API classes corresponding to the abstractions.

2.1 MRNet Abstractions

MRNet 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_commnode` program. The connection topology and host assignment of these processes is determined by a configuration file, thus 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 4.3 for further discussion on MRNet process topologies.

MRNet uses *communicators* to represent groups of network end-points. Like communicators in MPI [6], 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.

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 (e.g., "`%ad`" for an integer array).

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. Transformation filters operate on input data packets flowing

either upstream or downstream, yielding one or more output packets. Figure 2 shows the actions taken on packets by an MRNet internal process during an upstream data flow. Packets must be unbatched, demultiplexed onto streams, synchronized, perhaps aggregated, and re-batched before continuing their upstream journey toward the front-end.

2.2 MRNet API

MRNet has five top-level classes: `Network`, `NetworkTopology`, `Communicator`, `Stream`, and `Packet`. The `Network` class primarily contains methods for instantiating and destroying MRNet process trees. The `NetworkTopology` class represents the interface for discovering details about the topology of an instantiated network. Application back-ends are referred to as end-points, and the `Communicator` class is used to reference a group of end-points. A `Communicator` is used to establish a `Stream` for unicast, multicast, or broadcast communications via the MRNet infrastructure. The `Packet` class encapsulates the data packets that are sent on a stream.

Below, we introduce the most commonly used interfaces for each class. A user's guide detailing the complete API is available online [10]. All classes are included in the `MRN` namespace. For this discussion, we do not explicitly include reference to the namespace; for example, when we reference the class `Network`, we are implying the class `MRN::Network`.

2.2.1 Class Network

The first action taken by any MRNet front-end or back-end is initialization of the network. `Network::CreateNetworkFE` and `Network::CreateNetworkBE` are the methods used for this purpose.

```
void Network::CreateNetworkFE(topology,
                              backend_exe,
                              backend_argv,
                              attrs,
                              rank_backends);
void Network::CreateNetworkBE(argc, argv);
```

`Network::CreateNetworkFE` is used by a front-end to instantiate the MRNet process tree. `topology` is the path to a configuration file that describes the desired TBON topology. `backend_exe` is the path to the executable to be used for the application's back-end processes. `backend_argv` is a null terminated list of arguments to pass to the back-end application upon creation. If `backend_exe` is NULL, no back-end processes will be started, and the leaves of the topology specified by topology will be instances of `mrnet_commnode`. `attrs` is used only when internal processes of the MRNet tree are to be co-located with the application processes. In this case, `attrs` refers to a `map` data structure that associates the `string` "apid" to a unique identifier for a set of application processes started by the Cray `aprun` command. `rank_backends` indicates whether the back-end process ranks should begin at 0, similar to MPI rank numbering.

`Network::CreateNetworkBE` is used by a back-end program to connect to a network instantiated by a front-end. When the front-end uses MRNet to start the back-end processes, information necessary for the connection is appended to the program argument vector (i.e., `argc/argv`). If a separate mechanism is used to start the back-end processes, the back-end program must construct an argument array that contains five pieces of information: the parent process' hostname, port, and rank, and the back-end's hostname and rank.

Once the front-end has initialized the network, it can load custom transformation or synchronization filters using:

```
int Network::load_FilterFunc(so_file, func,
                             is_trans_filter);
```

This method is conveniently similar to the conventional `dlopen` facilities for opening a shared object and dynamically loading symbols defined within. `so_file` is the path to a shared object file that contains the filter function to be loaded and `func` is the name of the function to be loaded. The last parameter `is_trans_filter` defaults to true and can usually be omitted since the common case is to load transformation, not synchronization, filters. On success, this method returns a unique identifier for the filter that may be used in subsequent calls to `Network::new_Stream`.

`Network::recv` is used to invoke a stream-anonymous receive operation, where any packet available (addressed to any stream) will be returned via the output parameters.

```
int Network::recv(otag, opacket,
                  ostream, blocking);
```

`otag` is set to the tag value that was passed to `Stream::send`. `opacket` is set to point at the packet that was received. A pointer to the stream to which the packet was addressed will be returned in `ostream`. `blocking` specifies whether the call should block or return if data is not immediately available; it defaults to a blocking call. A return value of 0 indicates no packets were available, and 1 indicates success.

2.2.2 Class Communicator

Communicators are used to denote the set of end-points that are accessed within a stream. Instances of `Communicator` are network specific, so their creation methods are functions of a `Network` object. The three most common operations are creating a new communicator, adding end-points to a communicator, or getting a handle to the broadcast communicator representing all back-ends.

```
Communicator * Network::new_Communicator();
```

This method creates a new `Communicator` object. The object initially contains no end-points. Use `Communicator::add_EndPoint` to populate the communicator.

```
bool Communicator::add_EndPoint(rank);
```

This method adds the end-point with corresponding `rank` to the set contained by the communicator. This method returns true on success, false when the given `rank` is invalid.

```
Communicator*
Network::get_BroadcastCommunicator();
```

This method returns a pointer to a the network's broadcast communicator, which contains all the end-points available in

the network at the time the function is called. If the network's topology changes, as can occur when starting back-ends separately, the object will be updated to reflect the additions or deletions. The returned object pointer should not be deleted.

2.2.3 Class Stream

`Stream` objects are created within the context of a specific network, so their creation methods are functions of an instantiated `Network` object. The most common approach to creating a new stream uses:

```
Stream* Network::new_Stream(comm,
                            up_trans_id,
                            up_sync_id,
                            down_trans_id);
```

This method creates a `Stream` object attached to the end-points specified by the `Communicator` object `comm`. `up_trans_id` specifies the transformation filter to apply to data flowing upstream from the application back-ends toward the front-end. `up_sync_id` specifies the synchronization filter to apply to upstream packets. `down_trans_id` allows the user to specify a filter to apply to downstream data flows.

The following methods provide basic information about created streams:

```
unsigned Stream::get_Id();
```

This method returns the integer identifier for this stream. A `Stream` object can be acquired from the network by passing this identifier to `Network::get_Stream`.

```
const std::set<Rank>&
Stream::get_EndPoints();
```

This method returns the set of end-point ranks for this stream.

```
unsigned Stream::size();
```

This method returns an integer indicating the number of end-points for this stream.

Streams provide methods for sending packets, and blocking and non-blocking methods for receiving packets. The most common method for sending data is:

```
int Stream::send(tag, format_string, ...);
```

This method creates a `Packet` from the operands and sends it on the stream, returning 0 on success or -1 on failure. `tag` is an integer identifier that classifies the data in the packet to be transmitted. The tag must have a value greather than or equal to the constant `FirstApplicationTag` defined by MRNet, as values less than `FirstApplicationTag` are reserved for internal use. `format_string` is a character string describing the `varargs` data. After calling `Stream::send` one or more times, users can force all packets currently buffered by the stream to passed to the operating system for network transmission using `Stream::flush`.

To receive data available on a particular stream, one can use:

```
int Stream::recv(otag, opacket, blocking);
```

This method returns the next packet addressed to the calling stream via the output parameters. `otag` will be set to the tag value passed to `Stream::send` operation, and `opacket` is set to point to the recieved packet. `blocking` determines whether the receive should block or return if data is not immediately available; it defaults to a blocking call. A return value of 0 indicates no packets were available, and 1 indicates success.

Users can customize the behavior of filters associated with the current stream using:

```
int Stream::set_FilterParameters(ftype,
                                 format_str,
                                 ...);
```

This method allows users to dynamically configure the operation of a filter by passing arbitrary data in a similar fashion to `Stream::send`. When the filter executes, the data is available within a packet parameter, and the filter can extract the configuration settings. `ftype` should be given as `FILTER_UPSTREAM_SYNC` to configure the synchronization filter, `FILTER_UPSTREAM_TRANS` for the upstream transformation filter, and `FILTER_DOWNSTREAM_TRANS` for the downstream transformation filter.

2.2.4 Class Packet

A `Packet` encapsulates formatted data sent on a stream. Packets are created by specifying a format string and a varargs list of data elements. For example, the format string "`%s %d`" describes two data elements, a null-terminated character string and a 32-bit integer. MRNet front-end and back-end processes do not create instances of `Packet` directly; instead they are automatically produced from the formatted data passed to `Stream::send`. MRNet supports format strings specifying basic data types including signed and unsigned characters, 16-, 32-, and 64-bit signed and unsigned integers, floats and doubles, and character strings. Additionally, format strings can specify arrays of basic types (e.g., "%ac %ad" specifies two arrays, one of characters and one of 32-bit integers).

When receiving a packet via `Stream::recv` or `Network::recv`, the Packet instance is stored within a `PacketPtr` object. `PacketPtr` is a class based on the Boost library `shared_ptr` class, and helps with memory management of packets. A `PacketPtr` can be assumed to be equivalent to "`Packet*`", and all operations on packets require use of `PacketPtr`.

The following three methods extract basic information contained within packets:

```
int Packet::get_Tag();
unsigned short Packet::get_StreamId();
const char* Packet::get_FormatString();
```

These methods return the integer tag, stream identifier, and format string of the packet.

After receiving a packet, the data contained within can be extracted using:

```
void Packet::unpack(format_string, ...);
```

This method extracts the data according to `format_string`, which must match that of the packet. The `varargs` following `format_string` should be pointers to the appropriate types of each data item. For string and array data types, new memory buffers to hold the data will be allo-

cated using `malloc`, and it is the user's responsibility to `free` these strings and arrays.

Within filter code, the following method should be used to tell MRNet to deallocate strings and arrays after sending the packet, since the user cannot free the data elements before the filter function returns.

```
void Packet::set_DestroyData(flag);
```

If `flag` is true, string and array data members will be deallocated using `free` when the packet destructor is executed. Note this assumes they were allocated using `malloc`.

2.2.5 Class NetworkTopology

A `NetworkTopology` is specific to an instantiated network. API users should not need to create `NetworkTopology` instances. Rather, the following method can be used to retrieve the topology from a `Network` object:

```
NetworkTopology*
Network::get_NetworkTopology();
```

Using the returned object pointer, various information about the topology can be retrieved.

```
unsigned NetworkTopology::get_NumNodes();
```

This method returns the total number of nodes in the tree topology, including front-end, internal, and back-end processes.

```
NetworkTopology::Node*
NetworkTopology::find_Node(rank);
```

A sub-class, `NetworkTopology::Node` is used to represent nodes in the topology. This method returns a pointer to the tree node with rank equal to `rank`, or NULL if no such node is found.

```
NetworkTopology::Node*
NetworkTopology::get_Root();
```

This method returns a pointer to the root node of the tree, or NULL if the topology is empty.

```
void NetworkTopology::get_Leaves(leaves);
```

This method fills `leaves`, which is a `set` of `NetworkTopology::Node` pointers, with the leaf nodes in the topology. In the case where back-end processes are not started when the network is instantiated, a front-end process can use this function to retrieve information about the leaf internal processes to which the back-ends should attach.

The following methods retrieve specific information for a particular `NetworkTopology::Node`.

```
string NetworkTopology::Node::get_HostName();
Port NetworkTopology::Node::get_Port();
Rank NetworkTopology::Node::get_Rank();
Rank NetworkTopology::Node::get_Parent();
const std::set<NetworkTopology::Node*>&
NetworkTopology::Node::get_Children();
```

The first three methods provide the node's hostname, port, and rank. The last two methods are useful for programmed navigation of the topology.

3 Example Tool Scenarios

MRNet is a general-purpose communication and computation infrastructure for scalable tools and applications. In this section, we provide examples for three MRNet use cases on XT systems. First, we demonstrate the common code organization for all MRNet-based software using a simple master-worker style application. Next, we discuss and provide example code for *third-party* tools that run alongside a parallel application and *first-party* tools that are embedded within a parallel application.

3.1 A Simple Example: Master-Worker Application

Figure 3 shows example code for a typical MRNet front-end, and Figure 4 shows the corresponding back-end code. All MRNet-based programs start by initializing the network. Front-end programs use `Network::CreateNetworkFE` to start the TBON internal processes, and optionally the back-end processes, using the TBON topology supplied as the first operand. In this example, we assume MRNet is used to start the back-ends, so the path to the executable and its program arguments are also given as operands. MRNet uses the ALPS `aprun` command internally to launch the back-ends. Each back-end process calls `Network::CreateNetworkBE` and passes its program arguments, which MRNet has modified to include information that allows the back-end to connect to the instantiated network.

After network initialization, the front-end can load custom filters, define communicators representing sets of back-ends, and create new data streams with user-specified filtering behavior. Once created, the front-end sends data to the back-ends on these streams, and waits for aggregated responses.

Back-ends often enter into a progress loop that calls `Network::recv` to block waiting for data to arrive on any stream. When a packet arrives, the back-end uses the packet's tag to determine the action to take. When the actions produce results, the back-end sends these results back to the front-end using the stream on which work arrived.

In this example, the front-end and back-end use three message tags: DO_WORK, UPDATE_CONFIG, and EXIT. DO_WORK is used to send new data for processing on the back-ends. UPDATE_CONFIG is a control message that changes how data is processed on the back-ends. The EXIT tag tells back-ends that work has completed.

Once the front-end has informed the back-ends that all work has been completed, it deletes its `Network` object, which causes the MRNet TBON to be shut down. By default, network teardown terminates all internal and back-end processes. A front-end can tell MRNet to leave back-ends running using `Network::set_TerminateBackendsOnShutdown`.

3.2 Third-party Tools

There are generally two classes of third-party tools: *online* tools that are active during the execution of the application, and *offline* tools that perform work only when the application has finished or is idle. The class of the tool often guides the organization of the MRNet topology to be used, as online tools may

```
main( argc, argv )
{
  Network *net;
  Stream *ctl, *work;
  Communicator *comm;

  // initialize network
  const char* topol = argv[1];
  const char* be_exe = argv[2];
  char* be_args[] = argv+3;
  net = Network::CreateNetworkFE(topol,
                                 be_exe,
                                 be_args,
                                 NULL);

  // load work processing filter
  const char* lib = "mylib.so";
  const char* func = "process_work";
  int flt = net->load_FilterFunc(lib,func);

  // create streams
  comm = net->get_BroadcastCommunicator();
  ctl = net->new_Stream(comm, TFILTER_SUM,
                        SFILTER_WAITFORALL);
  work = net->new_Stream(comm, flt,
                         SFILTER_WAITFORALL);

  // set initial work parameters
  int param = get_initial_config();
  ctl->send( UPDATE_CONFIG, "%d", param );

  // work processing
  char* input;
  unsigned input_len;
  get_work_input( input, input_len );
  while( input != NULL ) {
    work->send( DO_WORK, "%ac",
                input, input_len );
    work->recv( &tag, pkt );
    save_work_output( pkt );
    get_work_input( input, input_len );
  }

  ctl->send( EXIT, NULL );
  delete ctl;
  delete work;
  delete net;
}
```

Figure 3. Simple Master-Worker Front-end Code

want to execute MRNet internal processes on separate nodes from those running the application processes as a means to reduce or eliminate interference, while offline tools may choose to run all MRNet processes only on the nodes hosting the application as a means to reducing the number of allocated nodes.

Both classes of third-party tools need to execute at least their back-end processes on the same nodes as the application, to enable monitoring or control of the application. On Cray XT systems, the ability to co-locate tool back-ends with application processes is provided by the ALPS tool helper library. MRNet depends on this library to start back-end processes on the

```
main( argc, argv )
{
  Network *net;
  Stream *strm;
  int config = 0;

  // initialize network
  net = Network::CreateNetworkBE(argc,argv);

  // work loop
  bool done = false;
  while( ! done ) {
    int tag;
    PacketPtr pkt;
    int rc = net->recv(&tag, pkt, &strm);
    switch( tag ) {
     case DO_WORK:
       be_do_work( config, pkt, strm );
       break;
     case UPDATE_CONFIG:
       pkt->unpack("%d", &config);
       break;
     case EXIT:
       done = true;
       break;
    }
  }
  delete net;
}
```

Figure 4. Simple Master-Worker Back-end Code

```
main( argc, argv )
{
  Network *net;

  // initialize network
  const char* topol = argv[1];
  const char* apid = argv[2]
  const char* tool_be_exe = argv[3];
  char* tool_be_args[] = argv+4;
  map<string,string> attrs;
  attrs.insert( make_pair("apid",apid) );
  net = Network::CreateNetworkFE(topol,
                                   tool_be_exe,
                                 tool_be_args,
                                   &attrs);
  ...
}
```

Figure 5. Third-party Tool Front-end Code

application nodes [12]. During network instantiation, the tool front-end must pass the `aprun` identifier (apid) for the application in the attribute map operand to `Network::CreateNetworkFE`. Figure 5 shows the additional code necessary for providing the apid.

3.3 First-party Tools

First-party tools are typically implemented as libraries that are linked into applications. These libraries contain tool back-end functionality for extracting runtime performance or tracing data from the application, and shipping the data to external

```
// global configuration parameter
int config;

Network_t* toollib_init_mrnet(void)
{
  config = 0;

  // construct argc/argv from topology
  // info available from external channel
  int argc;
  char **argv;
  get_network_connection( &argc, &argv );

  // initialize network
  Network_t *net;
  net = Network_CreateNetworkBE(argc,argv);
  return net;
}

bool toollib_progress_mrnet(Network_t* net)
{
  int tag;
  Stream_t *strm;
  Packet_t *pkt;
  int rc = Network_recv(net, &tag,
                        pkt, &strm);
  switch( tag ) {
   case DO_WORK:
     be_do_work(config, pkt, strm);
     break;
   case UPDATE_CONFIG:
     Packet_unpack(pkt, "%d", &config);
     break;
   case EXIT:
     done = true;
     break;
  }
  if( done ) {
    free(net);
    return false;
  }
  return true;
}
```

Figure 6. Lightweight First-party Back-end Code

processes for analysis. MRNet can be used within these libraries to send data for analysis within the TBON, and to receive control messages from the tool front-end that adapt the tools data collection behavior.

If the application is already programmed using C++ and supports threading, the standard MRNet API library can be used within the tool library to send data. Practically, however, few parallel applications are written using C++ and not all high-performance computing systems support general-purpose threading. In response to these facts, we have developed a lightweight, C-based and threadless version of the MRNet library for use with these applications and systems 3. The C library API uses a simple strategy to provide C function interfaces for C++ class methods by translating interfaces from `Class::method(...)` to `Class_method(Class*, ...)`. 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. Notably, only blocking network receives are supported and no filtering is performed within the back-end process. Figure 6 shows how the simple back-end code of Figure 4 could be implemented within a tool back-end library. The code is split into initialization and progress functions that the tool would be responsible for calling, and the API calls are translated to use the lightweight MRNet library.

4 Running MRNet Tools on Cray XT

We now provide specific instructions for building and running MRNet-based software on the Cray XT. Due to potential differences in system configuration, these instructions should be considered as a template rather than strict convention.

4.1 Building MRNet Applications and Tools

MRNet currently requires use of the GNU programming environment on Cray XT systems, due to problems encountered with C++ STL support when using the PGI environment. To use the GNU environment, a user should swap the environment from the default PGI:

```
module swap PrgEnv-pgi PrgEnv-gnu
```

Using the GNU environment, MRNet can be configured and built using the following commands (assuming a Bash or Korn shell):

```
cd mrnet_source_directory
export CC=gcc
export CXX=g++
./configure --prefix=/tmp/work/user
            --with-libfldir=/usr/lib64
            --with-startup=cray_xt
           [--with-alpstoolhelp-inc=dir]
           [--with-alpstoolhelp-lib=dir]
           [--enable-shared]
make && make install
```

The configure command above uses the `--prefix` switch to tell MRNet to install its executables, header files, and libraries into `bin`, `include`, and `lib` subdirectories of `/tmp/work/user`, which we assume is located on a Lustre file system. Installation to Lustre is necessary because a user's home directory is not accessible when jobs are run on XT compute nodes. The optional configure switches for the location of ALPS tool helper header files and libraries are necessary when building MRNet to be used by a tool that needs to co-locate with application processes. The optional switch `--enable-shared` can be specified to instruct MRNet to build shared object versions of the MRNet and XPlat libraries in addition to the regular library archives.

Once MRNet has been built and installed, software that uses MRNet can be built by supplying the appropriate compile flags that point to the installed header and library locations. Note that programs using MRNet must also be built within the GNU programming environment to avoid link-time problems with unresolved references. The compiler settings used to build custom MRNet filters can be found by referring to the Makefiles

MRNet generates for its examples (e.g., the IntegerAddition example uses a custom filter).

4.2 Running MRNet Applications and Tools

Because MRNet is dependent upon shared object libraries from the GNU environment, and these libraries are not installed on the compute nodes, the dependencies must be copied to the Lustre file system, preferably into the same directory used for the MRNet libraries. The exact dependencies may differ across systems, so we suggest using '`ldd` *`executable`*' and '`ldd` *`library`*' to determine the dependencies for executables and libraries. Generally, we find it is always necessary to copy the following dependencies: `libstdc++.so`, `libgcc_s.so`, and `libm.so`.

To ensure installed executables and libraries are found within the system search paths, we suggest adding the following lines to the user's login scripts, assuming the installation directories are in `/tmp/work/`*`user`*. For bash/ksh users, add the following two environment settings to `~/.bashrc`:

```
 export PATH="/tmp/work/user/bin:${PATH}"
 export LD_LIBRARY_PATH="/tmp/work/user/
lib:${LD_LIBRARY_PATH}"
```

while for csh/tcsh users, the following settings should be added to `~/.cshrc`:

```
 setenv PATH "/tmp/work/user/bin:${PATH}"
 setenv LD_LIBRARY_PATH "/tmp/work/user/
lib:${LD_LIBRARY_PATH}"
```

Once all the software has been installed and the user's environment has been setup, the next step is to create job scripts for running the application or tool. Figure 7 shows an example PBS job script that can be used to run the MRNet `microbench` test, which measures the performance of the network for round-trip latency and aggregation throughput. This example script assumes the execution platform is an XT5 with 12 cores per compute node. The topology used in the example is `1x24x576`, which means the front-end has 24 internal processes as its children, and each internal process has 24 back-end processes as children, for a total of 576 back-ends. The topology file is generated from a host list that is populated with the node identifiers (or "nids") of the front-end and all allocated compute nodes.

4.3 MRNet Topology Generation

MRNet provides a utility, `mrnet_topgen`, for generating TBON topology files from an input file that contains a list of hosts. Each host line in the input file is of the form "`host[:num-processors]`", which tells MRNet to place at most `num-processors` processes on `host`. In the example job script of Figure 7, all internal processes are placed together on two hosts as a result of specifying "`:12`" as the processor count for all compute nodes. Such overloading of internal processes onto the same host is not always desirable, since the `mrnet_commnode` process is multi-threaded and has two threads for every peer in the topology. For topologies with large fan-outs (e.g., 24 or 32), it can be beneficial to provide multiple processor cores to each `mrnet_commnode` process. Our experience has shown that running two or four internal processes on a 12-core XT5 compute node provides reasonable performance for a balanced network with a large fan-out. For workloads where the internal process is expected to be constantly communicating and/or performing heavy computation due to filtering, users may wish to place a single internal process on a compute node by specifying "`:1`" for the hosts to be used for internal processes, and adjusting the size of the job allocation to account for the increased processor count used by each internal process.

```
#!/bin/ksh
#PBS -N microbench
#PBS -j oe
#PBS -l size=600
#PBS -l walltime=0:10:00

echo ------- Environment -------
env
echo ---------------------------

echo ------- Job Info ----------
qstat -f $PBS_JOBID
echo ---------------------------

n_procs=600
cores_per_node=12
n_compute_nodes=$(expr $n_procs / $cores_per_node)
fanout=24
depth=2

hostfile=hosts.txt
topofile=mrnet.top


jobdir=${HOME}/log/microbench/$PBS_JOBID
logdir=${jobdir}/mrnlog
mkdir -p $jobdir || exit 1
mkdir $logdir || exit 1
cd $jobdir || exit 1
export LUSTREBIN=/tmp/work/user/bin
export MRNET_OUTPUT_LEVEL=1
export MRNET_DEBUG_LOG_DIRECTORY="${logdir}"
export XPLAT_RESOLVE_HOSTS=0

# build a list of nodes allocated to us
# (be sure to include the FE node)
cat /proc/cray_xt/nid | awk '{printf("nid%05u\n",$1)}' \
    > $hostfile
aprun -n $n_compute_nodes -N 1 /bin/hostname | sort \
    | tail -n +2 | awk '{printf("%s:12\n", $1)}' \
    >> $hostfile

# build a topology
${LUSTREBIN}/mrnet_topgen -b ${fanout}^${depth} \
    $hostfile $topofile > topgen.log 2>&1

# run the test
${LUSTREBIN}/microbench_FE 1000 1000 $topofile \
    ${LUSTREBIN}/microbench_BE > 1x24x576.log 2>&1
```

Figure 7. Sample PBS Job Script: microbench Application

5 Summary

MRNet provides scalable multicast and data aggregation facilities to tools and applications. Our goal is to encourage more tool and application developers to consider adopting MRNet as a solution for improving the scalability of their software. We have reviewed the general-purpose abstractions and API provided by MRNet, and given several examples of tools already using MRNet. We have shown example code for using MRNet on Cray XT platforms, and provided specific instructions for building and running MRNet-based software.

Acknowledgments

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

This material is based upon work supported by 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.

About the Authors

Michael J. Brim is a Ph.D. student in the Paradyn research group at the University of Wisconsin, Madison. His research interests include scalable tool and middleware development, high-end computing and communication architectures, and distributed and parallel file systems. He can be contacted at Department of Computer Sciences, 1210 W. Dayton St., Madison, WI 53706, USA. Email: mjbrim@cs.wisc.edu.

Dr. Luiz DeRose is a Sr. Principal Engineer and the Programming Environments Director at Cray Inc. He has more than twenty years of experience in HPC software design and development. He has published more than 40 peer-review articles in scientific publications, primarily on programming environment topics. He can be reached at ldr@cray.com.

Barton Miller is Professor of Computer Sciences at the University of Wisconsin, Madison. He directs the Paradyn Parallel Performance Tools project, which is investigating performance and instrumentation technologies for parallel and distributed applications and systems. His research interests include tools for high-performance computing systems, binary code analysis and instrumentation, computer security, and scalable distributed systems. He can be reached at Department of Computer Sciences, 1210 W. Dayton St., Madison, WI 53706, USA. Email: bart@cs.wisc.edu.

Ramya Olichandran is a graduate student in the Paradyn research group at the University of Wisconsin, Madison. Her research spans tool scalability and performance analysis. She can be reached at Department of Computer Sciences, 1210 W. Dayton St., Madison, WI 53706, USA. Email: ramya@cs.wisc.edu.

Philip C. Roth is a computer scientist in the Computer Science and Mathematics Division at Oak Ridge National Laboratory, where he is a founding member of the Future Technologies Group. His research interests include performance analysis, prediction, and tools with special emphases on scalability, automation, and non-traditional architectures. He earned his Ph.D. from the University of Wisconsin-Madison in 2005. Roth can be reached at Oak Ridge National Laboratory, PO Box 2008 MS 6173, Oak Ridge, TN 37830-6173, USA, Email: rothpc@ornl.gov.

References

[1] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory Lee, Barton P. Miller, and Martin Schulz, "Stack Trace Analysis for Large Scale Applications", *International Parallel & Distributed Processing Symposium (IPDPS 2007)*, Long Beach, California, March 2007.

[2] Michael J. Brim and Barton P. Miller, "Group File Operations for Scalable Tools and Middleware", *16th International Conference on High Performance Computing (HiPC)*, Kochi, India, December 2009.

[3] Emily R. Jacobson, Michael J. Brim, and Barton P. Miller, "A Lightweight Library for Building Scalable Tools", *State of the Art in Scientific and Parallel Computing (PARA 2010)*, Reykjavik, Iceland, June 2010.

[4] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit, "Lessons Learned at 208K: Towards Debugging Millions of Cores", *Supercomputing 2008 (SC'08)*, Austin, TX, November 2008.

[5] German Llort, Juan Gonzalez, Harald Servat, Judit Gimenez, and Jesus Lebarta, "On-line detection of large-scale parallel applications's structure", *International Parallel & Distributed Processing Symposium (IPDPS 2010)*, Atlanta, Georgia, April 2010.

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

[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] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller, "A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet", *International Workshop on Scalable Tools for High-End Computing (STHEC 2008)*, Kos, Greece, June 2008.

[9] "Open|SpeedShop", http://www.openspeedshop.org/.

[10] Paradyn Project, "MRNet API Programmer's Guide, Release 3.0", http:///www.paradyn.org/mrnet/release-3.0/APIGuide.pdf.

[11] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools", *Supercomputing 2003 (SC'03)*, Phoenix, AZ, November 2003.

[12] Philip C. Roth and Jeffrey S. Vetter, "Scalable Tool Infrastructure for the Cray XT Using Tree-Based Overlay Networks", *Cray User Group (CUG 2009)*, Atlanta, Georgia, May 2009.