Debugging Support for Charm++

Rashmi Jyothi, Orion Sky Lawlor, L. V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
jyothi@uiuc.edu, olawlor@acm.org, kale@cs.uiuc.edu

Abstract

This paper describes a parallel debugger and the related debugging support implemented for CHARM++*, a data-driven parallel programming language. Because we build extensive debugging support into the parallel runtime system, applications can be debugged at a very high level.*

1 Introduction

Parallel programming is more complicated than serial programming because of concurrency, nondeterminacy, and the sheer complexity of modern parallel programs. Debugging tools can help programmers by untangling concurrent execution, controlling nondeterminate messaging, and examining the dynamic state information of the parallel program. [3, 11, 12]

We believe the parallel runtime system is in a unique position to extract useful debugging and program analysis information. Because the runtime system manages all communication and directs control flow, it can present this information in a more useful form than a low-level sequential debugger such as *gdb*.

In this paper, we present a selection of parallel debugging techniques that overcome the shortcomings of existing sequential debugging schemes with a parallel program. Our goal is to provide an integrated debugging environment which allows the programmer to examine and understand the changing state of the parallel program during the course of its execution. As such, we present little brand new work here; but instead present an integrated, orthogonal environment in which these well-known techniques can be put into practice.

1.1 Prior work

The single most well-used debugging method, especially in the primitive runtime environments common to parallel machines, is the insertion of write statements into the code to log specific variables and important events. This method's popularity comes from its simplicity, and the fact that it requires no additional software or training. Nevertheless, the programmer must decide in advance which variables to print and where to insert the output statements, and adding new output statements translates to editing and compiling the program over again. Finding the one piece of critical information hidden in a large output log can be painfully frustrating. Logging in parallel is even more difficult, because network and buffering delays can reorder log statements, resulting in bizarre logs where effects sometimes precede their causes.

Traditional sequential debuggers can deal quite well with a single flow of control using the usual array of step commands, breakpoints, and data structure displays. Sequential debuggers, and sequential debugging tools, are still helpful in debugging the individual processes in a parallel program; but their single-process view of the program ignores concurrent accesses, so debugging message passing or concurrency related bugs is quite difficult.

There are a huge number of research parallel debuggers of varying quality, and a small smattering of commercial debuggers, of which TotalView is a well known example. Hooks for TotalView are available to directly examine the message queues of many MPI implementations[2]; but little additional runtime support is available for this debugger. In addition, the price of the debugger, being nonzero, is beyond the software budget for many small clusters.

Finally, CHARM++ already had a parallel debugger[15], but due to various shortcomings we will describe, the debugger was difficult to use on real applications.

2 Charm++

CHARM++[10, 8, 9] is an object oriented parallel programming language based on C++. CHARM++ is built on Converse [7], a message-passing layer that supports multilingual interoperability. CHARM++ supports a variety of

distributed and shared memory machines, and directly supports Linux or Windows clusters of PCs connected using Ethernet or Myrinet, Alpha servers using Quadrics interconnects, SGI shared-memory machines, the Cray T3E, or any machine using MPI or pthreads.

The execution model of CHARM++ is message-driven [4] wherein Converse treats the parallel machine as a collection of nodes that communicate primarily via messages. Each node is comprised of a number of processors that share memory. When a message arrives at a processor it triggers the execution of a handler function as specified by the message[14]. The message is a contiguous sequence of bytes and has two parts - the header and the data. The header contains a handler number which specifies which handler function is to be executed when the message arrives. Converse maintains a table mapping handler numbers to function pointers. Each processor has its own copy of the mapping.

Communication primitives send messages to the scheduler queues of remote processors, where the scheduler thread finds them and processes them. The Converse scheduler serves not only as a message receiver but also as a central allocator of CPU time. Both locally generated as well as messages from the network contend for scheduling time in the same way.

The parallel programming model of CHARM++ is based on the concept of processor virtualization [6], where the programmer divides the work into a large number of pieces called virtual processors or parallel objects, and lets the runtime system map these pieces to processors. Communication between pieces is based on virtual addresses managed by the runtime system [10], so the system can migrate pieces of the computation between processors without changing the way the pieces communicate, and hence without changing the programmer's view of the computation. The number of pieces a computation is broken into is typically independent of, and normally much larger than, the number of processors. The pieces of the computation are implemented by the programmer as parallel objects, which in CHARM++ are regular C++ objects. As regular C++ objects, CHARM++ parallel objects can contain public and private data and methods as usual.

A machine-generated "proxy" C++ object is used to invoke methods on these parallel objects from other processors. As with Smalltalk, we use the term "send an object a message" to refer to remote object method invocation via this proxy object. In accord with the message-driven execution model of CHARM++ all computations are initiated in response to messages being received. Method calls in CHARM++ are non-blocking—they are asynchronous method invocations [13], so the caller does not wait for the method to be executed or return a value. Because these remote methods can be called from "outside", they are called *entry methods* or *entry points*.

In CHARM++ parallel objects are normally stored in an *Array* [10]. An *Array* is a collection of parallel objects keyed by an "array index". The size of the array is not fixed, and not constrained in any way by the features of the underlying parallel machine such as the number of processors or nodes. Each array element of an array has a globally unique index, and messages are addressed to that index. Most of the data in CHARM++ programs is stored in array elements.

2.1 Existing debugging support

Syncprint

The simplest debugging support provided by CHARM++ is an ordered parallel logging facility, enabled by the command-line parameter "*+syncprint*". This forces causality by making output statements using *printf* block the calling object until the output is queued at a central location. This global ordering slows down output, but ensures no out-of-order debugging statements.

Standalone mode

Another simple feature is the ability to run a parallel program serially, in a single process. This "stand-alone" mode allows programmers to debug Charm++ programs on their local workstation using their favorite serial debugger, such as the graphical debugger included with Microsoft's Visual C++. Because of the virtualization aspect of CHARM++, programs on a single processor are not limited to using a single object or flow of control, so this trivial feature can be used for real programs and has allowed us to catch a number of bugs. There is no true concurrency with this method, however, as CHARM++ switches between in-process flows of control in a cooperative fashion.

Multiple sequential debuggers

We begin to track down concurrent bugs by spawning separate sequential debugger, such as *gdb* or *dbx* on each process of a parallel job using the command-line run-time option "*++debug*". Each debugger runs in a separate window, and shows the terminal output of its parallel process. Because of all the separate windows, this method becomes unusable for more than a few dozen processors.

Record and replay

Bugs due to message ordering can be extremely difficult to track down, because message ordering on many parallel machines is nondeterministic [16, 1]. CHARM++ provides a "record and replay" mechanism that allows a user to record and later reproduce a program's order of message

arrivals, which can help catch message ordering bugs. The key idea here is to tag messages at the sender, and record the message execution order to a file using the sender-generated tags. CHARM++ tags messages using the sending processor and an "outgoing message count" sequence number. This means the same message executions can be replayed by processing incoming messages in file order, as long as senders tag their messages the same way on re-execution. Because CHARM++ scheduling is deterministic and non-preemptive, the only nondeterminism in CHARM++ programs comes from message arrival order. Thus we only need to ensure senders also process incoming messages in file order to ensure the entire program repeats itself exactly.

To enable the required tracing for record and replay, a CHARM++ program is linked with the option "*-tracemode recordreplay*" and run with the "*+record*" option, which records message orders in a file for each processor. The same execution order can be replayed using the "*+replay*" runtime option; which can be used at the same time as the other debugging tools in CHARM++.

3 Integrated debugging system

We have created a new debugging system with a number of useful features for CHARM++ programmers. The system includes a Java GUI client program which runs on the programmer's desktop, and a CHARM++ parallel program which acts as a server. The client and server need not be on the same machine, and communicate over the network using a secure protocol described in Section 4.2.

The system provides the following new features.

- Provides a means to easily access and view the major programmer visible entities, including array elements and messages in queues[15], across the parallel machine during program execution. Objects and messages are extracted using the CHARM++ PUP framework described in Section 4.1.

- Provides an interface to set and remove breakpoints on remote entry points[13], which capture the major programmer-visible control flows in a CHARM++ program.

- Provides the ability to freeze and unfreeze the execution of selected processors of the parallel program, which allows a consistent snapshot by preventing things from changing as they are examined.

- Provides a way to attach a sequential debugger to a specific subset of processes of the parallel program during execution, which keeps a manageable number of sequential debugger windows open.

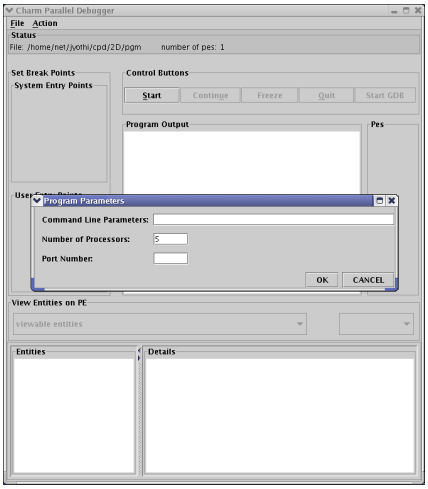

Figure 1. Using the menu to set parameters for the CHARM++ program being debugged

The debugging client provides these features via extensive support built into the CHARM++ runtime. The parallel runtime is in a unique position to provide this debugging information, as it is much closer to the application level than the machine binary used by sequential debuggers.

3.1 Example usage

The CHARM++ programmer starts the debugger client from the command-line specifying the program to be debugged, its parameters and the number of processor elements it should run on as command-line parameters. Alternatively, the program and the parameters could be set via a menu item provided by the debugger GUI. The menu usage is shown in Figure 1.

Once the debugger client's GUI loads, the programmer triggers the program execution by clicking the *Start* button. The program begins in a frozen state, displaying the "user" and "system" entry points as a list of check boxes. "System" entry points belong to libraries and CHARM++ code, while "user" entry points are defined in the application program being debugged. The programmer sets and removes breakpoints by checking and unchecking the checkboxes corresponding to the entry points, then begins execution by clicking the *Continue* Button. The program freezes when a breakpoint is reached. Figure 2 shows a snapshot of the debugger when a breakpoint is reached.

The server runtime inserts a breakpoint by changing the CHARM++ entry handler table, a table of function pointers that normally directly jump to application entry method code. By overwriting the entry method's function table entry with a jump to debugging code, the next message which attempts to execute that method will instead jump directly

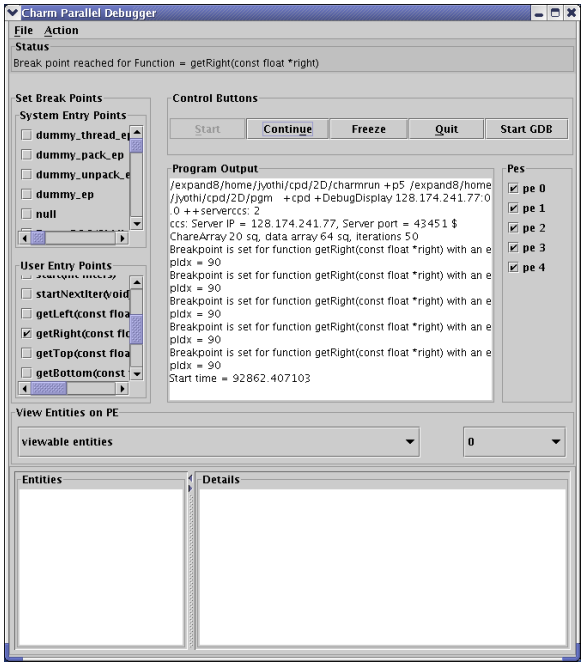

Figure 2. Parallel debugger when a break point is reached

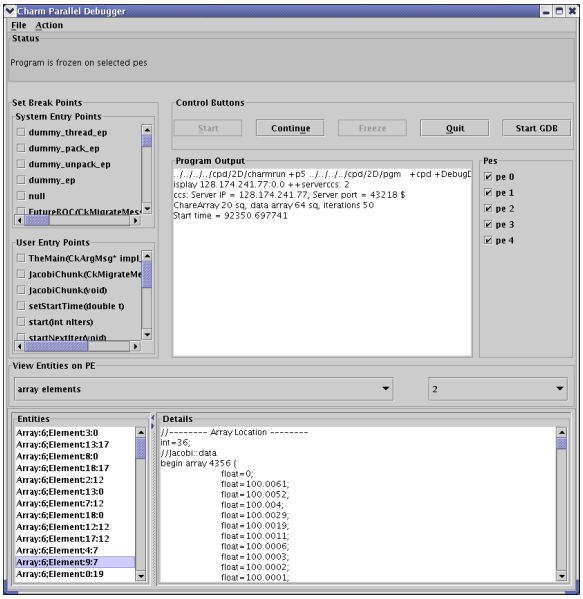

Figure 3. Viewing the contents of a parallel object

to the debugging runtime. This is a much more efficient implementation than our previous version[15], which kept a list of breakpoints to check against each incoming message. In fact, the new version imposes zero overhead if not used, so it can be permanently enabled rather than requiring a special debug build.

Clicking the *Freeze* button stops the selected processors before the start of their next message, and drains their network queues. The *Continue* button resumes execution. The *Quit* button exits the debugged program.

Entities (for instance, array elements) and their contents on any processor can be viewed at any point, as illustrated in Figure 3. The CHARM++ PUP framework, as described in Section 4.1, is used to retrieve and format program entities. The Converse scheduler, which is the core of CHARM++, interacts with a pool of messages placed in queues on each of the processors[7]. These messages could be generated locally or could be from remote processors. In CHARM++, a message could be due to an entry method invocation, a ready thread, a message sent to a ready thread or a handler posted previously. A message is a chunk of memory with a header and data. The debugger allows the user to freeze the program and inspect the messages in the queues. From the data part of the CHARM++ message the debugging framework encodes the destination object, the method being invoked and the parameters for the user to interpret.

Specific individual processes of the CHARM++ program

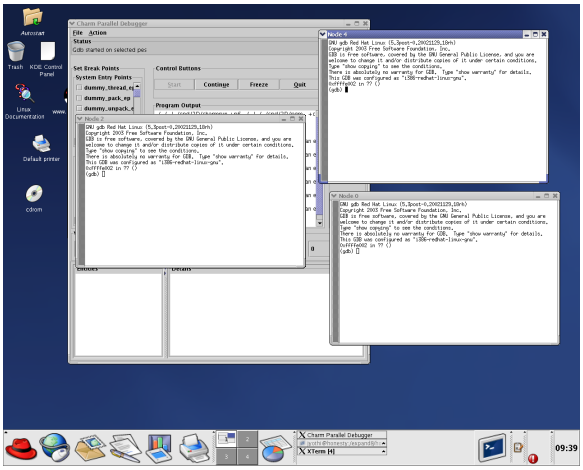

Figure 4. Parallel debugger showing *gdb* running for three processors

can be attached to instances of *gdb* during the course of program execution as shown in Figure 4. The older "*++debug*" option provides the same ability, but it always starts a sequential debugger for every process, while the new interface can start the debugger on a subset of processors.

4 Implementation

Our parallel debugger GUI interface, written using Java, connects to the parallel program across the network using a protocol called Converse Client/Server (CCS), as described in Section 4.2. To extract program state, it calls a debugging CCS handler described in Section 4.3, which traverses and sends runtime and user objects using the PUP framework described next.

4.1 CHARM++ PUP framework

The PUP framework is a method to describe the in-memory layout of an object, and was originally designed to support object migration in CHARM++. To copy a complicated object from one processor to another, we must pack the object into a network message, ship the message to another processor, and finally unpack the message into an object on the other side. This is an extremely common operation, used in Java RMI serialization/deserialization, parameter marshalling and unmarshalling for CORBA communication, and even MPI derived datatypes. PUP stands for Pack/UnPack, and is a compact, efficient, and flexible method to perform this packing and unpacking of user objects for C++.

Because of the type safety and introspection capabilities of the Java language and virtual machine, Java can pack and unpack arbitrary objects automatically, without any further effort. CORBA requires the user to describe the format of each communicated object in a CORBA IDL file, which is preprocessed to generate pack and unpack code. MPI requires the user to build a "derived datatype" at runtime, using type construction library calls that list each field of each communicated object; because this is complicated, users often write explicit packing and unpacking code to ship complicated objects.

CHARM++ originally required users to write an explicit pack and unpack routine for each object, as well as a size routine to determine the outgoing message size before packing. The motivation for PUP is that the code used to size the message, pack an object into a message, and unpack an object from a message must match up exactly—everything that is packed must be unpacked, and vice versa. Writing three interrelated routines for every object is tedious, error-prone, and contributes to the burden of parallel programming.

In the PUP framework, sizing, packing, and unpacking are all is controlled by a single user-written subroutine called a *pup* routine. The *pup* routine simply calls a virtual method on each of the object's fields, which are then sized, packed or unpacked as appropriate.

Consider a very simple C++ class with three fields:

```
class foo {
    int A;
    float B;
    long C;
public:
    ...
};
```

We define an abstract class named "PUP::er" with one virtual method named "bytes", which takes the address of an object field and a description of the type of data in the field. A *pup* routine for foo would then just pass each of the foo object's fields into the PUP::er.

```
void foo::pup(PUP::er &p) {
    p.bytes(&A,MPI_INT);
    p.bytes(&B,MPI_FLOAT);
    p.bytes(&C,MPI_LONG);
}
```

Because the PUP::er is given the address and data type of each of the objects' fields, it can perform arbitrary manipulations of those fields, including copying data into the fields, copying data out of the fields, or even building an MPI derived datatype using the field offsets.

```
// Compute total size of object fields
class SIZING_PUP_er : public PUP::er
{
public:
   int totalsize; // size of object
   SIZING_PUP_er() {totalsize=0;}

   virtual void
    bytes(void *field,int datatype) {
      totalsize+=size(datatype);
    }
};

// Copy data out of object fields
... define destbuf as PUP::er field ...
void PACKING_PUP_er::
bytes(void *field,int datatype) {
    memcpy(destbuf,field,size(datatype));
    destbuf+=size(datatype);
}
```

```
// Copy data into object fields
... define srcbuf as PUP::er field ...
void UNPACKING_PUP_er::
bytes(void *field,int datatype) {
    memcpy(field,srcbuf,size(datatype));
    srcbuf+=size(datatype);
}

// Build an MPI derived datatype
void MPI_DATATYPE::
bytes(void *field,int datatype) {
   displacements[n]=field-objbase;
   datatypes[n]=datatype
   n++;
}
```

It should be clear the very simple technique of calling a virtual method for each field of an object is quite powerful. CHARM++ actually uses the first three PUP::ers above to size network messages and copy data into and out of objects as they are sent across the network. The overhead for using the very general pup method to do the copy is exactly one virtual function call per field, which on many machines is faster than the memory copy itself. Other PUP::ers, not shown here, can read and write objects to and from disk, or even convert binary data formats between different machine architectures.

PUP operator

However, the application code must tediously pass to the "bytes" routine the address and datatype of each field of each object. Luckily, we can use C++ operator overloading to automatically extract the datatypes, and also to provide a simpler syntax:

```
void operator|(PUP::er &p,int &x)
{ p.bytes(&x,MPI_INT); }
void operator|(PUP::er &p,float &x)
{ p.bytes(&x,MPI_FLOAT); }
... and so on for other datatypes ...

void foo::pup(PUP::er &p) {
    p|A; // calls p.bytes
    p|B;
    p|C;
}
```

Users can treat operator| as a builtin operator, analogous to the << and >> C++ iostream operators. This operator overloading also provides a surprising benefit: we can now use the same syntax to pup user-defined classes that we use for builtin types like "int".

```
void operator|(PUP::er &p,foo &x)
    { x.pup(p); }

class bar {
    int I;
    foo F;
    ...
}
void bar::pup(PUP::er &p) {
    p|I;  // calls p.bytes
    p|F;  // calls foo::pup
}
```

Notice how C++'s operator overloading selects the appropriate way to pup the two fields I and F, even though the operator| call looks identical.

Operator overloading can also be applied to pup templated classes, with the template type determined by C++ type resolution. For example, we can easily define a pup operator for the standard C++ class std::vector. The elements of the vector are pup'd using their own pup operator, so they can be of any type.

```
template<class T>
void operator|(PUP::er &p,
               std::vector<T> &v)
{
    int length=v.size();
    p|length;
    p.resize(length);
    for (int i=0;i<length;i++)
        p|v[i];
}
```

This std::vector pup operator shows some of the strange beauty of using a single routine for both packing and unpacking. While packing, the length of x is known, the "p|length" call stores the length, and the "resize" call specifies the current size and hence does nothing. While unpacking, the length is initially zero, "p|length" extracts the true length, and the "resize" call actually allocates space in the vector for the new elements.

Because operator overloading follows the type system, we can now pup an array of ints, std::vector<int>; or a 2D array of foo objects, std::vector< std::vector<foo>>, using the same "p|x" syntax used to pup plain ints. CHARM++ includes builtin pup operators for std::vector, std::list, std::string, std::map, and std::multimap, templated over any object with a pup operator. PUP thus uses C++'s sophisticated type and template overloading system to approach the true type introspection ability of Java.

PUP field names

One final modification to the PUP::er syntax provides not only the value and data type of the fields, but the human-readable names of fields as well. This uses a macro to turn the field name into a string, which is passed to another PUP::er virtual method "fieldName", then pups the data using operator| as usual.

```
#define PUP(field) \
      p.fieldName(#field); \
      p|field;

void foo::pup(PUP::er &p) {
    PUP(A); // calls p.fieldName("A")
    PUP(B); //   then p.bytes
    PUP(C);
}
```

Most PUP::ers ignore the field names, but CHARM++ has several PUP::ers that use the field names to read and write objects from keyword/value ASCII files. Finally, a debugging PUP::er can send the annotated object data off to the parallel debugger for display. For parallel objects, our debugging support by default calls the same pup routine as is used for migration, but also provides a special "ckDebugPup" pup routine that can be used to make debugging-specific data available via pup.

Other PUP::er features allow for dynamically allocated data (which must be allocated during the unpack phase), the ability to easily pup a pointer-to-subclass, and pup routines written in C or Fortran. See the CHARM++ manual[13] for details.

Because the support for PUP is built into the runtime system and, for networking, always built into the application, there is no need to compile the application with '-g' (unless also using a sequential debugger). This means our parallel debugger can be used to examine the internal state of the optimized, production version of an application.

4.2 CCS network interface

The Converse Client-Server (CCS) network interface [7] enables Converse (and hence CHARM++) programs to act as parallel servers, responding to requests from the network. The server side of this interface is built into every CHARM++ program, and the client side is provided as a library for C and Java.

A CCS client, in this case the parallel debugger, connects to the server via a TCP connection and sends it a request, which consists of a string handler name and a block of binary request data. The CHARM++ runtime uses the handler name to look up and call the appropriate handler function from an extensible table. For example, when the parallel debugger sends the request name "ccs_set_break_point", the runtime executes a handler that installs a breakpoint. After the server has processed the request, it responds with a block of binary response data. This simple request/response protocol allows information to be injected into and extracted from a running parallel program.

Because the client opens the TCP connection for a CCS request, CCS can be used by clients behind firewalls or NAT routers. When CCS is running over the unsecured internet, it can be run in a secure authentication mode[14], which uses a SHA-1 hash of the request, a nonce, and a shared secret key for authentication. Authentication prevents arbitrary users from injecting messages, but because of export regulations we do not provide network encryption. If secrecy is also important, users can also add encryption.

4.3 Debugging access via CCS

The CHARM++ runtime provides a special CCS handler to extract formatted information about the entities in the parallel program. The CCS handler allows lists of objects to be registered, and provides a way to call the objects' pup routines and extract formatted information about the object structure. Various parts of the runtime system register the different classes of objects, including application parallel objects and network messages, with this single CCS handler. This allows the debugger to access these different objects in a uniform manner. CHARM++ applications or libraries can also register more detailed information, which can then be presented by the debugger.

Because this method uses the PUP framework, which CHARM++ applications already support for migration, zero additional code must be written to use an application in the debugger. The is both easier to use as well as more powerful than our previous debugger[15], which required a special "debugging display routine" in each object and even then could only display flat ASCII text.

5 Conclusions and future work

The debugging solutions presented in this work provide a number of useful features for CHARM++ programmers. The parallel debugger allows the programmer to inspect the state of a running parallel program at a very high level, including the contents of array elements and network messages. It allows the programmer to follow the control flow in the parallel program by setting break points at entry methods. It provides the programmer a more focused means for switching to sequential debugging on selected processors on the fly. The record and replay mechanism allows the programmer to deterministically reproduce a program's behavior.

In the future, we will use the CHARM++ PUP framework to provide better, higher-level views of objects and messages. In particular, it should be possible to support extensive analysis and visualization of the multidimensional arrays so common in scientific computing. We will extend the functionality of the parallel debugger with the capabilities of our performance analysis tool for CHARM++, Projections [5]. Projections includes detailed tracing, network and performance statistics, together with automated analysis, that could be very useful while debugging or performance tuning. The field of parallel debugging for novel runtime systems is virtually wide open, and there an immense amount of work to be done.

5.1 Acknowledgements

This work was supported in part by the National Science Foundation (NSF NGS 0103645) and the local Department of Energy ASCI center, CSAR (DOE B341494).

References

[1] T. J. Blanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[2] J. Cownie and W. Gropp. A standard interface for debugger access to message queue information in MPI. In *PVM/MPI*, pages 51–58, 1999.

[3] J. Cunha, J. Lourenco, and T. Antao. A debugging engine for parallel and distributed environment. In *Proceedings of 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 111–118, Miskolc, Hungary, 1996.

[4] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[5] L. Kalé and A. Sinha. Projections : A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Sympos ium*, pages 108–114, Apr. 1993.

[6] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[7] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[8] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[9] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[10] O. S. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[11] J. May and F. Berman. Panorama: A portable, extensible parallel debugger. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 96–106, San Diego, California, 1993.

[12] J. May and F. Berman. Designing a parallel debugger for portability. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 909–915, 1994.

[13] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *The Charm++ Programming Language Manual, Version 6.0*, Jan 2004.

[14] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *Converse Programming Manual*, Jan 2004.

[15] P. Ramachandran and L. V. Kalé. Mulitlingual debugging support for data-driven and thread-based parallel languages. In *Lecture Notes in Computer Science: Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC '99)*, pages 236–250. Springer-Verlag, August 1999.

[16] A. Sinha. *Performance Analysis of Object Based and Message Driven Programs*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, January 1995.